Akademia Pythona

VI Klasy i programowanie zorientowane obiektowo

KN Pythona - Kurs Pythona

KN Pythona wita na kursie Pythona.

Plan:

- Programowanie zorientowane obiektowo
- Podstawy tworzenia klas

Programowanie zorientowane obiektowo

Kod w Pythonie:

- Oparty na obiektach
- Zorientowany obiektowo

Po co używa się klas?

Rola klas:

- Dziedziczenie
- Kompozycja

Charakterystyka klas

Cechy klas:

- Wiele instancji
- Dostosowywanie do własnych potrzeb dzięki dziedziczeniu
- Przeciążanie operatorów

Programowanie zorientowane obiektowo

```
obiekt.atrybut
```

Znajdź pierwsze wystąpienie **atrybuty**, szukając w **obiekcie**, a następnie we wszystkich klasach powyżej niego, od dołu do góry i od lewej strony do prawej.

Klasy

Klasy służą jako fabryki instancji. Ich atrybuty udostępniają zachowanie: dane oraz funkcje dziedziczone przez wszystkie instancje z nich wygenerowane.

Instancje

Instancje reprezentują konkretne elementy w domenie programu. Ich atrybuty zapisują dane różniące się dla określonych obiektów.

Klasy

```
class Klasa(S1, S2):
    pass
```

Programowanie zorientowane obiektowo

Tworzenie drzew klas:

- Każda instrukcja **class** generuje nowy obiekt klasy.
- Za każdym razem gdy klasa jest wywoływana, generuje ona nowy obiekt instancji.
- Instancje są automatycznie połączone z klasami, przez które zostały utworzone.
- Klasy są połączone ze swoimi klasami nadrzędnymi dzięki podaniu ich w nawiasach w wierszu nagłówka instrukcji **class**. Uporządkowanie od lewej do prawej podaje kolejność w drzewie.

Drzewa klas

```
class C2: pass
class C3: pass
class C1(C2, C3): pass # dziedziczenie wielokrotne

I1 = C1()
I2 = C2()
```

Wywołanie metody

```
I2.w()
<==>
C3.w(I2)
```

Atrybuty obiektów

Atrybuty obiektów:

- Atrybuty są zazwyczaj dołączane do klas poprzez przypisanie wykonane wewnątrz instrukcji **class** i nie zagnieżdżone dodatkowo wewnątrz instrukcji **def**.
- Atrybuty są zazwyczaj dołączane do instancji przez przypisanie do specjalnego argumentu przekazanego do funkcji wewnątrz klasy i noszącego nazwę **self**.

Atrybuty

```
class C1(C2, C3):
    def setname(self, who):
        self.name = who
I1 = C1()
I2 = C1()
I1.setname('Amadeusz')
I2.setname('Alexander')
print(I1.name) # Amadeusz
```

Konstruktor

```
class C1(C2, C3):
    def __init__(self, who): # Przeciążenie operatora
        self.name = who

I1 = C1('Amadeusz')
I2 = C2('Alexander')
print(I1.name) # Amadeusz
```

Prosty przykład

```
class Employee:
    def computeSalary(self): ...
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```

Prosty przykład

```
class Engineer(Employee):
    def computeSalary(self): ... # Coś innego
```

Prosty przykład

```
amadeusz = Employee()
alexander = Engineer()
company = [amadeusz, alexander]
for emp in company:
    print(emp.computeSalary())
```

Inny przykład

```
def processor(reader, converter, writer):
    while 1:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Inny przykład

```
class Reader:
    def read(self): pass

class FileReader:
    def read(self): pass

class SocketReader:
    def read(self): pass
```

Inny przykład

```
processor(FileReader(), Converter, FileWriter)
processor(SocketReader(), Converter, TapeWriter())
processor(FtpReader(), Converter, XmlWriter())
```

Programowanie zorientowane obiektowo

Cechy klas:

- Klasy są przestzeniami nazwę
- Generują wiele obiektów
- Umożliwiają dziedziczenie przestrzeni nazw
- Umożliwiają przeciążanie operatorów

Obiekty instancji

Obiekty instancji są rzeczywistymi elementami:

- Wywołanie obiektu klasy w sposób podobny do wywołania funkcji tworzy nowy obiekt instancji.
- Każdy obiekt instancji dziedziczy atrybuty klasy oraz otrzymuje własną przestrzeń nazw.
- Przypisania do atrybutów **self** w metodach tworzą atrybuty instancji.

Przykład

```
class FirstClass:
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)
```

Przykład

```
x = FirstClass()
y = FirstClass()
x.setdata('MoSM')
y.setdata(3.41159)
x.display() # MoSM
y.display() # 3.41159
```

Przykład

```
x.data = 'new value'
x.display() # new value
x.second_data = 'second value'
```

Dziedziczenie

Dziedziczenie:

- Klasy nadrzędne są wymieniane w nawiasach w nagłówku instrukcji **class**.
- Klasy dziedziczą atrybuty po swoich klasach nadrzędnych.
- Instancje dzidziczą atrybuty po wszystkich dostępnych klasach.
- Każda referencja **obiekt.atrybut** wywołuje nowe, niezależne wyszukiwanie.
- Zmiany logiki wykonuje się przez tworzenie klas podrzędnych, a nie modyfikację klas nadrzędnych.

Przykład

```
class SecondClass(FirstClass):
    def display(self):
        print(f'Value: {self.data}')

z = SecondClass()
z.setdata(42)
z.display() # Value: 42
```

Przeciążanie operatorów

Przeciążanie operatorów:

- Metody zawierające w nazwie podwójne znaki _ (**X**) są specjalnymi punktami zaczepienia.
- Takie metody wywoływane są automatycznie, kiedy instancje pojawiają się w operacjach wbudowanych.
- Klasy mogą nadpisywać większość operacji na typach wbudowanych.
- Nie istnieją wartości domyślne dla metod przeciążania operatorów i nie są one potrzebne.
- Operatory pozwalaja klasom na integrację z modeleme obiektów Pythona.

Przykład

```
class ThirdClass(SecondClass):
    def __init__(self, value):
        self.data = value
    def __add__(self, other):
        return ThirdClass(self.data + other)
    def __str__(self):
        return f'Third Class: {self.data}'
    def mul(self, other):
        self.data *= other
```

Przykład

```
a = ThirdClass('abc')
a.display() # Value: abc
b = a + 'xyz'
b.display() # Value: abcxyz
print(b) # Third Class: abcxyz
a.mul(3)
print(a) # Third Class: abcabcabc
```