

MOOC Python

Corrigés de la semaine 4

pgcd - Semaine 4 Séquence 3

```
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     # boucle sans fin
9     while True:
10        # on calcule le reste
11        r = a % b
12        # si le reste est nul, on a terminé
13        if r == 0:
14            return b
15        # sinon on passe à l'itération suivante
16        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     while True:
11         # on calcule le reste
12         r = a % b
13         # si le reste est nul, on a terminé
14         if r == 0:
15             return b
16         # sinon on passe à l'itération suivante
17         a, b = b, r
```

distance - Semaine 4 Séquence 6

```
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]))
```

doubler_premier - Semaine 4 Séquence 6

```
1 def doubler_premier(f, first, *args):
2     """
3     renvoie le résultat de la fonction f appliquée sur
4     f(2 * first, *args)
5     """
6     # une fois qu'on a écrit la signature on a presque fini le travail
7     # en effet on a isolé la fonction, son premier argument, et le reste
8     # des arguments
9     # il ne reste qu'à appeler f, après avoir doublé first
10    return f(2*first, *args)
```

doubler_premier (bis) - Semaine 4 Séquence 6

```
1 def doubler_premier_bis(f, *args):
2     "marche aussi mais moins élégant"
3     first = args[0]
4     remains = args[1:]
5     return f(2*first, *remains)
```

doubler_premier_kwds - Semaine 4 Séquence 6

```
1 def doubler_premier_kwds(f, first, *args, **keywords):
2     """
3     équivalent à doubler_premier
4     mais on peut aussi passer des arguments nommés
5     """
6     # c'est exactement la même chose
7     return f(2*first, *args, **keywords)
8
9 # Complément - niveau avancé
10 # ----
11 # Il y a un cas qui ne fonctionne pas avec cette implémentation,
12 # quand le premier argument de f a une valeur par défaut
13 # *et* on veut pouvoir appeler doubler_premier
14 # en nommant ce premier argument
15 #
16 # par exemple - avec f=muln telle que définie dans l'énoncé
17 #def muln(x=1, y=1): return x*y
18
19 # alors ceci
20 #doubler_premier_kwds(muln, x=1, y=2)
21 # ne marche pas car on n'a pas les deux arguments requis
22 # par doubler_premier_kwds
23 #
24 # et pour écrire, disons doubler_permier3, qui marcherait aussi comme cela
25 # il faudrait faire une hypothèse sur le nom du premier argument...
```

compare_args - Semaine 4 Séquence 6

```
1 def compare_args(f, g, argument_tuples):
2     """
3     retourne une liste de booléens, un par entree dans entrees
4     qui indique si f(*tuple) == g(*tuple)
5     """
6     # c'est presque exactement comme compare, sauf qu'on s'attend
7     # à recevoir une liste de tuples d'arguments, qu'on applique
8     # aux deux fonctions avec la forme * au lieu de les passer directement
9     return [f(*tuple) == g(*tuple) for tuple in argument_tuples]
```