Olio-ohjelmointi 3

Edellisen päivän harjoituksen ratkaisu

Tehdään yhdessä edellisen päivän Vector-tehtävää.

Olio-ohjelmoinnin periatteet

Olio-ohjelmointi periatteet ovat:

1. **Kapselointi** (Encapsulation): Tämä periaate viittaa tietojen suojaamiseen ja objektin sisäisen tiedon piilottamiseen. Tietoihin pääsee käsiksi vain julkisten ja turvallisten metodien kautta, mikä takaa tiedon eheyden.
2. **Periytyminen** (Inheritance): Periytyminen mahdollistaa uusien luokkien luomisen olemassa olevien luokkien pohjalta. Tämä helpottaa koodin uudelleenkäyttöä ja luokkahierarkioiden luomista, jotka jakavat yhteisiä ominaisuuksia.
3. **Polymorfismi** (Polymorphism): Polymorfismi mahdollistaa sen, että eri objektit vastaavat samalla viestillä tai metodilla asianmukaisesti. Tämä saavutetaan metodien ylikuormituksella ja rajapintojen toteutuksella.
4. **Abstraktio** (Abstraction): Abstraktio yksinkertaistaa järjestelmän monimutkaisuutta piilottamalla toteutuksen yksityiskohdat ja näyttämällä vain olennaiset toiminnot. Tämä auttaa keskittymään siihen, mitä objekti tekee, sen sijaan, miten se tekee sen.

Nämä periaatteet ovat olennaisia, jotta voidaan hallita olio-ohjelmointia ja luoda vankkaa ja ylläpidettävää ohjelmistoa.

Periytyminen

Periytyminen Pythonissa mahdollistaa uusien luokkien (lapsiluokkien) luomisen olemassa olevien luokkien (emoluokkien) pohjalta. Tämä auttaa koodin uudelleenkäytettävyydessä ja ylläpidettävyydessä.

Perusperiaatteet

1. **Emoluokka**: Luokka, josta peritään attribuutteja ja metodeja.
2. **Lapsiluokka**: Luokka, joka perii attribuutteja ja metodeja emoluokasta.

```
# Emoluokka
class Eläin:
    def __init__(self, nimi):
        self.nimi = nimi

    def ääntele(self):
        print(f'{self.nimi} ääntelee.')

# Lapsiluokka
class Koira(Eläin):
    pass

# Lapsiluokka
class Kissa(Eläin):
    pass

koira = Koira('Rex')
koira.ääntele()  # Rex ääntelee.

kissa = Kissa('Felix')
kissa.ääntele()  # Felix ääntelee.
```

Mutta kyllä luodaan lapsiluokkia siksi, että halutaan muuttaa jollain tavalla emoluokan toiminnallisuus.

```python
# Emoluokka
class Eläin:
    def __init__(self, nimi):
        self.nimi = nimi

    def ääntele(self):
        print(f'{self.nimi} ääntelee.')

# Lapsiluokka
class Koira(Eläin):
    def ääntele(self):
        print(f'{self.nimi} haukkuu.')

# Lapsiluokka
class Kissa(Eläin):
    def ääntele(self):
        print(f'{self.nimi} maukuu.')

koira = Koira('Rex')
koira.ääntele()  # Rex haukkuu.

kissa = Kissa('Felix')
kissa.ääntele()  # Felix maukuu.
```

Periytyminen mahdollistaa myös moniperinnän, jossa lapsiluokka perii ominaisuuksia useammasta kuin yhdestä emoluokasta. Tämä voi olla hyödyllistä, mutta sitä kannattaa käyttää harkiten monimutkaisuuden välttämiseksi.

`super()`

Pythonin `super()`-funktio on hyödyllinen periytymisessä, koska sen avulla voidaan kutsua emoluokan metodeja lapsiluokasta. Tämä on erityisen kätevää, kun halutaan laajentaa tai ylikirjoittaa emoluokan toiminnallisuutta lapsiluokassa.

```python
class Eläin:
    def __init__(self, nimi):
        self.nimi = nimi

    def ääntele(self):
        print(f"{self.nimi} ääntelee.")

class Koira(Eläin):
    def __init__(self, nimi, rotu):
        super().__init__(nimi)  # Kutsuu Eläin-luokan __init__-metodia
        self.rotu = rotu        # Lapsiluokan oma attribuutti

    def ääntele(self):
        super().ääntele()       # Kutsuu Eläin-luokan ääntele-metodia
        print(f"{self.nimi} haukkuu.")

koira = Koira("Rex", 'Labradorinnoutaja')
koira.ääntele()     # Rex ääntelee. Rex haukkuu.
```

`super()`-funktiota käytetään yleisesti, kun halutaan varmistaa, että emoluokan alustukset ja metodit suoritetaan oikein lapsiluokassa.

Harjoitus (yhdessä)

Luo Person- ja Student-luokkia, jotka toimivat seuraavalla tavalla:

```
mika = Person('Mika', 'Suomalainen')
print(mika)                                  # <Person Mika Suomalainen>

seppo = Student('Seppo', 'Suomalainen')
print(f'{seppo} - Email: {seppo.email}')     # <Student Seppo Suomalainen (1)> - Email:
ss000001@edu.taitotalo.fi

chrisu = Teacher('Christian', 'Finnberg', 'Python')
print(f'{chrisu} - Email: {chrisu.email}')  # <Teacher Christian Finnberg> - Email:
christian.finnberg@taitotalo.fi

teijo = Student('Teijo', 'Tehokas')
print(f'{teijo} - Email: {teijo.email}')     # <Student Teijo Tehokas (2)> - Email:
tt000002@edu.taitotalo.fi
```

Moniperintä

Moniperintä Pythonissa tarkoittaa, että luokka voi periä ominaisuuksia ja metodeja useammasta kuin yhdestä yliluokasta. Tämä mahdollistaa monipuolisemman ja joustavamman koodin uudelleenkäytön. Moniperintä määritellään Pythonissa seuraavasti:

```python
class Luokka1:
    def metodi1(self):
        print("Metodi Luokka1:stä")

class Luokka2:
    def metodi2(self):
        print("Metodi Luokka2:sta")

class Aliluokka(Luokka1, Luokka2):
    pass

objekti = Aliluokka()
objekti.metodi1()  # Metodi Luokka1:stä
objekti.metodi2()  # Metodi Luokka2:sta
```

Moniperinnän käyttö Pythonissa voi olla hyödyllistä tietyissä tilanteissa, mutta sitä kannattaa käyttää harkiten.

Hyödyt:

- **Koodin uudelleenkäyttö**: Voidaan hyödyntää useiden luokkien ominaisuuksia ja metodeja ilman koodin toistamista.
- **Joustavuus**: Mahdollistaa monimutkaisempien rakenteiden luomisen ja eri luokkien yhdistämisen.

Haitat:

- **Monimutkaisuus**: Moniperintä voi tehdä koodista vaikeammin ymmärrettävää ja ylläpidettävää.
- **Timanttiperintäongelma**: Jos kaksi yliluokkaa perivät saman yliluokan, voi syntyä ongelmia, kun aliluokka perii molemmat. Python käyttää C3 Linearization -algoritmia ratkaistakseen tämän, mutta se voi silti aiheuttaa sekaannusta.

Yleisesti ottaen, jos moniperintä tuo selkeää hyötyä ja koodi pysyy hallittavana, sitä voi käyttää. Kuitenkin, usein kompositio (eli olioiden yhdistäminen) voi olla parempi vaihtoehto.

Timanttiongelma on osana ratkaistu `super()` avulla, mutta silti se voi olla sekävä

```
class Animal:
    def __init__(self, animal):
        print(animal, "is an animal")

class Mammal(Animal):
    def __init__(self, mammal):
        print(mammal, "is mammal animal")
        super().__init__(mammal)

class CanFlyAnimal(Animal):
    def __init__(self, can_fly_animal):
        print(can_fly_animal, 'can fly!')
        super().__init__(can_fly_animal)

class CanFlyMammal(Mammal, CanFlyAnimal):
    def __init__(self, can_fly_mammal):
        print(can_fly_mammal, "is a mammal that can fly!")
        super().__init__(can_fly_mammal)

bat = CanFlyMammal('bat')
```

Metodien "overloading" Pythonissa

Method overloading tarkoittaa sitä, että samaan luokkaan voidaan määritellä useita metodeja **samalla nimellä mutta eri parametreilla**. Tämä on yleistä kielissä kuten Java ja C++, mutta **Pythonissa ei ole perinteistä metodien overloadingia**.

Pythonissa funktiot ja metodit ovat dynaamisia. Jos määrittelaan saman nimisen metodin kahdesti, viimeisin määritelmä korvaa edellisen. Eli:

```
class Testi:
    def tulosta(self, x):
        print(x)

    def tulosta(self, x, y):
        print(x, y)

obj = Testi()
obj.tulosta(1, 2)  # OK
obj.tulosta(1)     # Virhe! (koska ensimmäinen versio korvattiin)
```

Kuitenkin, voidaan saavuttaa samanlaisen toiminnallisuuden käyttämällä erilaisia tekniikoita, kuten tarkistamalla argumenttien tyypit ja määrät manuaalisesti tai käyttämällä `singledispatch` tai `multipledispatch`-toimintoa.

Manuaalisesti

Ensimmäinen vaihtoehto olisi tehdä manuaalista tarkistusta argumenttien määrän perusteella:

```
class MyClass:
    def my_method(self, *args):
        if len(args) == 1 and isinstance(args[0], int):
            return self._my_method_int(args[0])
        elif len(args) == 2 and all(isinstance(arg, int) for arg in args):
            return self._my_method_two_ints(args[0], args[1])
        else:
            raise TypeError("Invalid arguments")

    def _my_method_int(self, x):
        return f"Single integer: {x}"

    def _my_method_two_ints(self, x, y):
        return f"Two integers: {x} and {y}"

obj = MyClass()
print(obj.my_method(10))  # Tulostaa: Single integer: 10
print(obj.my_method(10, 20))  # Tulostaa: Two integers: 10 and 20
```

Harjoitus

Toteutaa `__init__` -metodi seuraavalla koodilla, että se toimii:

```
class Point:
    def __init__(self, param1, param2=None) -> None:
        pass

p1 = Point(4, 5)    # Kaksi parametria: int 4 ja int 5
print(p1.x) # 4
print(p1.y) # 5
p2 = Point((7, 8))  # Yksi parametri: tupla (7, 8)
print(p2.x) # 7
print(p2.y) # 8
```

singledispatch

Toinen tapa on käyttää `singledispatch` -toimintoa. Se on kätevä työkalu funktioiden ylikuormittamiseen perustuen argumentin tyyppiin. Se löytyy `functools` -moduulista ja mahdollistaa eri funktioiden kutsumisen riippuen siitä, minkä tyyppinen argumentti funktiolle annetaan. Metodille pitää käyttää `singledispatchmethod` -funktio/decorator.

from functools import singledispatchmethod

```
class MyClass:
    @singledispatchmethod
    def my_method(self, value):
        print('Oletusmetodi', value)

    @my_method.register(int)
    def _(self, value):
        print('Int -metodi', value)

    @my_method.register(str)
    def _(self, value):
        print('Str -metodi', value)

my_object = MyClass()
my_object.my_method(10)
my_object.my_method('moi')
my_object.my_method([1,2,3])
```

Oletus -metodi on määritelty käyttämään `singledispatchmethod` -koristetta. Sitten määritellään erilliset metodit, jotka käsittelevät `int` - ja `str` -tyyppisiä argumentteja. Metodien määrittelyissä käytetään koristetta, joka perustuu oletus -metodin nimestä ja `.register(type)` -loppuosan. Jos metodia kutsutaan argumentilla, jolle ei ole määritelty erillistä käsittelijää, käytetään oletuskäsittelijää.

`singledispatch` kuitenkin pystyy käyttämään vain yhtä argumentia.

multipledispatch

Olisi myös mahdollista käyttää jotain kolmas-osa moduulia kuten esimerkiksi **multipledispatch**. Tämä kirjasto mahdollistaa funktioiden ylikuormittamisen useiden argumenttien tyyppien perusteella. Tämä tarkoittaa, että voidaan määritellä useita versioita samasta funktiosta, ja oikea versio valitaan automaattisesti argumenttien tyyppien perusteella.

Tämä kirjasto ei kuulu standard -Python ja pitää asentaa erikseen, esim. `pip install multipledispatch`.

```
from multipledispatch import dispatch

class MyClass:
    @dispatch(int, int)
    def process(self, a: int, b: int):
        print(f"Processing integers {a} and {b}")

    @dispatch(str, str)
    def process(self, a: str, b: str):
        print(f"Processing strings: '{a}' and '{b}'")

    @dispatch(bool, list)
    def process(self, a: bool, b: list):
        print(f"Processing bool {a} and list {b}")

my_object = MyClass()
my_object.process(10, 20)  # Processing integers 10 and 20
my_object.process("hello" , "world")  # Processing strings 'hello' and 'world'
my_object.process(True, [1, 2, 3])  # Processing bool True and list [1, 2, 3]
```

Kompositio

Kompositio on olio-ohjelmoinnin tekniikka, jossa luokka koostuu yhdestä tai useammasta muusta luokasta. Tämä mahdollistaa koodin uudelleenkäytön ja modulaarisuuden ilman moniperinnän monimutkaisuutta. Kompositiossa luokka sisältää toisen luokan olion ja delegoi tehtäviä tälle oliolle.

```python
class Moottori:
    def kaynnista(self):
        print("Moottori käynnistyy")

class Auto:
    def __init__(self):
        self.moottori = Moottori()  # Auto sisältää Moottori-olion

    def kaynnista_auto(self):
        self.moottori.kaynnista()  # Delegoi tehtävän Moottori-oliolle

auto = Auto()
auto.kaynnista_auto()  # Moottori käynnistyy
```