Olio-ohjelmointi 1

Miksi käyttää olio-ohjelmointia?

- Olio-ohjelmointi (Object oriented programming tai OOP englanniksi) tarjoaa selkeän rakenteen ohjelmille.
- Olio-ohjelmointi koodi voi olla selkeämpi ja yksikertaisempi kuin vastaavaa "funktio" koodia.
- Olio-ohjelmointi koodi voi olla helpompi ylläpitaa ja helpottaa koodin uudelleenkäyttöä.

Miten käyttää olio-ohjelmointia?

- Tietoa ja funktiota ryhmitellään luokkiin.
- Luokassa oleville tiedolle kutsutaan **attribuutiksi** (attributes)
- Luokassa oleville funktiolle kutsutaan **metodeiksi** (methods)
- Luokka on kuin suunnitelma/malli, josta tehdään oikeita **olioita** (instanssi)
- Yhdestä luokasta voi tehdä useita olioita
- Luokka kertoo, mitä asioita (attribuutteja ja/tai metodia) pitää olla olioissa
- Jokainen olio on *itsenäinen* tai *riippumaton* muista samasta luokasta luoduista olioista
- Luokkaa voi käyttää pohjana uudelle luokalle. Tälle prosesille kutsutaan **periytymiseksi**
- Periytymisessä on mahdollista lisätä ja/tai muokata attribuuttia tai metodia uuteen luokkaan alkuperäistä verrattuna

Yksikertainen esimerkki:

```
# yksinkertaisen luokan luonti:
class Car: # luokka luodaan class-sanalla
    pass

# instanssin eli "olion" luonti luokasta:
audi = Car() # luo olion Car-luokasta, jolla ei ole mitään attribuutteja (omia muuttujia)
skoda = Car()

print(audi) # tulostaa oletuksena olion muistipaikan
print(skoda) # joka oliolla on oma muistipaikkansa
```

Luokan ja olion elementteille (attribuuttia ja metodia) voidaan päästää pisteen avulla: `olio.attribuutti` tai `olio.metodi`.

Oliolle voidaan lisätä manuaalisesti attribuutteja:

```
audi.color = "Red"
audi.speed = 200

skoda.color = "Green"
skoda.height = 1.8

print(audi.color)
print(skoda.color)
```

Olion alustaminen

Tämä ei kuitenkaan ole kätevää, sillä joka luokalle joutuisi kirjoittamaan paljon koodia, mikäli aina manuaalisesti loisi oliolle attribuutteja.

Luokalla on useita erikoismetodia, joilla on tiettyä tarkoitusta. Yksi niistä on *initializer* tai alustaja metodi, joka tulee apuun siihen, kun oliolle halutaan luoda attribuutteja. Alustajaa kutsutaan automaattisesti, kun luokasta luodaan olio. Alustaja -metodin nimi on `__init__`.

Seuraavassa esimerkissä pelkistetty versio alustajan käytöstä:

```
class Car:
    def __init__(self): #tätä kutsutaan automaattisesti kun luokasta luodaan olio
        print("Car-luokasta luotiin olio!")

audi = Car()
skoda = Car()
```

Oliolla ei *tarvitse* olla alustajia, mutta sen käyttö on suositeltavaa, jotta oliota luotaessa määritetyt parametrit saadaan helposti kaikkien olion metodien käyttöön.

`__init__` ei ole konstruktori (constructor)

Pythonissa `__init__`-metodi kutsutaan usein **konstruktorina**, kun se on erikoismetodi, joka suoritetaan **automaattisesti aina, kun luokasta luodaan uusi olio**, mutta sen päätehtävä on oikeasti alustaa olion attribuutit.

Ohjelmoinnissa konstruktorin idea on "rakentaa" olio ja antaa sille alkuarvot. Pythonissa varsinainen olion luonti tapahtuu `__new__`-metodissa ja `__init__` hoitaa olion **alustamisen**, eli sitä voidaan kutsua alustajaksi.

- `__init__` **ei palauta mitään** (paitsi `None`).
- Jos `__init__`-metodia ei määritä, Python käyttää oletusalustaja.

self

Metodit, joita pystytään kutsumaan olion kautta (`olio.metodi`) tarvitsevat ensimmäisenä parametrina viite olioon. Tyyppillisesti käytetään sana ***self***. Nimen ei ole pakko olla *self*, vaan mikä tahansa käy, mutta *self* on yleinen konventio. Tämä parametri ei tarvitse mainita metodin kutsuessa, kun Python lisää sen **automaattisesti** metodin kutsuessa olion kautta. Jos metodi kutsutaan luokan kautta (`Luokka.metodi`), sitten ko. olio pitää lisätä ensimmäisenä parametrina.

Ylläolevassa esimerkissä *self* viittaa audi-oliota luodessa audiin, ja skodaa luotaessa skodaan.

Metodeissa *self*-sanalla voidaan määrittää oliolle omat attribuutit, esim. speed tai color (myöhemmässä esimerkissä)

Attribuuttia

Kätevämpi tapa asettaa oliolle attribuutteja, on joko asettaa niille oletusarvot luokan alustajissa:

```
class Car:
    def __init__(self):
        self.color = "White"
        self.speed = 160


audi = Car()
skoda = Car()
print(audi.color)
print(skoda.color)
```

... tai määrittää attribuutteja oliota luodessa, välittämällä ne parametreina:

```
class Car:
    def __init__(self, color, speed): # Vastaanotetaan parametrit color ja speed.
                                      # Huom! self pitää lisätä ensimmäiseksi,
                                      # siinä itse olio tulee parametrina

        self.color = color # asetetaan olion väriksi parametrina saatu väri
        self.speed = speed


audi = Car("Red", 200) # viedään parametrina väri ja nopeus
skoda = Car("Black", 180)
print(audi.color, audi.speed)
print(skoda.color, skoda.speed)
```

Tai sitten olio voi hyväksyä parametria ja silti laittaa oletusarvoa niille attribuutteille, joille ei ole tullut arvoa parametrina:

```
import random

class Car:
    def __init__(self, color=None, speed=150):
        COLORS = ['White', 'Red', 'Black', 'Grey', 'Brown'] # Apuvakio
        self.color = color if color else random.choice(COLORS)
                    # Asetetaan olion väriksi parametrina saatu väri, jos sitä on tullut.
                    # Asetetaan satunainen väri toisessa tapauksessa
        self.speed = speed
                    # Asetetaan nopeus parametrin mukaan.


audi = Car("Red", 200)
skoda = Car()
merssu = Car('White')
toyota = Car(speed=190)

print(audi.color, audi.speed)
print(skoda.color, skoda.speed)
print(merssu.color, merssu.speed)
print(toyota.color, toyota.speed)
```

Ylläolevassa esimerkissä saatujen parametrien nimet ovat samat, kuin mitkä oliolle asetetaan (`self.color = color` ja `self.speed = speed`). Tämä on yleinen käytäntö.

Parametrien nimet ei kuitenkaan tarvitse olla samaa:

```
class Car:
    def __init__(self, color, speed):
        self.vari = color
        self.nopeus = speed

audi = Car("Red", 200)

print(audi.vari, audi.nopeus)
```

On kuitenkin suositeltavaa käyttää samoja nimeämisiä, jotta koodin luettavuus on helpompaa.

HARJOITUS 1

Luo luokka Car, jonka alustaja saa seuraavat parametrit:

- brand
- color
- speed

Luo Car-luokasta olio, jossa asetat oliolle merkin(brand), värin(color) ja nopeuden(speed).

Lisää alustajaan myös `self.info` -attribuutti. Luo `self.info` :lle toiminnallisuus niin, että kun luokasta luodaan esim. `audi` -niminen olio, ja `print(audi.info)` , tulee tulostettua kuten alla:

```
audi = Car("Audi", "Red", 200)
print(audi.info)

Audi
Color:Red
Speed:200
```

HARJOITUS 2

Luo luokka Employee, jonka alustaja saa parametrit *first_name* ja *last_name*.

Luo alustajaan *self.email*, joka muodostaa sähköpostiosoitteen saaduista parametrista niin, että mikäli luokasta tehdään olio ja printataan *olion_nimi*.email, tulostuu esim. "seppo.suomalainen@taitotalo.fi"

Muista, että sisäänrakennetulla `str` -luokan `lower()` -metodilla saat kaikki merkkijonon kirjaimet pieniksi.

Metodia

Olion metodin luonti (ja metodin kutsuminen):

```
class Car:
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed

                                # self pitää aina laittaa,
    def print_car_color(self):  # mikäli metodi on olion käyttöön tarkoitettu!
        print(self.color)

audi = Car("Red", 200) # viedään parametrina väri ja nopeus
audi.print_car_color()
```

Huoma, että kun oliolle määritetään self:in avulla attribuutteja *alustajassa*, niin attribuutit ovat käytössä jokaisessa olion metodissa! (esim. yllä print_car_color()-metodissa)

Olion attribuutteja voidaan muuttaa metodien sisällä:

```
class Car:
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed

                                # self pitää aina laittaa,
    def change_color_to_green(self):  # mikäli metodi on olion käyttöön tarkoitettu!
        self.color = "Green"

audi = Car("Red", 200) # viedään parametrina väri ja nopeus
audi.change_color_to_green()
print(audi.color)
```

Myös olion metodeille voidaan antaa parametreja:

```
class Car:
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed

                                # self pitää aina laittaa,
    def change_color(self, color):  # mikäli metodi on olion käyttöön tarkoitettu!
        self.color = color

audi = Car("Red", 200) # viedään parametrina väri ja nopeus
audi.change_color("Black")
print(audi.color)
```

Python siis välittää instanssin automaattisesti parametrina oliota *luotaessa* tai olion metodia *kutsuessa*, mutta self-sanaa pitää käytää luokan sisällä, ja aina ensimmäisenä parametrina alustajassa ja metodeissa.

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def get_car_info(self):
        return f"{self.brand} {self.year}"

audi = Car("Audi", 2005)
print(audi.get_car_info())
```

Metodeita voi kutsua myös seuraavasti, mutta instanssi pitää sitten manuaalisesti välittää: (Tämä vain triviatietona, miten Python toimii)

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def get_car_info(self):
        return f"{self.brand} {self.year}"

audi = Car("Audi", 2005)
print(Car.get_car_info(audi))
```

Harjoitus 3

Luo Student -luokka, jossa on:

- Etunimi
- Sukunimi
- Lista suoritteua kurssia
- Olion luodessa tarvitaan vain etu- ja sukunimi: Student(etunimi, sukunimi)
- Metodi: lisätä suoritettu kurssi
- Metodi: printaa suoritettua kurssia

Harjoitus 4

Aikaisemmin (7. päivä perusteista) on tehty sellainen harjoitus/peli kuin 'hirsipuu.py'.
Tehdään luokkaa, joka toimii samalla tavalla ja sopii tälle koodille:

```
game = Hirsipuu('kuningas')

while not game.is_resolved():
    print(game.hidden_word())
    letter = input('Kirjain? ')
    game.resolve_letter(letter)

print('Voitit!')
```

Luokka-attribuutit

Luokka-attribuutti (class attribute) on ominaisuus, joka kuuluu **luokalle itselleen**, ei yksittäisille olioille. Tämä tarkoittaa, että kaikki luokan oliot jakavat saman arvon, ellei sitä erikseen ylikirjoiteta instanssissa.

Luokka-attribuutti eroaa instanssi-attribuutista, sillä, että:

- **Luokka-attribuutti** määritellään suoraan luokan sisällä, **muttei** `__init__`-metodissa ja sitä käytetään `luokka.attribuutti`-muodossa.
- **Instanssi-attribuutti** määritellään yleensä `self`-viittauksella alustajalla (`__init__`) ja sitä käytetään `self.attribuutti`-muodossa.

Esimerkki:

```
class Auto:
    valmistaja = "Toyota"  # Luokka-attribuutti

a1 = Auto()
a2 = Auto()

print(a1.valmistaja)  # Toyota
print(a2.valmistaja)  # Toyota

Auto.valmistaja = "Honda"  # Muutetaan luokan attribuuttia
print(a1.valmistaja)  # Honda
print(a2.valmistaja)  # Honda
```

Huomaa, että kun muutetaan `Auto.valmistaja`, **kaikki oliot näkevät muutoksen**, koska ne viittaavat samaan luokka-attribuuttiin.

Ylikirjoittaa/peittää luokka-attribuuttia

Instansissa voidaan ylikirjoittaa/peittää luokka-attributin arvon omalla arvolla. Eli sillä instansilla olisi oma arvo tälle attribuutille, mutta luokan attribuutin arvo on edelleen sama. Jos esimerkiksessa asetetaan arvon `self.valmistaja` olion sisällä, se luo **instanssi-attribuutin**, joka peittää luokan attribuutin:

```
a1.valmistaja = "Mazda"
print(a1.valmistaja)  # Mazda (instanssi-attribuutti)
print(a2.valmistaja)  # Honda (luokka-attribuutti)
```

Käyttötarkoitukset

Luokka-attribuuttia luodaan eri syistä. Esim:

- Yhteiset arvot kaikille olioille (esim. laskurit, oletusarvot).
- Staattiset tiedot, kuten `PI = 3.14159`.

Esimerkki laskurista:

```
class Auto:
    count = 0
    def __init__(self):
        Auto.count += 1

Auto.count  # 0

auto1 = Auto.count()
Auto.count  # 1
auto1.count # 1

auto2 = Auto.count()
Auto.count  # 2
auto1.count # 2
auto2.count # 2
```

On mahdollista päästää luokka-attribuuttiin olion kautta:

```
class Auto:
    count = 0
    def __init__(self):
        Auto.count += 1

    def get_count(self):
        return self.count

Auto.count          # 0

auto1 = Auto.count()
Auto.count          # 1
auto1.get_count()   # 1

auto2 = Auto.count()
Auto.count          # 2
auto1.get_count()   # 2
auto2.get_count()   # 2
```

Kuitenkin joskus asiat voivat olla sekäviä:

```
class Auto:
    count = 0
    def __init__(self):
        self.count += 1     # Huoma, että nyt on self. Eli ensimmäinen kerta tämä on samaa
kuin: self.count = Auto.count + 1

    def get_count(self):
        return self.count

Auto.count          # 0

auto1 = Auto.count()
Auto.count          # 0
auto1.get_count()   # 1

auto2 = Auto.count()
Auto.count          # 0
auto1.get_count()   # 1
auto2.get_count()   # 1
```