Olio-ohjelmointi 5

AoC_2024_11-harjoitus toteutettu oliolla

Katsotaan AoC_2024_11-harjoitusta toteutettu oliolla

Dataclass

https://docs.python.org/3/library/dataclasses.html

dataclass Pythonissa on ominaisuus, joka helpottaa luokkien määrittelyä, kun ne toimivat pääasiassa tietorakenteina (eli sisältävät vain attribuutteja ja vähän logiikkaa). Se tuli käyttöön Python 3.7:ssa. Vaikka dataclass on tarkoitettu pääasiassa tietorakenteiden määrittelyyn, sitä voidaan käyttää myös muissa tilanteissa, joissa halutaan yksinkertaistaa luokkien määrittelyä. Eli voidaan käyttää sitä myös perinteisissä olio-ohjelmointitehtävissä, joissa luokka sisältää sekä dataa että logiikkaa.

Mitä dataclass tekee?

- Luo automaattisesti metodit, kuten:
  - `__init__` (konstruktori)
  - `__repr__` (tekstiesitys)
  - `__eq__` (vertailu)
- Vähentää boilerplate-koodia, kun halutaan luoda luokkia, jotka säilyttävät dataa.

Eli, esimerkiksi, tämä luokka:

```
class Henkilo:
    def __init__(self, nimi: str, ika: int):
        self.nimi = nimi
        self.ika = ika

    def __repr__(self):
        return f"Henkilo(nimi={self.nimi}, ika={self.ika})"
```

voidaan toteuttaa tällä tavalla:

```
from dataclasses import dataclass

@dataclass
class Henkilo:
    nimi: str
    ika: int
```

Dataclass luo automaattisesti `__init__`, `__repr__` ja `__eq__` -metodit.

```
christian = Henkilo('Christian', 30)
print(christian)                    # Henkilo(nimi='Christian', ika=30)

chrisu = Henkilo('Christian', 30)
print(christian == chrisu)          # True

print(christian.ika)                # 30
```

Käyttännössä määritellään dataclass käyttämällä `@dataclass`-koristetta (decorator) luokan määrittelyn yläpuolella. Luokan attribuutit määritellään tyypitysten (type hints / type annotations) avulla.

Oletusarvot attribuuteille

Jos attribuuteille halutaan määritellä oletusarvoja, ne voidaan asettaa suoraan luokan sisällä. Attribuutit jossa on määritelty oletusarvoja, tulee määritellä attribuuttien listan lopussa.

```
from dataclasses import dataclass

@dataclass
class Henkilo:
    nimi: str
    ika: int
    kaupunki: str = 'Helsinki'          # oletusarvo

print(Henkilo('Anna', 25))              # Henkilo(nimi='Anna', ika=25, kaupunki='Helsinki')
print(Henkilo('Matti', 40, 'Tampere'))  # Henkilo(nimi='Matti', ika=40, kaupunki='Tampere')
```

Oletusarvot voivat olla myös monimutkaisempia, kuten listoja tai sanakirjoja, mutta tällöin tulee käyttää `field`-funktiota `dataclasses`-moduulista, että vältetään yhteiset mutoituvat oletusarvot (kaikki oliot jakavat saman listan esim.).

```
from dataclasses import dataclass, field

@dataclass
class Henkilo:
    nimi: str
    ika: int
    harrastukset: list = field(default_factory=list)  # oletusarvo tyhjä lista

anna = Henkilo('Anna', 25)
anna.harrastukset.append('jalkapallo')
print(anna)                 # Henkilo(nimi='Anna', ika=25, harrastukset=['jalkapallo'])

matti = Henkilo('Matti', 30)
print(matti)                # Henkilo(nimi='Matti', ika=30, harrastukset=[])
```

On myös mahdollista määritellä dataclassiin attribuutteja, joita pitää alustaa ulkoisella funktiolla käyttäen `field`-funktion `default_factory`-parametria.

```
from dataclasses import dataclass, field
import random
def satunnainen_ika():
    return random.randint(18, 65)

@dataclass
class Henkilo:
    nimi: str
    ika: int = field(default_factory=satunnainen_ika)

pekka = Henkilo('Pekka')
print(pekka)                # Henkilo(nimi='Pekka', ika=<satunnainen arvo>)
```

field-parametria

`field`-funktiolla voidaan määritellä erilaisia parametreja, jotka vaikuttavat dataclassin attribuuttien käyttäytymiseen:

- `default`: Määrittelee oletusarvon attribuutille.
- `default_factory`: Määrittelee funktion, joka palauttaa oletusarvon attribuutille.
- `init`: Jos asetettu `False`, attribuutti ei sisällytetä `__init__`-metodiin. (Oletusarvo on `True`.)
- `repr`: Jos asetettu `False`, attribuutti ei sisälly `__repr__`-metodiin. (Oletusarvo on `True`.)
- `compare`: Jos asetettu `False`, attribuuttia ei käytetä vertailussa (`__eq__`). (Oletusarvo on `True`.)
- `hash`: Jos asetettu `False`, attribuutti ei vaikuta hajautusarvoon (`__hash__`). (Oletusarvo on `None`, mikä tarkoittaa, että se perii luokan asetuksen.)
- `metadata`: Sanakirja, johon voidaan tallentaa lisätietoja attribuutista. Tämä ei vaikuta dataclassin toimintaan, mutta voi olla hyödyllinen lisätieto kehittäjille.

```python
from dataclasses import dataclass, field

@dataclass
class Henkilo:
    nimi: str
    ika: int = field(default=30, repr=False)    # ei näy __repr__:ssä
    salasana: str = field(init=False)           # ei sisällytä __init__:iin

    def __post_init__(self):
        self.salasana = 'salainen'              # alustetaan __post_init__:ssä

anna = Henkilo('Anna')
print(anna)                                     # Henkilo(nimi='Anna')
print(anna.salasana)                            # salainen
```

`__post_init__` on dataclass-metodi, jota käytetään alustamaan attribuutteja, jotka eivät ole osa `__init__`-metodia. Oletus dataclass `__init__`-metodi kutsu `__post_init__` sen lopussa.

`@dataclass`-koristelijan parametrit

`@dataclass`-koristelijalle voidaan antaa useita **parametreja**, jotka muuttavat sen toimintaa:

- `init` (oletus: True)
  - Luo automaattisesti `__init__`-metodin.
  - Jos asetetaan `False`, kehittäjä täytyy kirjoittaa oma `__init__`.

```python
@dataclass(init=False)
class Esimerkki:
    x: int

Esimerkki(1)    # Virhe! "TypeError: Esimerkki() takes no arguments"
```

- `repr` (oletus: True)
  - Luo automaattisesti `__repr__`-metodin.
  - Jos `False`, ei luoda tekstiesitystä.

- `eq` (oletus: True)
  - Luo `__eq__`-metodin (olioiden vertailu).
  - Jos `False`, vertailu ei ole automaattinen.
- `order` (oletus: False)
  - Luo vertailuoperaattorit (`<` , `>` , `<=` , `>=`).
  - Vaatii, että `eq=True`.
  - Attribuutit vertaillaan määrittelyjärjestyksessä ja tupleina.

```
@dataclass(order=True)
class Piste:
    x: int
    y: int

Piste(2,3) > Piste(1,4) # True: (2, 3) > (1, 4)
```

Jos on tarve vertailla tiettyllä tavalla, voi käyttää `field(compare=False)` tai määritellä oma vertailuavain. Esim:

```
from dataclasses import dataclass, field

@dataclass(order=True)
class Henkilo:
    nimi: str = field(compare=False)  # Ei vaikuta vertailuun
    ika: int
```

```
from dataclasses import dataclass, field

@dataclass(order=True)
class Henkilo:
    sort_index: int = field(init=False, repr=False)
    nimi: str
    ika: int

    def __post_init__(self):
        sort_index = len(self.nimi)
```

- `frozen` (oletus: False)
  - Tekee olion **immutettomaksi** (kenttiä ei voi muuttaa).
  - Hyödyllinen esim. sanakirjan avaimina.

```
@dataclass(frozen=True)
class Piste:
    x: int
    y: int

p = Piste(2,3)
p.x = 5     # Virhe! "dataclasses.FrozenInstanceError: cannot assign to field 'x'"
```

- `unsafe_hash` (oletus: False)
  - Luo `__hash__` vaikka `eq=True` ja `frozen=False`.
  - Käytetään varoen, koska voi rikkoa hashin konsistenssin.
- `slots` (Python 3.10+)
  - Luo `__slots__` → vähentää muistinkäyttöä ja nopeuttaa attribuuttien käsittelyä.

```
@dataclass(slots=True)
class Piste:
    x: int
    y: int
```

Metodien lisääminen dataclassiin

Dataclass-luokkiin voidaan lisätä omia metodeja aivan kuten tavallisiin luokkiin.

```
from dataclasses import dataclass

@dataclass
class Henkilo:
    nimi: str
    ika: int

    def tervehdys(self):
        return f"Hei, nimeni on {self.nimi} ja olen {self.ika} vuotta vanha."

anna = Henkilo('Anna', 25)
print(anna.tervehdys())  # Hei, nimeni on Anna ja olen 25 vuotta vanha.
```

Dataclass-luokan pereytyminen

Dataclass-luokat voivat periä toisista dataclass-luokista aivan kuten tavalliset luokat. Perityt attribuutit ja metodit toimivat odotetusti.

```
from dataclasses import dataclass

@dataclass
class Elain:
    nimi: str
    ika: int

@dataclass
class Koira(Elain):
    rotu: str

koira = Koira('Rex', 5, 'Labradori')
print(koira)  # Koira(nimi='Rex', ika=5, rotu='Labradori')
```

Huomioitavaa on, että lapsiluokan attribuutit tulevat emoluokan attribuuttien jälkeen, mikä vaikuttaa `__init__`-metodin parametreihin. Jos emoluokalla on attribuutteja, joilla on oletusarvot, lapsiluokan attribuuteille pitää määrätä myös oletusarvoa.

```
from dataclasses import dataclass

@dataclass
class Elain:
    nimi: str
    ika: int = 0  # oletusarvo

@dataclass
class Koira(Elain):
    rotu: str

# TypeError: non-default argument 'rotu' follows default argument 'ika'
```

Harjoitus

Muuta aikaisemmin tehty AoC_2024_11-harjoitus käyttäämään dataclassia

Generaattorit

Generator on erityinen tapa luoda **iteroitavia olioita**, jotka tuottavat arvoja **laiskasti (lazy evaluation)** eli yksi kerrallaan, sen sijaan että kaikki arvot laskettaisiin ja tallennettaisiin muistiin kerralla.

Keskeiset piirteet:

- Generaattori **ei palauta koko listaa**, vaan tuottaa arvot sitä mukaa kun niitä tarvitaan.
- Toteutetaan yleensä `yield`-avainsanalla funktiossa.
- **Muistitehokas**: ei varaa koko sarjaa muistiin.
- **Iteroitava**: toimii `for`-silmukassa kuten lista, mutta arvot lasketaan dynaamisesti.

Esimerkki:

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i  # palauttaa arvon ja jatkaa seuraavasta kohdasta
        i += 1

for num in count_up_to(5):
    print(num)
```

Generaattorin edut:

- **Muistitehokkuus**: Hyvä suurille datamäärille.
- **Laiska laskenta**: Arvot lasketaan vasta kun niitä tarvitaan.
- **Helppo luoda**: `yield` tekee koodista selkeää.

Generaattorit verrattuna listoihin:

- Lista: kaikki arvot tallennetaan muistiin.
- Generaattori: arvot tuotetaan yksi kerrallaan.

Toinen tapa luoda generaattori on käyttää **generaattorilauseketta (generator expression)**, joka muistuttaa listan comprehensiota (list comprehension):

```
squares = (x * x for x in range(10))  # generaattorilauseke

for square in squares:
    print(square)
```

Enum

Enum (lyhenne sanasta "enumeration") on luokka, joka tarjoaa tavan määritellä joukko nimettyjä vakioita. Enum-luokat ovat hyödyllisiä, kun halutaan ryhmitellä liittyviä vakioita yhteen paikkaan ja tehdä koodista luettavampaa ja helpommin ylläpidettävää. Enum-arvot ovat muuttumattomia, mikä tarkoittaa, että niiden arvoja ei voi muuttaa luomisen jälkeen.

Enum-luokat määritellään käyttämällä `enum`-moduulia.

```
from enum import Enum

class Colour(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
    YELLOW = 4
    BLACK = 5
    WHITE = 6

    def is_primary(self):
        return self in (Colour.RED, Colour.GREEN, Colour.BLUE)

print(Colour.RED)        # Colour.RED
print(Colour.RED.name)   # RED
print(Colour.RED.value)  # 1
```

Enum-instanssia voidaan verrata toisiaan koska jokaisella on oma arvo:

```
print(Colour.RED == Colour.GREEN) # False
```

Enum-luokka voidaan käydä läpi saamaan mahdolliset arvot/instanssit:

```
for colour in Colour:
    print(colour)
```

Enum-instassia voidaan saada myös attribuuttien nimellä tai arvolla:

```
colour = Colour['RED']          # Index-muoto saamaan instanssin nimen perustella
print(colour)                   # Colour.RED

colour = Colour(1)              # Funktio/kutsuminen-muoto saamaan instanssin arvon perustella
print(colour)                   # Colour.RED
```

Enum on Python luokka kuitenkin ja voi sisältää metodeja:

```
print(colour.is_primary())          # True (RED)
print(colour.YELLOW.is_primary())   # False
```

On myös mahdollista määrittää arvoja automaattisesti `auto` -funktiolla:

```
from enum import Enum, auto

class Colour(Enum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()
    YELLOW = auto()
    BLACK = auto()
    WHITE = auto()
```

Voidaan myös määrittää Enumia suoraan Enum-luokasta:

```
Colour = Enum('Colour', [('RED', 1), ('GREEN', 2), ('BLUE', 3)])
```

`Enum` -luokan lisäksi `enum` -moduuli tarjoa eri luokkaa kuten `IntEnum`, `StrEnum`, `IntFlag`...

https://docs.python.org/3/library/enum.html

Lisää aiheesta: https://realpython.com/python-enum/