Les décorateurs

Dans ce chapitre, nous allons apprendre à utiliser et à créer des décorateurs.

Un **décorateur** est simplement une fonction qui modifie le comportement d'autres fonctions. C'est très utile lorsque l'on veut ajouter du même code à plusieurs fonctions existantes.

Créer un décorateur `python`

Un décorateur permet de modifier le comportement d'une fonction. Il commence par un @ suivi de lettres ou de chiffres. Il se place sur la ligne précédant la définition d'une fonction. Comme ceci :

```
@decorator
def fonction():
    """documentation de la fonction"""
    print("Great!")
```

`Python` intègre de nombreux décorateurs standards mais vous pouvez également en définir vous-même. Pourquoi ? Car un décorateur est simplement une fonction qui prend en paramètre une fonction et renvoie une (autre) fonction.

Voici un premier exemple de décoration de fonction.

In [68]:
```
def mon_decorateur(in_function):
    def out_function():
        """titi"""
        print("On peut faire des choses avant")
        print("On exécute maintenant la fonction")
        in_function()
        print("Et on peut faire des trucs après")

    out_function.__name__ = in_function.__name__
    out_function.__doc__ = in_function.__doc__
    return out_function
```

In [69]:
```
def ma_fonction():
    print("Je ne fais rien du tout !")

ma_fonction()
print(ma_fonction.__name__)

ma_nouvelle_fonction = mon_decorateur(ma_fonction)

ma_nouvelle_fonction()
print(ma_nouvelle_fonction.__name__)
```

```
ma_fonction
ma_fonction
```

In [75]:
```
@mon_decorateur
def ma_fonction(x):
    """toto"""
    print("Je ne fais rien du tout !")
    print(x)

ma_fonction(0)
#print(ma_fonction.__name__)
#help(ma_fonction)
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-75-dc85520c8d03> in <module>
      5     print(x)
      6
----> 7 ma_fonction(0)
      8 #print(ma_fonction.__name__)
      9 #help(ma_fonction)

TypeError: out_function() takes 0 positional arguments but 1 was g
iven
```

In [72]:
```
x = 2

def f(y):
    return x+y

f(0)
```

Out[72]:
```
2
```

Lorsque la fonction a des arguments et retourne quelque chose, il est important de ne pas changer son comportement.

Pour les arguments passés à la fonction, nous utilisons les mots clés `*args` et `**kwargs` que l'on a déjà vus dans le cours sur les fonctions. Cela permet de passer des nombres arbitraires d'arguments et d'arguments optionnels.

Pour le return, il suffit que la nouvelle fonction retourne le même résultat.

Voici un nouvel exemple.

In [76]:
```
def mon_decorateur_avec_arguments(in_function):
    def out_function(*args, **kwargs):
        print(f"La fonction s'appelle {in_function.__name__}")
        print("ses arguments sont ", args)
        print("ses arguments optionnels sont ", kwargs)
        out = in_function(*args, **kwargs)
        print(f"la fonction a été exécutée et a retourné {out}")
        return out

    out_function.__name__ = in_function.__name__
    out_function.__doc__ = in_function.__doc__

    return out_function
```

In [78]:
```
@mon_decorateur_avec_arguments
def f(x):
    return x*x

f(1)
```

```
La fonction s'appelle f
ses arguments sont  (1,)
ses arguments optionnels sont  {}
la fonction a été exécutée et a retourné 1
```

Out[78]: 1

In [85]:
```python
def decorateur_multiplie(f_in):
    def f_out(*args, **kwargs):
        args_new = tuple([2*argk for argk in args])
        out = f_in(*args_new, **kwargs)
        if not isinstance(out, tuple):
            out = (out, )
        out_new = tuple([outk/2 for outk in out])
        if len(out_new) == 1:
            out_new = out_new[0]
        return out_new
    return f_out

@decorateur_multiplie
def f(x):
    return x*x

def g(x):
    return (2*x)**2/2

x = 7
print(f"f(x) = {f(x)}, g(x) = {g(x)}")
```

```
f(x) = 98.0, g(x) = 98.0
```

In [86]:
```python
@mon_decorateur_avec_arguments
def f(x, p=2):
    return x**p

f(2)
f(2, 3)
f(2, p=3)
```

```
La fonction s'appelle f
ses arguments sont  (2,)
ses arguments optionnels sont  {}
la fonction a été exécutée et a retourné 4
La fonction s'appelle f
ses arguments sont  (2, 3)
ses arguments optionnels sont  {}
la fonction a été exécutée et a retourné 8
La fonction s'appelle f
ses arguments sont  (2,)
ses arguments optionnels sont  {'p': 3}
la fonction a été exécutée et a retourné 8
```

Out[86]:
```
8
```

Prenons un autre exemple concret que certains d'entre vous connaissent déjà (attention à l'abus de caféine).

In [87]:
```python
class CoffeeMachine():
    """
    une classe pour la machine à café
    """
    water_level = 100  # variable partagée par toutes les instances

    def __init__(self):
        """initialise l'instance"""
        self.__nb_coffee = 0

    def _start_machine(self):
        """démarre la machine"""
        if self.water_level > 20:
            return True
        else:
            print("Ajoutez de l'eau s'il vous plait !")
            return False

    def __boil_water(self):
        """chauffe l'eau"""
        print("préchauffage...")

    def __grind_beans(self):
        """préparation du grain"""
        print("préparation du grain...")

    def __brew_coffee(self):
        """extraction du café"""
        print("extraction")

    def make_coffee(self):
        """prépare un nouveau café"""
        if self._start_machine():
            self.water_level -= 20
            self.__boil_water()
            self.__grind_beans()
            self.__brew_coffee()
            self.__nb_coffee += 1
            print("Prenez votre café !")
```

In [88]:
```
machine = CoffeeMachine()
for i in range(0, 5):
    machine.make_coffee()
```

```
préchauffage...
préparation du grain...
extraction
Prenez votre café !
préchauffage...
préparation du grain...
extraction
Prenez votre café !
préchauffage...
préparation du grain...
extraction
Prenez votre café !
préchauffage...
préparation du grain...
extraction
Prenez votre café !
Ajoutez de l'eau s'il vous plait !
```

Afin de débugger notre code, nous pourrions avoir envie que chaque fonction affiche son nom et les arguments qui lui sont passés.

In [117]:
```
def name(func):
    def inner(*args, **kwargs):
        print(f" running {func.__name__} {args} ".center(80, '-'))
        return func(*args, **kwargs)
    return inner

class CoffeeMachine():
    """
    une classe pour la machine à café
    """
    water_level = 100  # variable partagée par toutes les instances

    def __init__(self):
        """initialise l'instance"""
        self.__nb_coffee = 0
        self.nom = "ma machine"

    def __repr__(self):
        return self.nom

    @name
    def _start_machine(self):
        """démarre la machine"""
        if self.water_level > 20:
            return True
        else:
```

```
            print("Ajoutez de l'eau s'il vous plait !")
            return False

    @name
    def __boil_water(self):
        """chauffe l'eau"""
        print("préchauffage...")

    @name
    def __grind_beans(self):
        """préparation du grain"""
        print("préparation du grain...")

    @name
    def __brew_coffee(self):
        """extraction du café"""
        print("extraction")

    @name
    def make_coffee(self):
        """prépare un nouveau café"""
        if self._start_machine():
            self.water_level -= 20
            self.__boil_water()
            self.__grind_beans()
            self.__brew_coffee()
            self.__nb_coffee += 1
            print("Prenez votre café !")
```

In [118]:
```
machine = CoffeeMachine()
for i in range(0, 5):
    machine.make_coffee()
```

```
---------------------- running make_coffee (ma machine,) ---------
--------------
---------------------- running _start_machine (ma machine,) -------
--------------
---------------------- running __boil_water (ma machine,) --------
--------------
préchauffage...
---------------------- running __grind_beans (ma machine,) --------
--------------
préparation du grain...
---------------------- running __brew_coffee (ma machine,) --------
--------------
extraction
Prenez votre café !
---------------------- running make_coffee (ma machine,) ---------
--------------
---------------------- running _start_machine (ma machine,) -------
--------------
---------------------- running __boil_water (ma machine,) --------
--------------
```

```
préchauffage...
---------------------- running __grind_beans (ma machine,) --------
--------------
préparation du grain...
---------------------- running __brew_coffee (ma machine,) --------
--------------
extraction
Prenez votre café !
---------------------- running make_coffee (ma machine,) ---------
--------------
---------------------- running _start_machine (ma machine,) -------
--------------
---------------------- running __boil_water (ma machine,) --------
--------------
préchauffage...
---------------------- running __grind_beans (ma machine,) --------
--------------
préparation du grain...
---------------------- running __brew_coffee (ma machine,) --------
--------------
extraction
Prenez votre café !
---------------------- running make_coffee (ma machine,) ---------
--------------
---------------------- running _start_machine (ma machine,) -------
--------------
---------------------- running __boil_water (ma machine,) --------
--------------
préchauffage...
---------------------- running __grind_beans (ma machine,) --------
--------------
préparation du grain...
---------------------- running __brew_coffee (ma machine,) --------
--------------
extraction
Prenez votre café !
---------------------- running make_coffee (ma machine,) ---------
--------------
---------------------- running _start_machine (ma machine,) -------
--------------
Ajoutez de l'eau s'il vous plait !
```

Utiliser un ou plusieurs décorateurs `python`

De nombreux décorateurs ont déjà été codés et peuvent être utilisés. Vous pourrez les trouver à cette adresse [liste des décorateurs (https://wiki.python.org/moin/PythonDecoratorLibrary)](https://wiki.python.org/moin/PythonDecoratorLibrary)

Un décorateur que nous utilisons souvent est le décorateur `@property` qui fonctionne seulement si la classe hérite de la classe `object` (à partir de python 3).

In [131]:
```
def _CtoK(C):
    return C + 273.15

def _FtoK(F):
    return (F+459.67)*5/9

def _KtoC(K):
    return K - 273.15

def _KtoF(K):
    return 9/5*K-459.67


class temperature(object):
    """une classe pour la température"""
    def __init__(self, K=0, C=None, F=None):
        self._K = K
        if K is not None:
            self._K = K
        if C is not None:
            self._K = _CtoK(C)
        if F is not None:
            self._K = _FtoK(F)

    @property
    def Kelvin(self):
        return self._K

    @Kelvin.setter
    def Kelvin(self, K):
        self._K = K

    @property
    def Celsius(self):
        return _KtoC(self._K)

    @Celsius.setter
    def Celsius(self, C):
        self._K = _CtoK(C)

    @property
    def Fahrenheit(self):
        return _KtoF(self._K)

    ##@Fahrenheit.setter
    #def Fahrenheit(self, F):
    #    self._K = _FtoK(F)

    def __str__(self):
        return f"La température vaut K={self._K:.2f} (C={_KtoC(self
._K):.2f}, F={_KtoF(self._K):.2f})"
```

In [132]:
```
T = temperature(C=0)
print(T)
print(T.Kelvin)
T.Fahrenheit = 50
print(T)
print(T.Celsius)
```

```
La température vaut K=273.15 (C=0.00, F=32.00)
273.15

---------------------------------------------------------------------
---------
AttributeError                            Traceback (most recent c
all last)
<ipython-input-132-49901c91bc1e> in <module>
      2 print(T)
      3 print(T.Kelvin)
----> 4 T.Fahrenheit = 50
      5 print(T)
      6 print(T.Celsius)

AttributeError: can't set attribute
```

In [ ]:
```
```