Akademia Pythona

VIII Zagadnienia zaawansowane

KN Pythona - Kurs Pythona

KN Pythona wita na kursie Pythona.

Plan:

- Metaklasy

Zaawansowane zagadnienia Pythona

Zaawansowane zagadnienia Pythona:

- Atrybuty introspektywne
- Metody przeciążania operatorów
- Metody przechwytywania atrybutów
- Właściwości klas
- Deskryptory atrybutów klas
- Dekoratory funkcji i klas
- Metaklasy

Zaawansowane narzędzia Pythona

Narzędzia te umożliwiają na wstawianie automatycznie wykonywanego kodu w różnych kontekstach.

Metaklasy

Metaklasy wykonywane są w momencie tworzenia klasy. Służą one do zarządzania klasami i rozszerzania ich.

Metaklasy

Przykładowe zastosowania:

- Śledzenie
- Trwałość obiektów
- Logowanie wyjątków
- Konstruowanie klasy w oparciu o konfiguracje
- Uniwersalne stosowanie dekoratorów
- Sprawdzanie zgodności z oczekiwanymi interfejsami
- ORM
- Bazy danych

Metaklasy a dekoratory klas

Role metaklas i dekoratorów klas pokrywają się w wielu przypadkach. Metaklasy udostępniają jednak formalny model przygotowany z myślą o zarządzaniu klasami.

Wady funkcji pomocniczych

Zalety metaklas:

- Udostępniają formalną i jawną strukturę
- Faktoryzacja / redukcja powtarzalności

Dziedziczenie

```
class Extras:
    def extra(self, args):
        ...
class Client1(Extras): ...
class Client2(Extras): ...
class Client3(Extras): ...
x = Client1()
x.extra()
```

Rozszerzanie

```
def extra(self, args): ...
class Client1: ...
if required():
    Client1.extra = extra
class Client2: ...
if required():
    Client2.extra = extra
class Client3: ...
if required():
    Client3.extra = extra
x = Client1()
x.extra()
```

Funkcje pomocnicze

```
def extra(self, args): ...
def extras(Class):
    if required():
        Class.extra = extra
class Client1: ...
extras(Client1)
class Client2: ...
extras(Client2)
class Client3: ...
extras(Client3)
x = Client1()
x.extra()
```

Metaklasy

```
def extra(self, args): ...
class Extras(type):
    def __init__(Class, clsname, supercls, attrs):
        if required():
            Class.extra = extra
class Client1(metaclass=Extras): ...
class Client2(metaclass=Extras): ...
class Client3(metaclass=Extras): ...
x = Client1()
x.extra()
```

Dekoratory

```
def extra(self, arg): ...
def extras(Class):
    if required():
        Class.extra = extra
    return Class
@extras
class Client1: ...
@extras
class Client2: ...
@extras
class Client3: ...
x = Client1()
x.extra()
```

Metaklasy a dekoratory

Metaklasy - zaawansowane zarządzanie klasami

Dekoratory - zarządzanie klasami i instancjami

Model metaklasy

Klasy są instancjami obiektu type(który sam jest klasą).

Model metaklasy

```
type([]) # <class 'list'>
type(type([])) # <class 'type'>
type(list) # <class 'type'>
type(type) # <class 'type'>
```

Model metaklasy

Model typów w Pythonie:

- Typy definiowane są przez klasy pochodzące od type
- Klasy zdefiniowane przez użytkowników są instancjami klas typów
- Klasy zdefiniowane przez użytkowników są typami generującymi instancje.

Model metaklasy

```
class C: pass
X = C()
type(X) # <class '__main__.C'>
X.__class__ # <class '__main__.C'>
type(C) # <class 'type'>
C.__class__ # <class 'type'>
```

Model metaklasy

Model metaklasy:

- type jest klasą generującą klasy zdefiniowane przez użytkownika
- Metaklasy są klasami podrzędnymi klasy type
- Obiekty klas są instancjami klasy type
- Obiekty instancji są generowane z klas

Protokół instrukcji class

```
class C: ...
# wykonanie zagnieżdżonego bloku kodu
C = type(cls_name, base_cls, attr_dict)
# obiekt type wywołuje:
type.__new__(type_cls, cls_name, base_cls, attr_dict)
type.__init__(cls, cls_name, base_cls, attr_dict)
```

Protokół instrukcji class

```
class Spam(Eggs):
    data = 1
    def meth(self, arg):
        pass
Spam = type('Spam', (Eggs,),
    {'data': 1, 'meth': meth, '__module__': '__main__'})
```

Metaklasy

```
class Spam(metalcass=Meta): ...
Spam = Meta('Spam', (,), {})
# Meta.__call__ =>
Meta.__new__(Meta, cls_name, bases, attr_dict)
Meta.__init__(cls, cls_name, bases, attr_dict)
```

Metaklasy

```
class Meta(type):
    def __new__(meta, cls_name, bases, attr_dict):
        return type.__new__(...)
    def __init__(cls, cls_name, bases, attr_dict):
        # dostosowanie tworzenia klasy
        return cls
class A: ...
class B(A, metaclass=Meta): ...
```

Funkcje fabryczne jako metaklasy

```
def MetaFunc(cls_name, bases, attr_dict):
    return type(cls_name, bases, attr_dict)
class A: ...
class B(A, metaclass=MetaFunc): ...
```

Przeciążanie wywołań tworzących klasę

```
class SuperMeta(type):
    def __call__(meta, cls_name, bases, attr_dict):
        return type.__call__(meta, cls_name,
                             bases, attr_dict)
class SubMeta(type, metaclass=SuperMeta):
    def __new__(...): ...
    def __init__(...): ...
class B(metaclass=SubMeta): ...
```

Instancje a dziedziczenie

Instancje a dziedziczenie:

- Metaklasy dziedziczą po klasie type
- Deklaracje metaklas dziedziczone są przez klasy podrzędne
- Atrybuty metaklas nie są dziedziczone przez instancje klas

Rozszerzanie oparte na metaklasie

```
class Extender(type):
    def __new__(meta, cls_name, bases, attr_dict):
        attr_dict['func'] = func
        attr_dict['x'] = x
        return type.__new__(meta, cls_name,
                bases, attr_dict)
class A: ...
a = A()
a.func()
a.x
```

Zarządzanie instancjami zamiast klasami

```
def Tracer(cls_name, bases, attr_dict):
    a_cls = type(cls_name, bases, attr_dict)
    class Wrapper:
        def __init__(self, *args, **kwargs):
            self.wrapped = a_cls(*args, **kwargs)
        def __getattr__(self, attrname):
            return getattr(self.wrapped, attrname)
    return Wrapper
def A(metaclass=Tracer): ...
```

Śledzenie wywołań

```
class MetaTracer(type):
    def __new__(meta, cls_name, bases, attr_dict):
        for attr, val in attr_dict.items():
            if type(val) is FunctionType:
                attr_dict[attr] = tracer(val)
        return type.__new__(meta, cls_name,
                bases, attr_dict)
```

Zastosowanie dowolnego dekoratora

```
def decorateAll(deco):
    class MetaDecorate(type):
        def __new__(mt, cn, bs, ad):
            for a, v in ad.items():
                if type(v) is FunctionType:
                    ad[a] = deco(v)
            return type.__new__(mt, cn, bs, ad)
class Person(metaclass=decorateAll(tracer)): ...
```

Koniec

Dziękuję za uwagę i gratulacje!