Akademia Pythona

VI Klasy i programowanie zorientowane obiektowo

KN Pythona - Kurs Pythona

KN Pythona wita na kursie Pythona.

Plan:

- Bardziej realistyczny przykład
- Szczegóły kodu klas

Przykład

Utworzymy kod dwóch klas:

- **Person**: klasa tworząca i przetwarzająca informacje o osobach
- **Manager**: dostosowywanie klasy **Person** do własnych potrzeb, modyfikując odziedziczone działanie.

Przykład - Tworzenie konstruktorów

```
class Person:
    def __init__(self, name, job, pay):
        self.name = name
        self.job = job
        self.pay = pay
```

Przykład - Tworzenie konstruktorów

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
```

Przykład

```
if __name__ == '__main__':
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona',
                  job='Programista', pay=10**12)
    print(bob.name, bob.pay)
    print(anna.name, anna.pay)
```

Przykład

```
:> python person.py
Robert Zielony 0
Anna Czerwona 1000...

:> python
>>>import person
>>>
```

Przykład

```
class Person:
    ...
    def last_name(self):
        return self.name.split()[-1]

    def give_raise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Przykład

```
if __name__ == '__main__':
    ...
    print(bob.last_name(), anna.last_name())
    anna.give_raise(.10)
    print(anna.pay)
```

Przykład

```
class Person:
    ...
    def __str__(self):
        return f'[Person: {self.name} {self.pay}]'
```

Przykład

Przęciążona metoda **str** jest wywoływana przez **print**. W wierszu interaktywnym wywoływana jest metoda **repr**.

Przykład - źle

```
class Manager(Person):
    def give_raise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus))
```

Przykład - dobrze

```
class Manager(Person):
    def give_raise(self, percent, bonus=.10):
        Person.give_raise(self, percent + bonus)
```

Przykład

```
instancja.metoda(argumenty, ...)
```

vs

```
klasa.metoda(instancja, argumenty, ...)
```

Przykład

```
if __name__ == '__main__':
    ...
    tom = Manager('Tomasz Czarny', 'manager', 50000)
    tom.give_raise(.10)
    print(tom.last_name())
    print(tom)
```

Przykład

```
if __name__ == '__main__':
    ...
    print('---Wszystkie trzy---')
    for object in (bob, anna, tom):
        object.give_raise(.10)
        print(object)
```

Przykład

```
class Manager(Person):
    def __init__(self, name, pay):
        Person.__init__(self, name, 'manager', pay)
    ...
```

Programowanie obiektowe

Użyte koncepcje:

- Tworzenie instancji: wypełnianie atrybutów instancji.
- Metody i działanie: hermetyzacja logiki w metodach klas.
- Przeciążanie operatorów: udostępnianie działania operacjom wbudowanym, takim jak wyświetlanie.
- Dostosowywanie działania do własnych potrzeb: redefiniowanie metod w klasach podrzędnych w celu ich specjalizacji.
- Dostosowywanie konstruktorów do własnych potrzeb: dodanie logiki inicjalizującej do kroków z klasy nadrzędnej.

Przykład

```
class Department:
    def __init__(self, *args):
        self.members = list(args)
    def add_member(self, person):
        self.members.append(person)
    def give_raises(self, percent):
        for person in self.members:
            person.give_raise(percent)
    def show_all(self):
        for person in self.members:
            print(person)
```

Introspekcja klas

```
bob.__class__
bob.__class__.__name__
list(bob.__dict__.keys())
```

Przykład

```
class AttrDisplay:
    def gather_attrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append(f'{key}: {getattr(self, key)}')
        return '\n'.join(attrs)
    def __str__(self):
        return '{0}\n{1}'.format(self.__class__.__name__,
            self.gather_attrs())
```

Przykład

Klasa gotowa.

Szczegóły kodu klas

```
class <nazwa>(klasa nadrzędna, ...):
    data = value
    def method(self, ...):
        self.member = value
```

Atrybuty

```
class SharedData:
    x = 42
x = SharedData()
y = SharedData()
x.x, y.x # 42, 42
```

Atrybuty

```
SharedData.x = 99
x.x, y.x, SharedData.x # 99, 99, 99
```

Atrybuty

```
x.x = 88
x.x, y.x, SharedData.x # 88, 99, 99
```

Metody

```
class NextClass:
    def printer(self, text):
        self.message = text
        print(self.message)
```

Metody

```
x = NextClass()

x.printer('Wywołanie instancji')

x.message # 'Wywołanie instancji'
```

Metody

```
NextClass.printer(x, 'Wywołanie klasy')

x.message # 'Wywołanie klasy'
```

Konstruktory

```
class Super:
    def __init__(self, x):
        ...kod domyślny...

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)
        ..własny kod..
```

Specjalizacja metod

```
class Super:
    def method(self):
        print('w Super method.')

class Sub(Super):
    def method(self):
        print('Początek sub method')
        Super.method(self)
        print('Koniec sub method')
```

Klasy abstrakcyjne

```
from abc import ABCMeta, abstractmethod

class Super(metaclass=ABCMeta):
    def delegate(self):
        self.action()
    @abstractmethod
    def action(self):
        pass
```