Laboratorium **Programowanie w języku Python 2** Wydział Elektrotechniki Automatyki I Informatyki Politechnika Świętokrzyska	
Studia: **Stacionarne I stopnia**	Kierunek: **Informatyka**
Data wykonania: **08.04.2021**	Grupa: **3ID16B**
Imię I nazwisko: **Arkadiusz Więcław**	Temat ćwiczenia: **Dekoratory** **Właściwości**

Zad 1:

```
from functools import wraps
import functools
import time

def przyklad1():
    """
    Program tworzy dwie funcje my_decorator i say_whee. Funkcja
my_decorator przyjmuje jako argument
    funkcje say_whee. Wewnatrz funkcji my_decorator znajduje sie
definicja funkcji ktora wykonuje funkcje say_whee
    Funkcja my_decorator zwraca fukcje wrapper.
    """
    def my_decorator(func):
        def wrapper():
            print("Something is happening before the function is
called.")
            func()
            print("Something is happening after the function is
called.")
        return wrapper
    def say_whee():
        print("Whee!")
    say_whee = my_decorator(say_whee)
    say_whee()

def przyklad2():
    """
    Dziala podobnie co program wyżej jest uproszczony z powodu dodania
adnotacji @my_decorator,
    która umożliwia wywolanie dekorator  my_decorator  przy wywolaniu
fuknkcji say_whee
    """
    def my_decorator(func):
        def wrapper():
            print("Something is happening before the function is
called.")
            func()
            print("Something is happening after the function is
called.")
```

```
        return wrapper
    @my_decorator
    def say_whee():
        print("Whee!")
    say_whee()

def przyklad3():
    """
    Program tworzy dekorator którzy przyjmuje dowolno liczbe
argumentow  zdefiniowanych w funckji my_print.
    Funkcja my_print jest wywolywane dwa razy wewnatrz funkcji
wrapper_do_twice.Funkcja my_print
    wyswietla wszystkie arugemnty.
    """
    def do_twice(func):
        def wrapper_do_twice(*args, **kwargs):
            func(*args, **kwargs)
            func(*args, **kwargs)
        return wrapper_do_twice
    @do_twice
    def my_print(text):
        print(text)
    my_print('test')

def przyklad4():
    """
    Program tworzy dekorator my_decorator który zawiera wbudowany
dekorator @wraps który przyjmuje
    funkcje i dekoruje funkcje wraper. Funkcja wraper zwraca
wprowadzona funkcje.
    """
    def my_decorator(f):
        @wraps(f)
        def wrapper(*args, **kwds):
            print('Calling decorated function')
            return f(*args, **kwds)
        return wrapper
    @my_decorator
    def example():
        """Docstring"""
        print('Called example function')
    example()
```

```
    print(example.__name__)
    print(example.__doc__)

def przyklad5():
    """
    Program jest podobny do poprzedniego tylko zamiast zwracac funkcje
    bezposrednio wykorzystuje do tego celu zmienna value
    """
    def my_decorator(func):
        @wraps(func)
        def wrapper(*args, **kwds):
            print('Wywołanie funkcji dekorowanej')
            value = func(*args, **kwds)
            print('Koniec wywołania funkcji dekorowanej')
            return value
        return wrapper
    @my_decorator
    def example():
        """Dokumentacja funkcji"""
        print('Funkcja dekorowana')
    example()
    print(example.__name__)
    print(example.__doc__)

def przyklad6():
    """
    Program tworzy dekorator timer który służy do zliczania czasu
wykonywania
    funkcji dekorowanej waste_some_time która podnosi do potegi liczby
od 0 do 99999.
    """
    def timer(func):
        """Print the runtime of the decorated function"""
        @functools.wraps(func)
        def wrapper_timer(*args, **kwargs):
            start_time = time.perf_counter()    # 1
            value = func(*args, **kwargs)
            end_time = time.perf_counter()  # 2
            run_time = end_time - start_time    # 3
#             print(f"Finished {func.__name__!r} in {run_time:.4f}
secs")
            return value
```

```
        return wrapper_timer
    @timer
    def waste_some_time(num_times):
        for _ in range(num_times):
            sum([i**2 for i in range(10000)])
    waste_some_time(5)

def przyklad7():
    """
    Program tworzy klase Person która ma metody fget ,fset i
fdel.Metoda fget wyswietla i zwraca przypisana
    wartość do wlasciwości name. Metoda fset odpowiada ze ustawiania
wartosc atrybutowi name. Metoda fdel
    odpowiada za usuwanie  zmiennej name , funkcja property  tworzony
deskryptor.
    """
    class Person(object):
        def init(self):
            self._name = ''
        def fget(self):
            print ("Getting: %s" % self._name  )
            return self._name
        def fset(self, value):
            print ("Setting: %s" % value)
            self._name = value.title()
        def fdel(self):
            print ("Deleting: %s" %self._name)
            del self._name
        name = property(fget, fset, fdel, "I'm the property.")
    user = Person()
    user.name = 'john smith'
    print(user.name)
    del user.name

def przyklad8():
    """
    Program tworzy metody które bedą odpowiadać za usuwanie zmiennej ,
wyswietlanie zawartości
    zmiennej i przypisywania wartości zmiennej .Funkcja property
tworzy deskryptor klasy .
    Konstruktor klasy Alphabet przyjmuje wartość która zostanie
przypisana do zmiennnej value.
```

```
    """
    # Alphabet class
    class Alphabet:
        def __init__(self, value):
            self._value = value
        # getting the values
        def getValue(self):
            print('Getting value')
            return self._value
        # setting the values
        def setValue(self, value):
            print('Setting value to ' + value)
            self._value = value
        # deleting the values
        def delValue(self):
            print('Deleting value')
            del self._value
        value = property(getValue, setValue, delValue, )
    # passing the value
    x = Alphabet('GeeksforGeeks')
    print(x.value)
    x.value = 'GfG'
    del x.value

def przyklad9():
    """
    Program tworzy deskryptor wykorzystujac @property dodatkowo w
klasie Person znajduja sie
    dekorator .setter i deleter. Setter wskazuje  na metode która
ustawiania wlasciwości name a deleter
    wskazuje na metoda która  usuwa wlasciwości name.
    """
    class Person(object):
        def init(self):
            self._name = ''
        @property
        def name(self):
            print("Getting: %s" % self._name)
            return self._name
        @name.setter
        def name(self, value):
            print ("Setting: %s" % value )
```

```
            self._name = value.title()
        @name.deleter
        def name(self):
            print (">Deleting: %s" % self._name)
            del self._name
    user = Person()
    user.name = 'john smith'
    print(user.name)
    del user.name


def przyklad10():

    """
    Program tworzy deskryptor wykorzystujac  dekarator property
dodatkowo w klasie Alphabet znajduja się
    dekorator .setter i deleter. Setter wskazuje  na metode ktora
ustawiania wlasciwosc value a deleter
    wskazuje na metoda ktore jest odpowiedzialna za usuwania
wlasciwosc value. K
    """
    class Alphabet:
        def __init__(self, value):
            self._value = value
        # getting the values
        @property
        def value(self):
            print('Getting value')
            return self._value
        # setting the values
        @value.setter
        def value(self, value):
            print('Setting value to ' + value)
            self._value = value
        # deleting the values
        @value.deleter
        def value(self):
            print('Deleting value')
            del self._value

    # passing the value
    x = Alphabet('Peter')
```

```
    print(x.value)
    x.value = 'Diesel'
    del x.value

def main():
    print("zad 1 =")
    print("\nPrzyklad 1 =")
    przyklad1()
    print("\nPrzyklad 2 =")
    przyklad2()
    print("\nPrzyklad 3 =")
    przyklad3()
    print("\nPrzyklad 4 =")
    przyklad4()
    print("\nPrzyklad 5 =")
    przyklad5()
    print("\nPrzyklad 6 =")
    przyklad6()
    print("\nPrzyklad 7 =")
    przyklad7()
    print("\nPrzyklad 8 =")
    przyklad8()
    print("\nPrzyklad 9 =")
    przyklad9()
    print("\nPrzyklad 10 =")
    przyklad10()

main()
```

Wyniki:

```
Przyklad 1 =

Something is happening before the function is called.

Whee!

Something is happening after the function is called.


Przyklad 2 =

Something is happening before the function is called.

Whee!

Something is happening after the function is called.
```

```
Przyklad 3 =
test
test

Przyklad 4 =
Calling decorated function
Called example function
example
Docstring

Przyklad 5 =
Wywołanie funkcji dekorowanej
Funkcja dekorowana
Koniec wywołania funkcji dekorowanej
example
Dokumentacja funkcji

Przyklad 6 =

Przyklad 7 =
Setting: john smith
Getting: John Smith
John Smith
Deleting: John Smith

Przyklad 8 =
Getting value
```

```
GeeksforGeeks
Setting value to GfG
Deleting value

Przyklad 9 =
Setting: john smith
Getting: John Smith
John Smith
>Deleting: John Smith

Przyklad 10 =
Getting value
Peter
Setting value to Diesel
Deleting value
```

Zad 2:

```
def p1():
    def my_decorator(f):
        def wrapper(*args, **kwds):
            """Wraper doscstring"""
            print('Calling decorated function')
            return f(*args, **kwds)
        return wrapper
    @my_decorator
    def example():
        """Docstring"""
        print('Called example function')
    example()
    print(example.__name__)
    print(example.__doc__)

def p2():
    def my_decorator(f):
        @wraps(f)
```

```
        def wrapper(*args, **kwds):
            print('Calling decorated function')
            return f(*args, **kwds)
        return wrapper
    @my_decorator
    def example():
        """Docstring"""
        print('Called example function')
    example()
    print(example.__name__)
    print(example.__doc__)

print("\nPrzyklad 1 =")
p1()
print("\nPrzyklad 2 =")
p2()
```

Wyniki:

```
Przyklad 1 =

Calling decorated function

Called example function

wrapper

Wraper doscstring


Przyklad 2 =

Calling decorated function

Called example function

example

Docstring
```

Zad 3:

```
"""
  Program tworzy dekorator z parametrem i funkcje dekorujaca która
dadaje liczby.
"""
def example(funkcja):
    def wrapper(*args, **kwargs):
        print("Poczatek wywolania: ")
        funkcja(*args, **kwargs)
        print("Po wywolaniu: ")
    return wrapper
@example
def dodaj(liczba1, liczba2):
    print("Wynik = ",liczba1 + liczba2)

"""
Program tworzy dekorator z parametrem i funkcje dekorujaca która
wykonuje działania na liczbach.
"""
def example2(funkcja):
    def wrapper(*args, **kwargs):
        print("Poczatek wywolania: ")
        funkcja(*args, **kwargs)
        print("Po wywolaniu: ")
    return wrapper
@example2
def wykonaj(a, b, c, d):
    print ("Wynik = ",a * b + c / d )


print("\nPrzyklad 1")
dodaj(95,5)
print("\nPrzyklad 2")
wykonaj(2,9,56,2)
```

Wyniki:

```
Przyklad 1 =

Poczatek wywolania:

Wynik =  100
```

```
Po wywolaniu:


Przyklad 2 =
Poczatek wywolania:
Wynik =  46.0
Po wywolaniu:
```

Zad 4:

```
"""
Przyklad tworzy deskryptor przy uzyciu funkcji property. Deskryptor
sprawdza przy przypisywaniu wartość
czy liczba jest typu int i czy jest mniejsza od 219.
"""
class Descriptor:
    def __init__(self, wartosc=None):
        self.__liczba = wartosc

    def get_liczba(self):
        try:
            return "liczba = {}".format(self.__liczba)
        except:
            print("Liczba nie jest zadekralorowane")

    def set_liczba(self, value):
        if isinstance(value, int):
            if value < 219:
                print("Zaktualizowano liczbe ")
                self.__liczba = value
            else:
                raise ValueError("Liczba musi byc mniejsza od 219")
        else:
            raise TypeError("Liczba musi byc typu int")

    def delete_liczba(self):
        print("Usunieto atrybut")
        try:
            del self.__liczba
        except:
            print("Liczba nie jest zadekralorowane")
```

```
    liczba = property(get_liczba, set_liczba, delete_liczba)

print("\nPrzyklad 1 =")
p2 = Descriptor(78)
p2.liczba = 7
print(p2.liczba)
del p2.liczba
```

Wynik:

```
Przyklad 1 =

Zaktualizowano liczbe

liczba = 7

Usunieto atrybut
```

Zad 5:

```
"""
Przyklad tworzy deskryptor przy uzyciu dekoratora property. Deskryptor
sprawdza przy przypisywaniu wartość
czy liczba jest typu int i czy jest mniejsza od 146.
"""
class Descriptor:
    def __init__(self, wartosc=None):
        self.__liczba = wartosc

    def get_liczba(self):
        try:
            return "liczba = {}".format(self.__liczba)
        except:
            print("Liczba nie jest zadekralorowane")

    def set_liczba(self, value):
        if isinstance(value, int):
            if value < 146:
                print("Zaktualizowano liczbe ")
                self.__liczba = value
            else:
                raise ValueError("Liczba musi byc mniejsza od 219")
        else:
            raise TypeError("Liczba musi byc typu int")
```

```
    def delete_liczba(self):
        print("Usunieto atrybut")
        try:
            del self.__liczba
        except:
            print("Liczba nie jest zadekralorowane")

    liczba = property(get_liczba, set_liczba, delete_liczba)

print("\nPrzyklad 1 =")
p2 = Descriptor(99)
p2.liczba = 20
print(p2.liczba)
del p2.liczba
```

Wynik:

```
Przyklad 1 =

Zaktualizowano liczbe

liczba = 20

Usunieto atrybut
```