W Pythonie funkcję możemy przekazać jako parametr do innej funkcji, możemy także zwrócić z funkcji inną funkcję:

```
def fun(pf):
    return pf

def fsum(p):
    return sum(range(p))

f=fun(fsum)
f(10)
```

Idąc dalej, definicja funkcji może się pojawić wewnątrz definicji innej funkcji i funkcja zagnieżdżona może zostać zwrócona z funkcji zewnętrznej:

```
def fsum(p):
    return sum(range(p))

def dec(pf):
    def fw(p):
        return pf(p)/2
    return fw

fsum=dec(fsum)
fsum(10)
```

czy nieco mniej czytelnie:

```
dec(fsum)(10)
```

# Dekorator

Zdefiniowaliśmy zatem funkcję przyjmującą jako parametr inną funkcję i zwracającą funkcję będącą jej zmodyfikowaną wersją. Funkcja taka nazywana jest dekoratorem. Python zapewnia mechanizm ułatwiający korzystanie z dekoratorów. Sygnaturę funkcji możemy poprzedzić dekoratorem (mówimy, że funkcja została obłożona dekoratorem):

```
def dec(pf):
    def fw(p):
        return pf(p)/2
    return fw

@dec
def fsum(p):
    return sum(p)
```

Funkcję taką wywołujemy „normalnie”:

```
fsum(range(10))
```

ale jej wywołanie następuje poprzez dekorator, przez co nie modyfikując samej funkcji możemy dodać określone funkcjonalności.

W przypadku, gdy do dekoratora będziemy chcieli przekazać parametry, konieczne jest zdefiniowanie kolejnej funkcji otaczającej:

```
def fsum(p):
    return sum(p)

def modulo(par):
    def fz(pf):
        def fw(p):
            return pf(filter(lambda x: x%par, p))
        return fw
    return fz
```

gdzie:

- par parametry przekazywane do dekoratora
- pf funkcja okładana dekoratorem
- p parametry dekorowanej funkcji

Wywołanie bez okładania funkcji dekoratorem:

```
modulo(2)(fsum)(range(10))
```

i zdecydowanie bardziej czytelne przy użyciu dekoratora:

```
@modulo(2)
def fsum(p):
    return sum(p)

fsum(range(10))
```

Funkcję możemy obłożyć więcej niż jednym dekoratorem, ważna jest wtedy ich kolejność (proszę się zastanowić/sprawdzić co się zmieni w poniższym przykładzie, pomimo otrzymania takiego samego wyniku, jeśli zmienimy kolejność dekoratorów):

```
def modulo(par):
    def fz(pf):
        def fw(p):
            return pf(filter(lambda x: x%par, p))
        return fw
    return fz

def power(par):
    def fz(pf):
        def fw(p):
            return pf(map(lambda x: x**par, p))
        return fw
    return fz

@modulo(2)
@power(2)
def fsum(p):
    return sum(p)

fsum(range(10))
```

Nieco inne możliwości daje zdefiniowanie dekoratora jako funktora. W Pythonie żeby klasa stała się funktorem należy w niej zdefiniować metodę __call__. Metoda inicjalizacyjna klasy wywoływana jest w momencie okładania funkcji dekoratorem.

Dekorator bez parametru:

```
class Modulo2:
    def __init__(self, pf):
        self._pf=pf

    def __call__(self, p):
        return self._pf(filter(lambda x: x%2, p))

@Modulo2
def fsum(p):
 return sum(p)

fsum(range(10))
```

Dekorator z parametrem:

```
class Modulo:
    def __init__(self, par):
        self._p=par

    def __call__(self, pf):
        def fw(p):
            return pf(filter(lambda x: x%self._p, p))
        return fw

@Modulo(2)
def fsum(p):
    return sum(p)

fsum(range(10))
```

Język dostarcza także dekoratory wbudowane. Korzystaliście już z nich Państwo tworząc metody statyczne czy abstrakcyjne. Dekoratory mogą być też użyte do implementacji metod służących do nadawania i zwracania wartości składowych obiektów, np. tych pseudo prywatnych:

```
class A:
    def __init__(self):
        self.a=0

    @property
    def a(self):
        return self._a

    @a.setter
    def a(self,w):
        self._a=w

    @a.getter
    def a(self):
        return self._a

ob=A()
print(ob.a)
ob.a=9
print(ob.a)
```