Akademia Pythona

VI Klasy i programowanie zorientowane obiektowo

KN Pythona - Kurs Pythona

KN Pythona wita na kursie Pythona.

Plan:

- Zaawansowane zagadnienia związane z klasami

Rozszerzanie typów wbudowanych

Technika:

- Rozszerzanie za pomocą osadzania
- Rozszerzanie za pomocą klas podrzędnych

Rozszerzanie typów za pomocą osadzania

```
class Set:
    def __init__(self, value = []):
        self.data = []
        self.concat(value)

    def concat(self, value):
        for x in value:
            if x not in self.data: # Powolne!
                self.data.append(x)
```

Rozszerzanie za pomocą osadzania

```
...
def union(self, other):
    res = self.data[:]
    for x in other:
        if not x in res:
            res.append(x)
    return Set(res)

def intersect(self, other):
    res = []
    for x in other:
        if x in other:
            res.append(x)
    return Set(res)
```

Rozszerzanie za pomocą osadzania

```
def __len__(self):
    return len(self.data)
def __getitem__(self, key):
    return self.data[key]
def __and__(self, other):
    return self.intersect(other)
def __or__(self, other):
    return self.union(other)
def __repr__(self):
    return f"Set: {repr(self.data)}"
```

Rozszerzanie za pomocą klas podrzędnych

```
# Lista indeksowana od 1..N
class MyList(list):
    def __getitem__(self, offset):
        print(f"Indeksowanie {self} na pozycji {offset}")
        return list.__getitem__(self, offset - 1)
```

Klasy w nowym stylu

Od Pythona 3.0 wszystkie klasy są automatycznie tworzone jako klasy w nowym stylu - dziedziczą po klasie **object** niezależnie od podania jawnej deklaracji class A(object).

Połączenie klas i typów

Klasy są typami, a typy są klasami. Funkcja wbudowana type(I) zwróci klasę, z której utworzono obiekt I, nie generyczny typ i najczęściej jest to ta sama klasa, którą zawiera I.__class__.

Połączenie klas i typów

Co więcej, klasy są instancjami klasy type; Można również tworzyć klasy potomne klasy type, co pozwala na dostosowanie do własnych potrzeb mechanizmu tworzenia klas. Wszystkie klasy dziedziczą po klasie object i to samo dotyczy klasy type.

Klasa typu

```
class A:
    pass

x = A()
print(type(x)) # <class '__main__.A'>
```

Klasa typu

```
A = type("A", (), {})
x = A()
print(type(x)) # <class '__main__.A'>
```

Klasa typu

```
def outside_init(self, name):
    self.name = name
Robot = type('Robot', (), {"__init__": outside_init,
                           "say_hello": lambda self: f"Hello {self.name
```

Dziedziczenie diamentowe

W dziedziczeniu diamentowym wyszukiwanie w drzewie klas zachodzi najpierw wszerz, od lewej do prawej.

Sloty

```
class limiter:
    __slots__ = ['age', 'name', 'job']

x = limiter()
x.age # Attribute error: age
x.age = 42
x.age # 42
x.ape = 1000 # Attribute error
```

Sloty

```
class D:
    __slots__ = ['a', 'b', '__dict__']

d = D()
d.a, d.b, d.c = 1, 2, 3
```

Właściwości klas

```
class newprops:
    def getage(self):
        return 40
    def setage(self, value):
        print("Ustawianie wieku " + str(value))
        self._age = value
    age = property(getage, setage, None, None)
```

Metody statyczne i metody klas

```
class Methods:
    def imeth(self, x):
        print(self, x)
    def smeth(x):
        print(x)
    def cmeth(cls, x):
        print(cls, x)
    smeth = stathicmethod(smeth) # ew. wstawienie funkcji
    cmeth = classmethod(cmeth)
```

Metody statyczne

```
class A:
    count = 0
    def __init__(self):
        A.count += 1
    def print_count():
        print(A.count)
    print_count = stathicmethod(print_count)
```

Metody klas

```
class B:
    count = 0
    def __init__(self):
        B.count += 1
    def print_count(cls):
        print(cls.count)
    print_count = classmethod(print_count)
```

Metody klas

```
class C:
    count = 0
    def count(cls):
        cls.count += 1
    def __init__(self):
        self.count()
    count = classmethod(count)
```

Dekoratory - wstęp

```
class E:
    def meth():
        pass
    meth = staticmethod(meth)
```

Dekoratory - wstęp

```
class E:
    @staticmethod
    def meth():
        pass
```

Dekoratory

```
class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print(f"{self.calls} call of func")
        return self.func(*args, **kwargs)
```

Dekoratory

```
@tracer
def spam(a):
    print(a)

spam(1)
spam(2)
spam(3)
```