Akademia Pythona

VIII Zagadnienia zaawansowane

KN Pythona - Kurs Pythona

KN Pythona wita na kursie Pythona.

Plan:

- Dekoratory

Dekoratory

Dotychczas:

- @property
- @abstractmethod
- @classmethod
- @staticmethod

Dekoratory

Dekoratory to obiekty wywoływalne.

Dekoratory

Dekoracja jest sposobem określania kodu zarządzającego dla funkcji oraz klas.

Dekoratory

Dekoracja przetwarza obiekty wywoływalne (funkcje oraz klasy).
Dekorator dodaje instrukcje wykonywalne po kodzie obiektów.

Zarządzanie

Dekoratory mogą zarządzać wywołaniami oraz instancjami.

Dekoratory

Dekoracja zachodzi poprzez ponowne dowiązanie nazw przetwarzanych obiektów.

Dekoratory

```
@decorator
def F(arg):
    ...
F(99)

def F(arg):
    ...
F = decorator(F)
F(99)
```

Cechy dekoratorów

Cechy dekoratorów:

- Niska powtarzalność
- Strukturyzacja i hermetyzacja
- Jawność i czytelność

Dekoratory funkcji

```
func(6, 7) # udekorowana funkcja
decorator(func)(6, 7) # dekorator zwraca obiekt
```

Dekoratory funkcji

```
def decorator(F):
    return F
@decorator
def func(a, b):
    ...
```

Dekoratory funkcji

```
def decorator(F):
    # zapisanie lub użycie F
    # zwrócenie innego obiektu
```

Dekoratory funkcji

```
def decorator(F):
    def wrapper(*args):
        # użycie F oraz args
        # F(*args)
    return wrapper
```

Dekoratory funkcji

```
class decorator:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args):
        # użycie self.func oraz args
        # self.func(*args)
```

Dekoratory metod

Powyższy kod nie udekoruje poprawnie metody klasy.

Dekoratory metod

```
class C:
    @decorator
    def method(self, x, y):
        ...
```

Dekoratory klas

```
@decorator
class C:
    ...
x = C(99)
```

Dekoratory klas

```
class C:
    ...
C = decorator(C)
x = C(99)
```

Dekoratory klas

```
def decorator(C):
    # przetworzenie klasy C
    return C
```

Dekoratory klas

```
def decoratory(C):
    # zapisanie lub przetworzenie C
    # zwrócenie klasy
```

Dekoratory klas

```
def decorator(cls):
    class Wrapper:
        def __init__(self, *args):
            self.wrapped = cls(*args)
        def __getattr__(self, name):
            return getattr(self.wrapped, name)
    reutrn Wrapper
```

Dekoratory klas

```
class Decorator: # tylko jedna instancja!
    def __init__(self, C):
        self.C = C
    def __call__(self, *args):
        self.wrapped = self.C(*args)
        return self
    def __getattr__(sefl, name):
        return getattr(self.wrapped, name)
```

Dekoratory klas

```
class Wrapper: ...
def decorator(cls):
    def on_call(*args):
        return Wrapper(C(*args))
    return on_call
```

Zagnieżdżanie dekoratorów

```
@A
@B
@C
def f(...):
    ...
def f(..):
    ...
f = A(B(C(f)))
```

Zagnieżdżanie dekatorów

```
def d1(F): return lambda: 'a' + F()
def d2(F): return lambda: 'b' + F()
def d3(F): return lambda: 'c' + F()
@d1
@d2
@d3
def func():
    return 'd'
print(func()) # abcd
```

Argumenty dekoratorów

```
@decorator(A, B)
def F(arg):
    ...
F(99)
```

Argumenty dekoratorów

```
def F(arg):
    ...
F = decorator(A, B)(F)
F(99)
```

Argumenty dekoratorów

```
def decorator(A, B):
    # zapisanie lub użycie A i B
    def actual_decorator(F):
        # zapisanie lub użycie F
        # zwrócenie obiektu
        return callable
    return actual_decorator
```

Śledzenie wywołań

```
class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args):
        sefl.calls += 1
        print(f"Wywołanie {self.func.__name__}" +\
              " nr {self.calls}")
        sefl.func(*args)
@tracer
def spam(a, b, c):
    print(a, b, c)
```

Śledzenie wywołań 2

```
class tracer:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        return self.func(*args, **kwargs)
```

Zakresy zawierające, tzw. closures

```
calls = 0
def tracer(func):
    def wrapper(*args, **kwargs):
        global calls
        calls += 1
        print(calls)
        return func(*args, **kwargs)
    return wrapper # do obiektu dołączony jest zakres
```

Dekoracja przy użyciu deskryptorów

```
class tracer:
    def __init__(sefl, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print(self.calls)
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):
        return wrapper(self, instance)
class wrapper:
    def __init__(self, desc, subj):
        self.desc = desc
        self.subj = subj
    def __call__(self, *args, **kwargs):
        return self.desc(self.subj, *args, **kwargs)
```

Inne zastosowania

Dekoratory:

- Routing we frameworkach webowych
- Mierzenie czasu wykonania
- Systemy ORM
- Systemy bazodanowe
- Wzorce projektowe (np. singleton)
- Atrybuty prywatne
- Sprawdzanie zakresów argumentów
- Testy