Kurs rozszerzony języka Python

Wykład 13.

Marcin Młotkowski

14 stycznia 2025

Plan wykładu

Plan wykładu

Rozszerzanie właściwości funkcji

```
def szalenie_skomplikowana_funkcja(arg1, arg2, arg3):
    ...
```

Śledzenie wywołania funkcji

Chcemy śledzić wywołania zaimplementowanych funkcji, tj. informacje o wywołaniu oraz informacja o argumentach wywołania. Bez ingerowania w te funkcje.

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print(f"Wywoływana funkcja: foo z argumentami {args}")
    return foo(*args)
```

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print(f"Wywoływana funkcja: foo z argumentami {args}")
    return foo(*args)
```

Co z tym zrobić 1.

Zamiast *foo* używamy *log_foo*.

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print(f"Wywoływana funkcja: foo z argumentami {args}")
    return foo(*args)
```

Co z tym zrobić 1.

Zamiast *foo* używamy *log_foo*.

Co z tym zrobić 2.

foo = log_foo

Ale prosimy się o kłopoty!

Uniwersalna funkcja opakowująca inne funkcje

Funkcja zwracająca inną funkcję

```
def log(fun):
    def opakowanie(*args):
        print("funkcja:", fun.__name__, "argumenty:", args
        return fun(*args)
    return opakowanie
```

Uniwersalna funkcja opakowująca inne funkcje

Funkcja zwracająca inną funkcję

```
def log(fun):
    def opakowanie(*args):
        print("funkcja:", fun.__name__, "argumenty:", args
        return fun(*args)
    return opakowanie
```

Zastosowanie

foo = log(foo)

Dekoratory

```
def log(fun):
    def opakowanie(*args):
        print(f"{fun.__name__} {args}")
        return fun(*args)
    return opakowanie
```

Dekoratory

```
def log(fun):
    def opakowanie(*args):
        print(f"{fun.__name__} {args}")
        return fun(*args)
    return opakowanie
```

Zastosowanie

```
@log
def fib(args):
    ...
```

Pomiar czasu wykonania funkcji

```
import time

def wydajnosc(fun):
    def opakowanie(*args):
        t_start = time.perf_counter()
        wynik = fun(*args)
        czas = time.perf_counter() - t_start
        print(f"{fun.__name__} {args} {czas}")
        return wynik
    return opakowanie
```

Przykład z życia: keszowanie wyników

```
def cache_function(func):
    """Keszowanie zapytań"""
    def wrapper(conn, query, typ):
        # inicjowanie cache
        db = shelve.open("/tmp/query.cache")
        if query in db:
            res = db[query]
            return res
        res = func(conn, query, typ)
        db[query] = list(res)
        db.sync()
        res = db[query]
        return res
    return wrapper
```

Dekoratory standardowe

Dekorowanie programów wielowątkowych

```
from threading import Lock

my_lock = Lock()

@synchronized(my_lock)
def critical1(): ...

@synchronized(my_lock)
def critical2(): ...
```

Implementacja dekoratora

```
def synchronized(lock):
    def wrap(f):
        def new_function(*args, **kw):
            lock.acquire()
            try:
                return f(*args, **kw)
            finally:
                lock.release()
        return new_function
    return wrap
```

Plan wykładu

Instrukcja **with-as**

```
with open("file.txt", 'r') as fh:
    print(fh.read()
```

jest równoważne

```
try:
    fh = open("file.txt", 'r')
    print(fh.read())
except:
    # obsługa błędu
finally:
    fh.close()
```

context manager

Context manager to obiekt implementujący dwie metody:
`__enter__(self)` i
`__exit__(self, exc_type, exc_val, exc_tb)`

```
open("file.txt", 'r')
```

Funkcja `open()` nie tylko zwraca uchwyt do pliku, ale także zarządzanie kontekstem.

context manager

Context manager to obiekt implementujący dwie metody:
`__enter__(self)` i
`__exit__(self, exc_type, exc_val, exc_tb)`

```
open("file.txt", 'r')
```

Funkcja `open()` nie tylko zwraca uchwyt do pliku, ale także zarządzanie kontekstem.

```
help(open("plik.txt", 'w'))
```

Podsumowanie

Context manager implementuje szkielet zarządzania kontekstem, instrukcja `with-as` zarządza tym kontekstem.

Przykład: pomiar czasu grupy instrukcji

```
with Pomiar("Pomiar czasu instrukcji"):
    fib(n)
```

Implementacja

```
class Pomiar:
    def __init__(self, nazwa):
        self.nazwa = nazwa

    def __enter__(self):
        self.t = time.perf_counter_ns()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        czas = time.perf_counter_ns() - self.t
        print(f"Pomiar bloku {self.nazwa} {czas:0.2f}ns")
        return True
```

Wasze programy :)

```
engine = create_engine('sqlite:///wyklad.db', echo=True)
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)

@app.route
def dodaj(osoba):
    sesja = Session()
    sesja.add(osoba)
    sesja.commit()

@app.route
def usun(osoba):
    sesja = Session()
    sesja.delete(osoba)
    sesja.commit()
```

Zadanie

Chcę mieć obsługę żądań zapakowaną w kontekście i jeszcze mieć logowanie błędów.

Definiujemy klasę

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s',
        filename='myapp.log', level=logging.INFO)

class AlchSession:
    def __enter__(self):
        logging.info("Początek sesji alchemy")
        self.session = Session()
        return session

    def __exit__(self, exc_type, exc_val, exc_tb):
        logging.info("Koniec sesji alchemy")
        if exc_val is not None:
            logging.info(f"{exc_type}")
            self.session.rollback()
        else:
            self.session.commit()
        self.session.close()
        return True
```

Nowa implementacja

Stary kod

```
@app.route
def dodaj(osoba):
    sesja = Session()
    sesja.add(osoba)
    sesja.commit()
```

zamieniamy na

```
@app.route
def dodaj(osoba):
    with AlchSession() as session:
        sesja.add(osoba)
```

Context manager + generatory + dekoratory

Fabryki menadżerów kontekstu można tworzyć za pomocą dekoratorów.

Implementacja

```
from contextlib import contextmanager

@contextmanager
def zarzadzanie_kontekstem(*args, **kwargs):
    f = open("plik.txt", 'w')
    try:
        yield f
    finally:
        f.close()
```

Wykorzystanie

```
with zarzadzanie_kontekstem(timeout=3600) as session:
    session.write("A kuku!")
```

Plan wykładu

Wproawadzenie

W Pythonie wszystko jest obiektem!

Wproawadzenie

W Pythonie wszystko jest obiektem!

A klasy?

Trochę eksperymentów

```
class Foo:
    pass

f = Foo()
f.__class__
F.__class__
```

Trochę szczegółów

Metaklasa to szablon tworzenia klas.

Trochę więcej o klasach

- `__init__(self)` inicjalizacja obiektu;
- `__new__(cls)` tworzy instancję klasy i przekazuje do `__init__`.

Te metody są wywoływane przez metotę statyczną `__call__`.

Do czego są metaklasy

Np. automatyczne dodawanie metod czy pól statycznych do klas.

Jak

Przykład:
wybrane klasy powinne zliczać swoje instancje.

Budujemy metaklasę

```
class Zliczanie(type):
    def __new__(cls, name, bases, dct):
        x = super().__new__(cls, name, bases, dct)
        x.licznik = 0
        x.ile = lambda : x.licznik
        return x

    def __call__(cls, *args, **kwargs):
        instance = super().__call__(*args, **kwargs)
        cls.licznik += 1
        return instance
```

Jak zastosować

```
class Foo(metaclass=Zliczanie):
    pass
```