

# W4-S8-E2-passage-arguments

December 15, 2014

## 1 Passage d'arguments

### 1.1 Exercice - niveau basique

Vous devez écrire une fonction `distance` qui prend un nombre quelconque d'arguments numériques non complexes, et qui retourne la racine carrée de la somme des carrés des arguments.

Plus précisément :  $distance(x_1, \dots, x_n) = \sqrt{\sum x_i^2}$

Par convention on fixe que  $distance() = 0$

```
In []: # des exemples
      from corrections.w4_fun_args import exo_distance
      exo_distance.exemple()

In []: # ATTENTION vous devez aussi définir les arguments de la fonction
      def distance(votre, signature):
          "<votre code>"

In []: # la correction
      exo_distance.correction(distance)
```

### 1.2 Exercice - niveau intermédiaire

On vous demande d'écrire une fonction qui prend en argument \* une fonction `f`, dont vous savez seulement que le premier argument est numérique, et qu'elle ne prend **que des arguments positionnels** (sans valeur par défaut) \* un nombre quelconque - mais au moins 1 - d'arguments positionnels `args`, dont on sait qu'ils pourraient être passés à `f`

Et on attend en retour le résultat de `f` appliqués à tous ces arguments, mais avec le premier d'entre eux multiplié par deux.

Formellement:  $doubler\_premier(f, x_1, x_2, \dots, x_n) = f(2 * x_1, x_2, \dots, x_n)$

```
In []: # quelques exemples de ce qui est attendu.
      # add et mul sont les opérateurs binaires du module operator,
      # soit l'addition et la multiplication respectivement.
      # distance est la fonction de l'exercice précédent.
      from corrections.w4_fun_args import exo_doubler_premier
      exo_doubler_premier.exemple()

In []: # ATTENTION vous devez aussi définir les arguments de la fonction
      def doubler_premier(votre, signature):
          "<votre_code>"

In []: exo_doubler_premier.correction(doubler_premier)
```

### 1.3 Exercice - niveau intermédiaire

Vous devez maintenant écrire une deuxième version qui peut fonctionner avec une fonction quelconque (elle peut avoir des arguments nommés avec valeurs par défaut).

La fonction `doubler_premier2` que l'on vous demande d'écrire maintenant prend donc un premier argument `f` qui est une fonction, un second argument positionnel qui est le premier argument à `f` (et donc qu'il faut doubler), et le reste des arguments à `f`.

```
In []: # quelques exemples de ce qui est attendu
      # avec ces deux fonctions

      def addn(x, y=0):
          return x + y

      def muln(x=1, y=1):
          return x * y

      from corrections.w4_fun_args import exo_doubler_premier2
      exo_doubler_premier2.exemple()
```

Vous remarquerez que l'on n'a pas mentionné dans cette liste d'exemples

`doubler_premier2 (muln, x=1, y=1)`

que l'on ne demande pas de supporter puisqu'il est bien précisé que `doubler_premier` a deux arguments positionnels.

```
In []: # ATTENTION vous devez aussi définir les arguments de la fonction
      def doubler_premier2(votre, signature):
          "<votre code>"
```

```
In []: exo_doubler_premier2.correction(doubler_premier2)
```

### 1.4 Exercice - niveau avancé

#### 1.4.1 validation revisitée

Nous avons déjà fait un peu plus tôt cette semaine, au sujet de la programmation fonctionnelle, un exercice au sujet d'une fonction `validation` qui comparait le résultat de deux fonctions, toutes deux à un argument, sur une liste de valeurs d'entrée.

Nous reprenons ici la même idée, mais en levant l'hypothèse que les deux fonctions attendent un seul argument. Il faut écrire une nouvelle fonction `validation2` qui prend en entrée \* deux fonctions `f` et `g` comme la dernière fois, \* mais cette fois une liste (ou un tuple) `argument_tuples` de `tuples` d'arguments d'entrée

Et comme la fois précédente on attend en retour une liste `retour` de booléens, de même taille que `argument_tuples`, telle que, si `len(argument_tuples)` vaut  $n$ :

$\forall i \in \{1, \dots, n\}$ , si `argument_tuples[i] == [ a1, ..., aj ]`, alors  
`return(i) == True`  $\iff$  `f (a1, ..., aj) == g (a1, ..., aj)`

```
In []: from corrections.w4_fun_args import exo_validation2
      exo_validation2.exemple()
```

```
In []: # ATTENTION vous devez aussi définir les arguments de la fonction
      def validation2(votre, signature):
          "<votre_code>"
```

```
In []: exo_validation2.correction(validation2)
```

Pour information: `* factorial` correspond à `math.factorial` `* fact` et `broken_fact` sont des fonctions implémentées par nos soins, la première est correcte alors que la seconde retourne 0 au lieu de 1 pour l'entrée 0. `* add` correspond à l'addition binaire `operator.add` `* plus` et `broken_plus` sont des additions binaires écrites par nous, l'une étant correcte et l'autre étant fausse lorsque le premier argument est nul.