

(/dashboard)

 RUN CELL RUN ALL

()



()

(834)

(831)

(/hub/dashboard)

 UNVALIDATE

SAVE

Robin
BIRON

()

DataScientest • com

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

Python avancé

Décorateurs

1. Les Décorateurs

Un **décorateur** en Python peut être vu comme une fonction qui modifie le comportement d'autres fonctions.

En général, on utilise les décorateurs lorsqu'on veut ajouter un certain code à plusieurs fonctions, sans avoir à modifier ces dernières.

Pour appliquer un décorateur à une fonction, on précède la ligne de définition de cette dernière par une ligne comportant un @ puis le nom du décorateur.

Exemple d'implémentation: On cherche un moyen d'introduire les fonctions par leur nom au moment de leur exécution.

```
def print_before_execution(function):  
    def print_then_execute(*args, **kwargs):  
        print('voici ce que renvoie la fonction {}'.  
              format(function.__name__))  
        function(*args, **kwargs)  
    return print_then_execute
```

```
@print_before_execution  
def print_hello_world():  
    print("hello world")
```

```
print_hello_world()
```

Le résultat renvoyé est le suivant

```
>>> voici ce que renvoie la fonction print_hello_  
rld  
hello world
```

- a- Construire un décorateur nommé **affiche_doc** qui affiche la documentation d'une fonction avant de retourner cette dernière.

(/dashboard)



RUN CELL



RUN ALL



La documentation d'une fonction peut être renvoyée avec fonction.doc



(

)



(

)

(834)

(831)

In [2]:

(/hub/dashboard)



(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

UNVALIDATE SAVE

def print_before_execution(function):
 def print_then_execute(*args, **kwargs):
 print('voici ce que renvoie la fonction {}'.format(function.__name__))
 function(*args, **kwargs)
 return print_then_execute

@print_before_execution
 def print_hello_world():
 print("hello world")

print_hello_world()

voici ce que renvoie la fonction print_hello_world
 hello world

In [3]:

Insérer votre code ici

```
def affiche_doc(function):
    def doc_then_execute(*args, **kwargs):
        print(function.__doc__)
        return function(*args, **kwargs)
    return doc_then_execute
```

In [4]:

(/dashboard)

(/hub/dashboard)

(/hub/cheatsheet)

(/hub/progress)

(/typo)

(/forum/271)

```

import pandas as pd
@affiche_doc
def importer_csv(*args, **kwargs):
    '''
    Fonction qui permet d'importer un fichier csv dans un DataFrame pandas
    '''
    return pd.read_csv(*args, **kwargs)
importer_csv('country_vaccinations.csv')
  
```

Fonction qui permet d'importer un fichier csv dans un DataFrame pandas.

Out[6]:

	country	iso_code	date	total_vaccinations	people_vaccinated	people_fully_vaccinated
0	Argentina	ARG	2020-12-29	700.0	NaN	NaN
1	Argentina	ARG	2020-12-30	NaN	NaN	NaN
2	Argentina	ARG	2020-12-31	32013.0	NaN	NaN
3	Argentina	ARG	2021-01-01	NaN	NaN	NaN

On utilise souvent les décorateurs pour afficher le temps d'exécution d'une fonction sans avoir à modifier son code.

- **b-** Construire un décorateur nommé **temps_execution** qui affiche le temps qu'une fonction a mis pour s'exécuter.

Pour rappel, on utilise le module **time** pour mesurer le temps d'exécution d'une fonction :

```

import time

heure_debut = time.time()
ma_fonction()
heure_fin = time.time()

temps_execution = heure_fin - heure_debut
  
```

In [7]:

(/dashboard)

▶

Insérer votre code ici

⌵

🗑️

▶

RUN CELL

⌵

RUN ALL

```

def temps_execution(function):
    import time
    def timer(*args, **kwargs):
        start = time.time()
        function(*args, **kwargs)
        end = time.time()
        print("Timing time : {}".format(end-start))
    return function(*args, **kwargs)
    return timer

```

✓

(834)

(831)

(/hub/dashboard)



UNVALIDATE



SAVE



Robin

(/hub/cheatsheet)



Hide solution

(/hub/progress)



In [28]:

(/typo)



(/forum/271)

```

# solution

import time

def temps_execution(function):
    def timer(*args, **kwargs):
        heure_debut = time.time()
        function(*args, **kwargs)
        heure_fin = time.time()
        temps = heure_fin - heure_debut
        print("Cette fonction s'est exécutée en {} s".format(temps))
    return function(*args, **kwargs)
    return timer

```

Lancer la cellule de code suivante pour tester votre décorateur



In [29]:

(/dashboard)

```
@temps_execution
def importer_csv(*args, **kwargs):
    '''
    Fonction qui permet d'importer un fichier csv dans un DataFrame pandas
    '''
    return pd.read_csv(*args, **kwargs)
```

(/hub/dashboard)

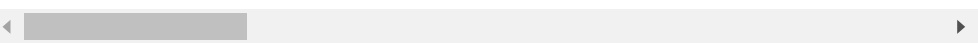
```
importer_csv('country_vaccinations.csv')
```

Cette fonction s'est exécutée en 0.006929159164428711 s

(/hub/cheatsheet) Out[29]:

		country	iso_code	date	total_vaccinations	people_vaccinated	people_full
(/hub/progress)	0	Argentina	ARG	2020-12-29	700.0	NaN	
	1	Argentina	ARG	2020-12-30	NaN	NaN	
(/typo)	2	Argentina	ARG	2020-12-31	32013.0	NaN	
	3	Argentina	ARG	2021-01-01	NaN	NaN	
(/forum/271)	4	Argentina	ARG	2021-01-02	NaN	NaN	
	
	1497	Wales	NaN	2021-01-20	190831.0	190435.0	
	1498	Wales	NaN	2021-01-21	212732.0	212317.0	
	1499	Wales	NaN	2021-01-22	241016.0	240547.0	
	1500	Wales	NaN	2021-01-23	265054.0	264538.0	
	1501	Wales	NaN	2021-01-24	271376.0	270833.0	

1502 rows × 15 columns



2. Paramétrer un décorateur

On peut paramétrer le comportement d'un décorateur de la même manière qu'on paramètre celui d'une fonction.

(/dashboard)

(/hub/dashboard)

(/hub/cheatsheet)

(/hub/progress)

(/typo)

(/forum/271)

Un décorateur paramétré est en fait une fonction qui retourne un décorateur simple.

def decorateur_parametre(description):

def decorateur(function):

def print_before_execution(*args, **kwargs):

 print('Voici ce que renvoie la fonction {}'.format(function.__name__))

 function(*args, **kwargs)

return print_before_execution

return decorateur

@decorateur_parametre('voici ce que renvoie la fonction {}')

def print_hello_world():

 print('hello world')

print_hello_world()

Le résultat renvoyé est le suivant

```
>>> voici ce que renvoie la fonction print_hello_world
hello world
```

- **a-** Construire un décorateur nommé **entree_contient** qui prend en paramètre une chaîne de caractère **a_contenir**.
Si **a_contenir** est incluse dans le premier argument de la fonction décorée, alors cette dernière s'exécute normalement.
Sinon, un message d'erreur doit être renvoyé.

In [10]:

```
def print_hello_world(a, b):
    print('hello world {}'.format(a+b))

print_hello_world.__code__.co_varnames[0]
```

Out[10]:

'a'

In [15]:

(/dashboard)



(/hub/dashboard)



(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

Insérer votre code ici

RUN CELL

RUN ALL

def entree_contient(a_contenir:str):

def decorateur (function):

def test_before_execution(*args, **kwargs):

if a_contenir in str(args[0]):

function(*args, **kwargs)

else:

print("ERROR")

return test_before_execution

return decorateur

(834)

(831)

UNVALIDATE

SAVE

ROBIN BIRON

Hide solution

In [32]:

```
# solution

def entree_contient(a_contenir):
    def decorator(function):
        def new_function(*args, **kwargs):
            if a_contenir in str(args[0]):
                return function(*args, **kwargs)
            else:
                return "la première entrée doit contenir {}".format(a_co
        return new_function
    return decorator
```

Lancer les cellules de code suivantes pour tester votre décorateur

In [33]:

```
@entree_contient('.csv')
def importer_csv(*args, **kwargs):
    """
    Fonction qui permet d'importer un fichier csv dans un DataFrame pand
    """
    return pd.read_csv(*args, **kwargs)
```

In [34]:

(/dashboard)

▶

importer_csv('country_vaccinations.csv')

RUN CELL

≡

RUN ALL

🗑

Out[34]:

👤

📄

📊

📈

✎

🔍

✓

✕

(country)

(iso_code)

(date)

(total_vaccinations)

(people_vaccinated)

(people_full)

(834)

(831)

	country	iso_code	date	total_vaccinations	people_vaccinated	people_full
0	Argentina	ARG	2020-12-29	NaN	NaN	NaN
1	Argentina	ARG	2020-12-30	NaN	NaN	NaN
2	Argentina	ARG	2020-12-31	32013.0	NaN	NaN
3	Argentina	ARG	2021-01-01	NaN	NaN	NaN
4	Argentina	ARG	2021-01-02	NaN	NaN	NaN
...
1497	Wales	NaN	2021-01-20	190831.0	190435.0	
1498	Wales	NaN	2021-01-21	212732.0	212317.0	
1499	Wales	NaN	2021-01-22	241016.0	240547.0	
1500	Wales	NaN	2021-01-23	265054.0	264538.0	
1501	Wales	NaN	2021-01-24	271376.0	270833.0	

1502 rows × 15 columns

Hide solution

In []:

importer_csv('country_vaccinations.csv')

3. Enchaîner les décorateurs

Il est tout à fait possible d'affecter plusieurs décorateurs à une fonction pour accumuler leurs effets

(/dashboard)



RUN CELL



```
def print_before_execution(function):
    def print_then_execute(*args, **kwargs):
        print('voici ce que renvoie la fonction {}'.format(function.__name__))
        function(*args, **kwargs)
    return print_then_execute
```

(834)

(831)

(/hub/dashboard)



UNVALIDATE



SAVE

Robin
BIRON

```
def print_after_execution(function):
    def execute_then_print(*args, **kwargs):
        function(*args, **kwargs)
        print('La fonction a fini de tourner')
    return execute_then_print
```

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

```
@print_after_execution
@print_before_execution
def print_hello_world():
    print('hello world')
```

print_hello_world()

Le résultat renvoyé est le suivant

```
>>> voici ce que renvoie la fonction print_hello_world
hello world
La fonction a fini de tourner
```

En fait, **@print_before_execution** s'applique à la fonction `print_hello_world`, puis **@print_after_execution** s'applique à la fonction renvoyée par `@print_before_execution`.

ATTENTION, l'ordre dans lequel on applique les décorateurs est important!

```
@print_before_execution
@print_after_execution
def print_hello_world():
    print('hello world')
```

print_hello_world()




Renvoie

```
>>> voici ce que renvoie la fonction print_after_execution
# @print_before_execution s'applique à la fonction renvoyée par @print_after_execution
hello world
La fonction a fini de tourner
```

- a- Enchaîner les décorateurs **temps_execution** et **entree_contient** créés dans les parties précédentes. On reprendra le code de la fonction **importer_csv**.

(/dashboard)

In [37]:


 RUN CELL  RUN ALL 




```

# Insérer votre code ici

@entree_contient
@temps_execution
def importer_csv(*args, **kwargs):
    ...
    ()
    UNVALIDATION qui permet d'im
    ...
    return pd.read_csv(*args, **kwargs)

```

 (834) (831)

   Robin BIRON

(/hub/dashboard)



(/hub/cheatsheet)



In [39]:

(/hub/progress)



(/typo)



(/forum/271)

Solution

```

@entree_contient('.csv')
@temps_execution
def importer_csv(*args, **kwargs):
    ...
    Fonction qui permet d'importer un fichier csv dans un DataFrame pand
    ...
    return pd.read_csv(*args, **kwargs)

```



In [41]:

(/dashboard)

▶

importer_csv('country_vaccinations.csv')

RUN CELL

≡

RUN ALL

🗑

Cette fonction s'est exécutée en 0.006882429122924805 s



Out[1]: () (834) (831)

(/hub/dashboard)



	country	iso_code	date	total_vaccinations	people_vaccinated	people_full
UNVALIDATE			SAVE	ROBIN		
0	Argentina	ARG	2020-12-29	BIRON 700.0	NaN	
1	Argentina	ARG	2020-12-30	NaN	NaN	
2	Argentina	ARG	2020-12-31	32013.0	NaN	
3	Argentina	ARG	2021-01-01	NaN	NaN	
4	Argentina	ARG	2021-01-02	NaN	NaN	
...	
1497	Wales	NaN	2021-01-20	190831.0	190435.0	
1498	Wales	NaN	2021-01-21	212732.0	212317.0	
1499	Wales	NaN	2021-01-22	241016.0	240547.0	
1500	Wales	NaN	2021-01-23	265054.0	264538.0	
1501	Wales	NaN	2021-01-24	271376.0	270833.0	

1502 rows × 15 columns



Hide solution

In []:

```
importer_csv('country_vaccinations.csv')
```



4. Envelopper une fonction

Quand nous décorons une fonction comme dans les parties qui précèdent, nous perdons les informations annexes de la fonction (sa documentation par

(/dashboard)



RUN CELL



RUN ALL



ne perdons pas les informations utiles de la fonction (sa documentation par exemple).
Lancer la cellule de code suivante pour le voir par vous-même.



(

)



(

)

(834)

(831)

(/hub/dashboard)

In [42]:

()



UNVALIDATE



SAVE



Robin

BIRON

```
def print_before_execution(function):
    def print_then_execute(*args, **kwargs):
        print('voici ce que renvoie la fonction {}'.format(function.__name__))
        function(*args, **kwargs)
    return print_then_execute

@print_before_execution
def print_hello_world():
    """
    Description de ma fonction
    """
    print("hello world")

help(print_hello_world)
```

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

Help on function print_then_execute in module __main__:

print_then_execute(*args, **kwargs)

Pour remédier à ce problème, on enveloppe notre fonction à décorer à l'aide de **wraps**, qui retourne un décorateur lorsqu'appelé avec une fonction. wraps est disponible dans la librairie **functools**.



In [43]:

(/dashboard)

(/hub/dashboard)

(/hub/cheatsheet)

(/hub/progress)

(/typo)

(/forum/271)

```

from functools import wraps
def print_before_execution(function):
    @wraps(function)
    def print_then_execute(*args, **kwargs):
        print('voici ce que renvoie la fonction {}'.format(function.__name__))
        return function(*args, **kwargs)
    @print_before_execution
    def print_hello_world():
        '''
        Description de ma fonction
        '''
        print("hello world")
help(print_hello_world)

```

Help on function print_hello_world in module __main__:

```
print_hello_world()
Description de ma fonction
```

Aussi contre-intuitifs qu'ils puissent paraître, les décorateurs sont, une fois maîtrisés, des **outils puissants** permettant de **modifier des fonctions** sans modifier leurs définitions. On comprend donc que lorsque l'on cherche à exploiter à plusieurs reprises certaines fonctionnalités précises, définir un décorateur réalisant ces dites fonctionnalités **allègera de façon conséquente notre code**, sans avoir à réaliser d'importantes modifications à celui-ci.

En pratique, certains décorateurs sont plus utilisés que d'autres, un cas d'utilisation notoire est la création d'API en Python, que ce soit avec Flask, FastAPI ou même Streamlit, chacune de ses librairies propose ses propres décorateurs, permettant à un utilisateur de rapidement faire appel à certaines fonctionnalités sans avoir à les redéfinir constamment. Mais, est-ce là la seule fonctionnalité pratique intéressante proposée par les décorateurs ?

5. Exemple intéressant d'utilisation : les décorateurs @cache

En plus de pouvoir alléger notre code, certains décorateurs peuvent **raccourcir nos temps de calculs** s'ils sont utilisés correctement. C'est le cas pour les décorateurs @cache, notamment @lru_cache(), de la librairie native [functools](https://docs.python.org/3/library/functools.html) (<https://docs.python.org/3/library/functools.html>).

Ces décorateurs permettent de garder en cache les différentes valeurs calculées par la fonction qu'il décore. Ceci est extrêmement intéressant pour des fonctions dites **récurrentes**, qui font appel à elles-mêmes lors de leur exécution.

Prenons l'exemple d'une fonction qui calcule la factorielle d'un nombre. On

(/dashboard)

rappelle que la factorielle d'un nombre entier n est définie par :

RUN CELL



RUN ALL



$$\forall n \in \mathbb{N}^*, \\ n! = n \times (n-1) \times \dots \times 1$$



Cette définition nous permet d'envisager les factorielles comme une suite entière :



(834)

(831)

(/hub/dashboard)



UNVALIDATE



SAVE



Robin
BIRON

$$(F_n) : \begin{cases} F_n = n \times F_{n-1} \\ F_0 = 1 \end{cases}$$

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

Cette définition peut se traduire aisément en Python de la manière suivante :

```
def factorial(n):
    if n==1:
        return(n)
    else:
        return(n*factorial(n-1))
```

- a- Implémentez la fonction `factorial` telle que définie plus haut.
- b- Définissez une fonction `boucle` qui calculera et stockera dans une liste les factorielles de 1 à 500. Décorez cette fonction à l'aide du décorateur `@temps_execution` et exécutez là.

In [51]:

```
# Insérez votre code ici
def factorial(n):
    if n==1:
        return(n)
    else:
        return(n*factorial(n-1))

@temps_execution
def boucle():
    l = []
    for i in range(1,500):
        l.append(factorial(i))
    return l
```



In [52]:

(/dashboard)

boucle()
RUN CELL

RUN ALL



Cette fonction s'est exécutée en 0.026744365692138672 s



()



()

(834)

(831)

(/hub/dashboard)



UNVALIDATE



SAVE

Robin
BIRON

()

(/hub/cheatsheet)



(/hub/progress)



(/typo)



Hide solution

(/forum/271)

In []:

```

### Définition de la fonction factorielle
def factorial(n):
    if n==1:
        return(n)
    else:
        return(n*factorial(n-1))

### Définition de la boucle décorée
@temps_execution
def boucle(n=500):
    L=[]
    for k in range(1,n):
        L.append(factorial(k))
    return(L)

### Exécution
boucle(500)


```



Dans notre exemple, la fonction en elle-même ne prend pas plus de 0.07 secondes à s'exécuter, on peut s'imaginer que pour davantage de factorielles à calculer, ce temps peut devenir non-négligeable. Voyons comment on peut améliorer la performance de notre code à l'aide du décorateur `@lru_cache()` si l'on décore la fonction `factorial` avec ce dernier.

- **c-** Décorez la fonction `factorial` avec `@lru_cache()`. Ré-exécutez la fonction `boucle` et observez. N'oubliez pas d'importer le décorateur à partir du module **functools**.

(/dashboard)

 RUN CELL
In [55]:

 RUN ALL



(/hub/dashboard)



(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

```
# Insérez votre code ici (834) (831)
from functools import lru_cache
# Insérez votre code ici
@lru_cache()
def factorial(n):
    if n==1:
        return(n)
    else:
        return(n*factorial(n-1))

@temps_execution
def boucle():
    l = []
    for i in range(1,500):
        l.append(factorial(i))
    return l
```

In [56]:

boucle()

Cette fonction s'est exécutée en 0.0003044605255126953 s



Hide solution






In [57]:




(/dashboard)

```

from functools import lru_cache
@lru_cache()
def factorial(n):
    if n==1:
        return(n)
    else:
        return(n*factorial(n-1))

```

 RUN CELL  RUN ALL  SAVE

 UNVALIDATE  SAVE  Robin BIRON

(/hub/dashboard)

(/hub/cheatsheet)

```

TypeError                                Traceback (most recent call last)
<ipython-input-57-5afe27b5ffc9> in <module>
      7         return(n*factorial(n-1))
      8
----> 9 boucle(500)

<ipython-input-28-b8e5e61e7d4b> in timer(*args, **kwargs)
      6     def timer(*args, **kwargs):
      7         heure_debut = time.time()
----> 8         function(*args, **kwargs)
      9         heure_fin = time.time()
     10         temps = heure_fin - heure_debut

```

```
TypeError: boucle() takes 0 positional arguments but 1 was given
```

À stocker les valeurs en cache, on gagne du temps sur l'exécution de notre fonction. Voyons un nouvel exemple plus flagrant.

- **d-** Définissez une nouvelle fonction `print_hello` qui retournera **au bout de 5 secondes** une chaîne de caractère valant par défaut 'hello'. Décorez la avec `@lru_cache()` et `@temps_execution` puis exécutez-la sans préciser d'argument.
- **e-** Stockez la valeur retournée dans une variable à quelconque.

In []:

```
# Insérez votre code ici
```

```
def print_hello():
```

Hide solution

In [58]:

(/dashboard)

```
@tempo.execution
@RUN_CELL
@lrucache()
def print_hello(a='hello'):
    time.sleep(5)
    return a
a=print_hello()
```

(/hub/dashboard)

UNVALIDATE Cette fonction s'est exécutée en 0.005033254623413 s

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

In [59]:

```
# Insérez votre code ici
```

```
b=print_hello()
```

Cette fonction s'est exécutée en 1.430511474609375e-06 s

Hide solution

In []:

```
b= print_hello()
```

Cette fois-ci, l'exécution a été **quasi-instantanée** et la mise en cache a permis de **court-circuiter le temps d'attente** de 5 secondes prévu par la fonction lors de sa seconde itération. On peut légitimement penser que la mise en cache n'est pas responsable de ce gain de temps, vous pouvez exécuter la cellule suivante pour vous rendre compte du contraire.

In [60]:

(/dashboard)

```

@temps_execution
def print_hello(a='hello'):
    time.sleep(5)
    return a
a= print_hello()
b= print_hello()

```

(/hub/dashboard)

UNVALIDATE SAVE  Robin BIRON

Cette fonction s'est exécutée en 5.005028247833252 s

Cette fonction s'est exécutée en 5.0050249099731445 s

(/hub/cheatsheet)



Conclusion

(/hub/progress) >>



(/typo)



(/forum/271)

- Les décorateurs sont des outils qui s'apparentent à des fonctions et qui permettent de **modifier le comportement** d'autres fonctions.
- Ces décorateurs se paramètrent comme n'importe quelle autre fonction, permettant une grande **flexibilité** d'exploitation.
- Outre un aspect pratique évident, certains décorateurs tels que les décorateurs `@cache` permettent même d'optimiser le code afin d'économiser un temps de calcul non-négligeable.

✖ Unvalidate

