Python decoradores

/via: https://medium.com/@LuisMBaezCo/decoradores-con-clases-y-funciones-en-python-2fafb22dba43

- https://codigofacilito.com/articulos/decoradores-python

Decorador mediante una función

```
def decorator(func):
  print("Decorator")
  return func

@decorator
def Hello():
  print("Hello World")

Hello()
# [Output]:
# Decorator
# Hello World
```

Decorador mediante dos funciones

```
#print_args: Función decoradora
def printArgs(func):
  def innerFunc(*args, **kwargs):
    print(args)
    print(kwargs)
    return func(*args, **kwargs)

  return innerFunc

#foobar: Función decorada
@printArgs
def foobar(x, y, z):
  return x * y + z

print(foobar(3, 5, z=10))
# (3, 5)
# {'z': 10}
# 25 = 3 * 5 + 10
```

Decorador simple mediante Clases

(se puede crear una funcion estática `@staticmethod` en una clase e invocarla de esa manera)

```
class Decorator(object):
  """Clase de decorador simple."""
  def __init__(self, func):
```

```
    self.func = func

  def __call__(self, *args, **kwargs):
    print('Antes de ser llamada la función.')
    retorno = self.func(*args, **kwargs)
    print('Despues de ser llamada la función.')
    print(retorno)
    return retorno

@Decorator
def function():
  print('Dentro de la función.')
  return "Retorno"

function()
# Antes de ser llamada la función.
# Dentro de la función.
# Despues de ser llamada la función.
# 'Retorno'
```

```
import types
print(isinstance(function, types.FunctionType))
# False
print(type(function))
# <class '__main__.Decorator'>
```

Decorando métodos de una clase

```
from types import MethodType

class Decorator(object):

  def __init__(self, func):
    self.func = func

  def __call__(self, *args, **kwargs):
    print('Dentro del Decorador.')
    return self.func(*args, **kwargs)

  def __get__(self, instance, cls):
    # Retorna un método si se llama en una instancia
    return self if instance is None else MethodType(self, instance)

class Test(object):
  @Decorator
  def __init__(self):
    print("Dentro de la función decorada")

a = Test()
# Dentro del Decorador.
# Dentro de la función decorada
```

Decorador con parámetros mediante clases

```
class MyDec(object):
  def __init__(self, flag):
    self.flag = flag

  def __call__(self, original_func):
    decorator_self = self
    def wrappee(*args, **kwargs):
      print('en decorador antes de wrapee ', decorator_self.flag)
      original_func(*args,**kwargs)
      print('en decorador despues de wrapee', decorator_self.flag)
    return wrappee

@MyDec(flag='foo de fa fa')
def bar(a,b,c):
  print('En bar(...) : ',a,b,c)

if __name__ == "__main__":
  bar(1, "Hola", True)

#Out:
# en decorador antes de wrapee  foo de fa fa
# en bar 1 Hola True
# en decorador despues de wrapee foo de fa fa
```

Ejemplo decoración con clases

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from types import MethodType

class Decorator():
    def __init__(self,func):
        print(f"Decorator.__init__")
        self.func = func

    def __call__(self, *args, **kwargs):
        print("Decorator pre")
        print(nom_funcio := self.func.__name__)
        print(B.classvar)

        retorn = self.func(*args,nom_funcio=nom_funcio,**kwargs)
        print("Decorator post")

        return retorn

    def __get__(self,instance,cls):
        # Retorna un método si se llama en una instancia
        return self if instance is None else MethodType(self, instance)
```

```
class Decorator2():
    def __init__(self,func):
        print(f"Decorator2.__init__")
        self.func = func

    def __call__(self):
        print("Decorator2 pre")
        print(self.func.__name__)
        print(B.classvar)

        self.func(self)
        print("Decorator2 post")


class A():
    @staticmethod
    def decorator(func):
        def wrapper(*args,**kwargs):
            print("A.decorator pre")

            print(nom_funcio := func.__name__)
            print(B.classvar)

            retorn = func(*args,nom_funcio=nom_funcio,**kwargs)
            print("A.decorator post")

            return retorn


        return wrapper


class B(A):
    classvar = None

    def __init__(self,var=None):
        self.var = var
        B.classvar = var

    def decorator(func):
        def wrapper(*args,**kwargs):
            print("decorator pre")

            print(nom_funcio := func.__name__)
            print(B.classvar)

            retorn = func(*args,nom_funcio=nom_funcio,**kwargs)

            print("decorator post")

            return retorn


        return wrapper

    @Decorator
```

```
    def run(self,nom_funcio=None):
        print(f"B.run({nom_funcio})")

    @Decorator2
    def run2(self,nom_funcio=None):
        print(f"B.run2({nom_funcio})")

    @A.decorator
    def run3(self,nom_funcio=None):
        print(f"B.run3({nom_funcio})")

    @decorator
    def run4(self,nom_funcio=None):
        print(f"B.run4({nom_funcio})")


o = B('mate')
o.run()
o.run2()
o.run3()
o.run4()
print(o.__dict__)
```

- `classvar` se usa como variable de intercambio de información, pero es una variable de clase. Varias instancias para diferentes cometidos darian conflictos.