# Guide d’utilisation du package « outils de controles »

Le package « outils de controles » permet d'avoir des moyens de haut niveaux pour contrôler et vérifier que certaines conditions sont remplies, dans l'objectif d'éviter que votre programme ne lève pas des erreurs en cours de processus.
Par exemple, en contrôlant les arguments à l'appel de la méthode, votre programme ne commence pas à modifier l'objet alors qu'il risque de pas pouvoir finir les modifications attendues.

Ce package contient plusieurs classes qui vérifient que des objets sont valides :
● Verificateur qui vérifie qu'un objet non conteneur respecte certains critères (type, minimum, maximum, …)
● VerificateurStr qui vérifie qu’une chaîne respecte certains critères (longueur minimum, maximum, correspondance avec une expression régulière). Hérite de Verificateur.
● VerificateurConteneurs qui vérifie que les objets conteneurs sont composés d'éléments valides. (Pour chaque élément attendu, il est fourni un ou des identifiants ainsi qu'un objet verificateur.)
● VerificateurListes qui vérifie qu'une liste est bien valide : on contrôle le nombre d’éléments de la liste et les éléments de la liste. Hérite de VerificateurConteneurs.
● VerificateurDict qui vérifie qu'un dictionnaire est bien conforme vis-à-vis de l'objet attendu. Hérite de VerificateurConteneurs. *(classe prévue pour une version ultérieure)*
● VerificateurStr qui vérifie qu’une chaîne de caractère est bien valide. Hérite de Verificateur.
● VerificateurTableaux qui vérifie que des tableaux (c'est à dire une liste contenant d'autres listes, qui forme un tableau bi-dimensionnel) sont bien valides. Hérite de VerificateurListes.
● VerificateurArgumments qui vérifie que les arguments fournis sont bien valides. Hérite de VerificateurConteneurs.

Ce package contient aussi un module avec des décorateurs. Parmi eux, on trouve le décorateur « controle_types » qui vérifie que les arguments passées à des fonctions ou méthodes sont du bon type.
Un autre décorateur (« controle_arguments ») permet que les arguments vérifient tous les critères de validités définis par les objets vérificateurs.
Un troisième décorateur (« controle_temps_execution ») peut vérifier qu’une fonction (ou une méthode) s’exécute suffisamment rapidement.

Pour plus d’informations sur le fonctionnement et les caractéristiques techniques de ce package, voir le fichier « Documentation technique outils de controles ».

## Utilisation des classes

Les objets des classes citées ci-dessus, en plus de pouvoir les passer en paramètres aux décorateurs, peuvent vérifier les objets fournis à l’aide de leurs propres méthodes.

### *Utilisation de la classe Verificateur*

Tout d'abord, chaque objet de la classe Verificateur permet de contrôler un objet, en principe non conteneur – c'est-à-dire qui ne contient pas d'autres objets, de n'importe quel type. (Il est cependant conseillé d'utiliser la classe du package la plus adaptée au type de l'objet dont on veut évaluer la validité afin de contrôler plus précisément l'objet). La classe Verificateur est la plus générique des classes du package et permet notamment de contrôler les entiers, flottants, booléens.

Un objet Verificateur permet de contrôler le type de l'objet (via l'attribut types, passer en paramètre un type ou un tuple contenant plusieurs types), le minimum valide de l'objet (valeur au-dessous de laquelle l'objet n'est plus valide) et le maximum valide.

Pour vérifier qu'un entier correspond bien à une heure de la journée, on peut créer un objet Verificateur grâce à :

```
V1 = Verificateur(types=int, minimum=0, maximum=23)
```

### *Utilisation de la classe VerificateurStr*

La classe VerificateurStr, comme son nom l'indique, sert à contrôler les chaînes de caractères. Cette classe vérifie systématiquement que l'objet contrôlé est bien une chaîne de caractères. Mais cette classe ne se limite pas à cela : les deux attributs minimum et maximum permettent de contrôler la longueur de la chaîne en précisant deux nombres entiers positifs. On peut également fournir une expression régulière pour l'attribut regex pour vérifier que la chaîne contrôlée correspond bien à cette expression.

Par exemple, pour vérifier que la chaîne est une adresse mail, on peut saisir :

```
VS1 = VerificateurStr(regex=r"^[A-Za-z0-9\.-]+\@[A-Za-z0-9-]+\.[a-z]{2,3}$")
```

Et pour vérifier un numéro de téléphone :

```
VS2 = VerificateurStr(10, 17, r"^((\+33[ -]?)|0)[1-9]([ -]?[0-9]{2}){4}$")
```

### *Utilisation de la classe VerificateurConteneur*

Il est vivement conseillé de ne pas utiliser directement cette classe. En effet, elle a uniquement pour vocation d'être la classe mère des classes qui contrôlent des conteneurs. Ainsi, la classe VerificateurConteneurs n'est adaptée à aucun conteneur en particulier. (S'il n'existe pas de classe contrôlant un type de conteneur, il est souhaitable, plutôt que d'utiliser cette classe, de la dériver en créant un classe fille correspondant à ce type de conteneur.) Ainsi, certaines méthodes de VerificateurConteneurs lèvent des erreurs indiquant qu'elles ne sont volontairement pas définies – c'est aux classes filles de les définir. Si vous utilisez directement cette classe, vous risquez d'avoir des erreurs lors de la vérification de la validité du conteneur : votre programme risque de planter.

### *Utilisation de la classe VerificateurListes*

La classe VerificateurListes permet de vérifier la taille et le contenu de listes (et éventuellement de tuples). Tout d'abord les attributs minimum et maximum permettent de contrôler la taille de la liste. Ensuite, on peut ajouter des vérificateurs pour contrôler les éléments de la liste.

### *Utilisation de la classe VerificateurTableaux*

*---Texte à venir dans une prochaine version de la documentation---*

### *Utilisation de la classe VerificateurArguments*

La classe VerificateurArguments permet de vérifier les arguments passés en arguments d'une fonction ou d'une méthode. Tout d'abord les attributs minimum et maximum permettent de contrôler le nombre d'arguments qui peuvent être passés à la fonction/méthode. Ensuite, on peut ajouter des vérificateurs pour contrôler les différents arguments. Pour cela, on utilise la méthode append qui prend 3 arguments. En premier, on indique l'ordre de définition de l'argument. Par exemple, pour contrôler le premier argument de la fonction on met 0. Si on contrôle un argument qui n'est pas positionnel, on met None. Le deuxième argument de append est le nom de l'argument. Le troisième est le vérificateur associé à l'argument contrôlé.

Exemple d'utilisation des objets vérificateurs :

```
VL = VerificateurListes(types=list, minimum=2)
VL.append(0,3, Verificateur(float, 0))
VA = VerificateurArguments([0,'liste', VL],[1,'b', Verificateur((int, float))])
VA.append(nom=i, Verificateur(int))

@controle_types(VA)
def ma_fonction(a,b,i=0) :
        a=a[1 :-1]
        return a, b+i
```

## Utilisation de la fonctionnalité « conversion »

Grâce aux package « outils de controles », on peut vérifier si le type d'un objet correspond aux types valides, précisés par l'attribut types. Si l'objet n'est pas d'un type valide, la méthode de contrôle lève une erreur. Cependant, on peut aussi tenter de convertir l'objet dans un type valide.
On utilise cette fonctionnalité via les méthodes de contrôle. Pour cela, lors du contrôle d'un objet, le paramètre conversion doit être vrai. La méthode de contrôle renvoie l'objet valide (qu'il soit le même que celui d'origine ou pas). Par exemple, on peut écrire :

```
V1 = Verificateur(int, 0, 10)
objet_valide = V1.controle_total('5', conversion=True) # objet_valide = 5
```

Ce paramètre est également disponible pour les décorateurs. Le décorateur transmet automatiquement les paramètres corrigés à la fonction/méthode appelée.

## Utilisation des décorateurs

Pour modifier l'exécution d'une fonction ou d'une méthode, il faut écrire juste avant sa définition « @nom du décorateur ». Si le décorateur prend des paramètres, on met des parenthèse après son nom, et on y rajoute les paramètres

Pour « controle_types », on écrit par exemple :

```
@controle_types(float, (str, list), i=int)
def ma_fonction(a,b,i=0) :
        b=b[2 :-3]
        return a+i, b
```

Car dans cet exemple, ma_fonction attend un nombre à virgule pour l'argument a, une liste ou une chaîne pour b et pour l'argument nommé i, il faut un entier.

On peut aussi passer en paramètre au décorateur un objet VerificateurArguments. Utiliser un objet de ce type permet de représenter les critères de validités des arguments de manière plus précise, et d'améliorer la flexibilité d'utilisation de vos fonctions. En effet, si vous n'utilisez pas de d'objet VerificateurArguments, les arguments nommés doivent obligatoirement avoir leur nom précisé lors de l'appel à la fonction et les arguments non nommés ne peuvent pas avoir de nom. (Dans l'exemple précédent, si l'instruction `ma_fonction(3, 'abcdefg', '5')` avait été exécutée, une erreur aurait été levée. De même `ma_fonction(a=3, b='abcdefg', i=5)` n'aurait pas été acceptée.)
De plus grâce aux objets VerificateurArguments, on peut également contrôler les conteneurs en utilisant les autres classes de ce module (voir section précédente).

Pour vérifier qu'une fonction ou méthode s'exécute suffisamment rapidement on peut utiliser le décorateur « controle_temps_execution ». Ce décorateur prend un paramètre optionnel « temps_maximal ». Si la fonction ou méthode met plus de temps_maximal secondes pour s'exécuter, un avertissement est levé. Il indique que le temps d'exécution est trop long et précise le temps d'exécution de la fonction ou méthode. Un deuxième paramètre optionnel est « affichage » qui est un booléen indiquant si le temps d'exécution doit être affiché à chaque fois. (Vaut False par défaut).
Exemple d'utilisation :

```
@controle_temps_execution(temps_maximal=0.05)
def ma_fonction(paramètres) :
```

ou encore, pour une fonction qui parcours des listes, dictionnaires ou tableaux très grands :

```
@controle_temps_execution(temps_maximal=0.5, affichage=True)
def ma_fonction(paramètres) :
```

# Crédits :

**Développeur :** Cyprien BONTRON

**Contact :** c.b.e.python@gmail.com

Pour tous renseignements ou problème de fonctionnement, faites un mail à l'adresse sus citées. Nous ne sommes pas tenu de répondre.



**Remerciements :** Python software foundation

**Documentation de la version :** 2.5.3

**Mise à jour :** Mars 2023

**Référence de la documentation :** ODC0323Cu (Documentation pour Outils De Controle de mars (03) 2023, version C, concernant l'utilisation du package)