Olio-ohjelmointi 2

Edellisen päivän harjoitus 2: Lisätään useita kurssia

Muokataan `Student.add_course()`, että se hyväksyy jotain tällaista:

```
Student.add_course('Python', 'Linux')
```

Staattinen metodi

Pythonissa staattiset metodit ovat luokan sisällä määriteltyjä funktioita, jotka eivät riipu luokan instanssista. Niitä käytetään, kun ei tarvita pääsyä luokan tai sen instanssien tilaan tai ei haluta muuttaa sitä. Staattinen metodi määritellään käyttämällä `@staticmethod`-decorator.

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

    @staticmethod
    def hp_to_watts(hp: int|float):
        return hp * 745.699872

# Kutsutaan staattista metodia
Car.hp_to_watts(200)
```

Esimerkissä `hp_to_watts` on staattinen metodi luokassa `OmaLuokka`. Voidaan kutsua sen suoraan luokasta ilman sitä, että tarvitaan luoda luokan instanssia. Huoma, että tällaisia metodia **ei ole** mahdollista kutsua olion kautta (`audi.hp_to_watts` antaa virhe)

Staattiset metodit ovat hyödyllisiä, kun halutaan määrittellä funktioita, joilla on looginen yhteys luokkaan, mutta jotka eivät tarvitse pääsyä sen attribuutteihin tai instanssimetodeihiin.

> **Huom: decorator**
>
> Pythonissa koristelija (decorator) on erikoisrakenne, joka mahdollistaa funktioiden tai metodien toiminnallisuuden laajentamisen ilman, että niiden alkuperäistä koodia muutetaan. Käytännössä se on funktio, joka ottaa toisen funktion (tai metodin) argumenttina ja palauttaa uuden funktion, usein lisäten siihen jotain ylimääräistä logiikkaa.
>
> Sitä käytetään lisäämällä `@`-merkki ja ko. koristelijan nimi juuri ennen funktiota tai metodia.

Luokkametodi

Luokkametodi on funktio, joka on sidottu luokkaan eikä sen instanssiin. Tämä tarkoittaa, että luokkametodi voi käyttää ja muokata luokan tilaa, mutta ei yksittäisten instanssien tilaa. Luokkametodi ottaa ensimmäisenä parametrina `cls`, joka viittaa itse luokkaan. Pythonissa `@classmethod` on koristelija (decorator), jota käytetään määrittelemään luokkametodi.

```
class Example:
    level = 0

    def __init__(self, value):
        self.value = value

    @classmethod
    def update_level(cls, new_value):
        cls.level = new_value


Example.update_level(10)

print(Example.level)  # 10
```

Voidaan kutsua luokkametodia suoraan luokasta ilman, että tarvitse luoda instanssia. On kuitenkin mahdollista kutsua niitä myös oliosta.

Luokkametodit ovat hyödyllisiä, kun halutaan tehdä muutoksia, jotka vaikuttavat koko luokkaan eikä vain yksittäisiin instansseihin. Luokkametodia voivat olla hyodyllisiä myös pereytymisessä.

Getters and Setters

Getters ja setters ovat menetelmiä, joita käytetään ohjelmoinnissa olion attribuuttien (eli ominaisuuksien) lukemiseen ja asettamiseen. Ne tarjoavat tavan hallita pääsyä luokan attribuutteihin ja voivat auttaa suojaamaan ja kapseloimaan dataa.

Getter-menetelmät palauttavat luokan attribuutin arvon. Ne tarjoavat tavan lukea attribuutin arvoa ilman, että pääsee suoraan käsiksi attribuuttiin.

Setter-menetelmät asettavat luokan attribuutin arvon. Ne tarjoavat tavan muuttaa attribuutin arvoa ja voivat sisältää logiikkaa, joka tarkistaa tai muokkaa arvoa ennen sen asettamista.

```java
// Esimerkki Java - kiellellä

public class Henkilo {
    private String nimi;

    // Konstruktori
    public Henkilo(String nimi) {
        this.nimi = nimi;
    }

    // Getter for 'nimi'
    public String getNimi() {
        return nimi;
    }

    // Setter for 'nimi'
    public void setNimi(String uusiNimi) {
        if (uusiNimi != null && !uusiNimi.isEmpty()) {
            this.nimi = uusiNimi;
        } else {
            throw new IllegalArgumentException("Nimen täytyy olla ei-tyhjä merkkijono");
        }
    }

    public static void main(String[] args) {
        Henkilo henkilo = new Henkilo("Matti");
        System.out.println(henkilo.getNimi());  // Tulostaa: Matti
        henkilo.setNimi("Maija");
        System.out.println(henkilo.getNimi());  // Tulostaa: Maija
    }
}
```

Getters ja setters auttavat hallitsemaan ja suojaamaan luokan attribuutteja, mikä tekee koodista luotettavampaa ja helpommin ylläpidettävää.

Huoma, että ne ovat metodia, joita pitää kutsua "fuktiona". Alkuperäinen attribuutti ei voi käyttää ulkoa (ei edes lukea), kun se on yksityinen.

Pythonissa ei ole yksityisiä attribuuttia tai metodia, mutta on tapana lisätä `_` (alaviiva) attribuutin tai metodin nimen eteen, jos se on tarkoitettu käyttää vain sisäisesti. Voidaan sitten luoda getter ja setter normaalilla tavalla.

Javan vastaava esimerkki Pythonissa olisi:

```
class Henkilo:
    def __init__(self, nimi):
        self._nimi = nimi

    # Getter for 'nimi'
    def get_nimi(self):
        return self._nimi

    # Setter for 'nimi'
    def set_nimi(self, uusi_nimi):
        if isinstance(uusi_nimi, str):
            self._nimi = uusi_nimi
        else:
            raise ValueError("Nimen täytyy olla merkkijono")

henkilo = Henkilo("Matti")
print(henkilo.get_nimi())  # Matti
henkilo.set_nimi("Maija")
print(henkilo.get_nimi())  # Maija
```

Property

Pythonissa `@property` on koristelija, joka mahdollistaa luokan attribuuttien kapseloinnin ja hallinnan. Se muuttaa metodin ominaisuudeksi, jota voidaan käyttää kuten attribuuttia, mutta joka tarjoaa lisälogiikkaa, kuten validointia tai laskentaa.

`@property`-koristelijaa käytetään määrittelemään getter-metodi, joka palauttaa attribuutin arvon. Voidaan myös määritellä setter- ja deleter-metodit, jotka asettavat ja poistavat attribuutin arvon: `@property_name.setter` ja `@property_name.deleter`. Eli setter ja deleter -decoratorit riippuvat ensisijaisesta property -nimesta (se metodi, jossa on `@property`)

```
class Henkilö:
    def __init__(self, nimi):
        self._nimi = nimi

    # Getter for 'nimi'
    @property
    def nimi(self):
        return self._nimi

    # Setter for 'nimi'
    @nimi.setter
    def nimi(self, uusi_nimi):
        if isinstance(uusi_nimi, str):
            if uusi_nimi:
                self._nimi = uusi_nimi
            else:
                raise ValueError('Nimi ei voi olla tyhjä')
        else:
            raise TypeError("Nimi täytyy olla merkkijono")

henkilö = Henkilö("Matti")
print(henkilö.nimi)  # Matti
henkilö.nimi = "Maija"
print(henkilö.nimi)  # Maija
```

Pythonissa on ensisijaisesti parempi käyttää propertya kuin getters ja setters.

Selkeä etu on, että voidaan käyttää property-attribuutin tavallisella tavalla, eli `olio.attribuutti`, sekä lukemiseen, että sijoittamiseen.

Toinen etu on, että voidaan alussa käyttää jotain tietoa tavallisena attribuuttina ja tarvittaessa myöhemmin voidaan muuttaa sen propertyksi ilman sitä, että asiakas koodi pitää muutaa.

`@property` -avulla voidaan luoda 'vain-luku' attribuuttia:

```
class Henkilö:
    def __init__(self, nimi):
        self._nimi = nimi

    # Getter for 'nimi'
    @property
    def nimi(self):
        return self._nimi

henkilö = Henkilö('Matti')
print(henkilö.nimi)     # Matti
henkilö.nimi = 'Maija'  # Virhe! : AttributeError: property 'nimi' of 'Henkilö' object has no
setter
```

Olisi myös mahdollista tehdä 'vain-kirjoitus' attribuuttia (getter palauttaa virhe):

```
import hashlib
import os

class User:
    def __init__(self, name, password):
        self.name = name
        self.password = password

    @property
    def password(self):
        raise AttributeError("Password is write-only")

    @password.setter
    def password(self, plaintext):
        salt = os.urandom(32)
        self._hashed_password = hashlib.pbkdf2_hmac("sha256", plaintext.encode("utf-8"), salt,
100_000)
```

Ja myös on mahdollista luoda ns. "computed attributes". Eli sellaiset attribuutit, jotka ovat laskettu lennosta

```python
class Square:
    def __init__(self, side):
        self.side = side

    @property
    def area(self):
        return self.side**2

    @property
    def perimeter(self):
        return self.side*4

box = Square(5)
print(box.area)         # 25
print(box.perimeter)    # 20
```

Harjoitus 1

Muoka Guessword -luokka sillä tavalla, että `hidden_word` on property

Dunder -metodit

https://docs.python.org/3/reference/datamodel.html#specialnames

"Dunder-metodit" (lyhenne sanoista "double underscore methods") ovat Pythonin erityisiä metodeja, jotka alkavat ja päättyvät kahdella alaviivalla (`__`). Näitä metodeja kutsutaan myös "magic methods" tai "special methods". Ne tarjoavat erityisiä toimintoja ja mahdollistavat luokkien käyttäytymisen mukauttamisen.

Yleisiä dunder-metodeja

1. `__init__` : Alustaja, joka suoritetaan aina, kun uusi instanssi luodaan.

```python
def __init__(self, arvo):
    self.arvo = arvo
```

2. `__str__` : Määrittää, mitä tulostetaan, kun objektia yritetään tulostaa `print` -funktiolla.

```
def __str__(self):
    return f"Objekti, jonka arvo on {self.arvo}"
```

3. `__repr__` : Määrittää, mitä tulostetaan, kun objektia yritetään tulostaa komentorivillä tai `repr` -funktiolla.

```
def __repr__(self):
    return f"Objekti({self.arvo})"
```

4. `__len__` : Määrittää, mitä palautetaan, kun `len` -funktiota käytetään objektin kanssa.

```
def __len__(self):
    return len(self.arvo)
```

5. `__getitem__` : Mahdollistaa indeksin käytön objektin kanssa.

```
def __getitem__(self, indeksi):
    return self.arvo[indeksi]
```

6. `__setitem__` : Mahdollistaa arvon asettamisen indeksin avulla.

```
def __setitem__(self, indeksi, arvo):
    self.arvo[indeksi] = arvo
```

```
class OmaLuokka:
    def __init__(self, arvo):
        self.arvo = arvo

    def __str__(self):
        return f"OmaLuokka, jonka arvo on {self.arvo}"

    def __repr__(self):
        return f"OmaLuokka({self.arvo})"

    def __len__(self):
        return len(self.arvo)

    def __getitem__(self, indeksi):
        return self.arvo[indeksi]

    def __setitem__(self, indeksi, arvo):
        self.arvo[indeksi] = arvo


obj = OmaLuokka([1, 2, 3])
print(obj)          # "OmaLuokka, jonka arvo on [1, 2, 3]"
print(repr(obj))    # "OmaLuokka([1, 2, 3])"
print(len(obj))     # 3
print(obj[1])       # 2
obj[1] = 10
print(obj[1])       # 10
```

Dunder-metodit tekevät luokista joustavampia ja mahdollistavat niiden käytön Pythonin sisäänrakennettujen toimintojen kanssa. Ne auttavat myös tekemään koodista intuitiivisempaa ja helpommin ymmärrettävää.

On myös olemassa useita vertailu operaattoria:

1. **`__eq__`**: `==` (yhtäsuuruus)

   ```
   def __eq__(self, toinen):
       return self.arvo == toinen.arvo
   ```

2. `__ne__` : `!=` (erisuuruus)

```
def __ne__(self, toinen):
    return self.arvo != toinen.arvo
```

3. `__lt__` : `<` (pienempi kuin)

```
def __lt__(self, toinen):
    return self.arvo < toinen.arvo
```

4. `__le__` : `<=` (pienempi tai yhtäsuuri kuin)

```
def __le__(self, toinen):
    return self.arvo <= toinen.arvo
```

5. `__gt__` : `>` (suurempi kuin)

```
def __gt__(self, toinen):
    return self.arvo > toinen.arvo
```

6. `__ge__` : `>=` (suurempi tai yhtäsuuri kuin)

```
def __ge__(self, toinen):
    return self.arvo >= toinen.arvo
```

Harjoitus 2

Tee `Henkilö` -luokka, joka hyväksi kaksi parametria: nimi ja ikä. Implementoi `__eq__` -metodi, joka palauttaa True jos olioiden attribuutit ovat samoja.

Käyttö esimerkki:

```
h1 = Henkilö('Matti', 23)
h2 = Henkilö('Matti', 23)
h3 = Henkilö('Matti', 24)

h1 == h2    # True
h1 == h3    # False
```

Yleisimmät matemaattiset dunder-metodit ja niiden vastaavat operaattorit:

1. **`__add__`**: `+` (yhteenlasku)

```
def __add__(self, toinen):
    return self.arvo + toinen.arvo
```

2. **`__sub__`**: `-` (vähennyslasku)

```
def __sub__(self, toinen):
    return self.arvo - toinen.arvo
```

3. **`__mul__`**: `*` (kertolasku)

```
def __mul__(self, toinen):
    return self.arvo * toinen.arvo
```

4. **`__truediv__`**: `/` (jakolasku)

```
def __truediv__(self, toinen):
    return self.arvo / toinen.arvo
```

5. **`__floordiv__`**: `//` (lattiajakolasku)

```
def __floordiv__(self, toinen):
    return self.arvo // toinen.arvo
```

6. **`__mod__`**: `%` (jakojäännös)

```
def __mod__(self, toinen):
    return self.arvo % toinen.arvo
```

7. `__pow__` : `**` (potenssi)

```
def __pow__(self, toinen):
    return self.arvo ** toinen.arvo
```

8. `__neg__` : `-` (negatiivinen)

```
def __neg__(self):
    return -self.arvo
```

Esimerkki luokasta, joka käyttää joitakin näistä matemaattisista dunder-metodeista:

```
class Numero:
    def __init__(self, arvo):
        self.arvo = arvo

    def __add__(self, toinen):
        return Numero(self.arvo + toinen.arvo)

    def __sub__(self, toinen):
        return Numero(self.arvo - toinen.arvo)

    def __mul__(self, toinen):
        return Numero(self.arvo * toinen.arvo)

    def __truediv__(self, toinen):
        return Numero(self.arvo / toinen.arvo)

    def __str__(self):
        return str(self.arvo)

num1 = Numero(10)
num2 = Numero(5)

print(num1 + num2)  # 15
print(num1 - num2)  # 5
print(num1 * num2)  # 50
print(num1 / num2)  # 2.0
```

On olemassa myös "peilittu" -metodia, joita käytetään kun "normaali" -metodi palauttaa `NotImplemented`:

- `__radd__(self, other)`
- `__rsub__(self, other)`
- `__rmul__(self, other)`
- `__rmatmul__(self, other)`
- `__rtruediv__(self, other)`
- `__rfloordiv__(self, other)`
- `__rmod__(self, other)`
- `__rdivmod__(self, other)`

- `__rpow__(self, other[, modulo])`

Eli kun Python löytää esim. `obj1 + obj2`, ensin suorittaa `obj1.__add__(obj2)`. Jos tämä palauttaa `NotImplemented`, sitten suorittaa `obj2.__radd__(obj1)`.

```
a = 2       # int
b = 3.0     # float

a.__add__(b)    # NotImplemented
b.__radd__(a)   # 5.0
```

Huomataan myös, että dunder metodit, joissa on kyseessä kaksi oliota, voivat implementoida kyseessä oleva toiminallisuus eri olion kanssa. Esim. Numero + int

```
class Numero:
    def __init__(self, arvo) -> None:
        self.arvo = arvo

    def __repr__(self) -> str:
        return f'Numero({self.arvo})'

    def __add__(self, toinen):
        if isinstance(toinen, Numero):
            return Numero(self.arvo + toinen.arvo)
        elif isinstance(toinen, int):
            return Numero(self.arvo + toinen)
        else:
            return NotImplemented

    def __radd__(self, toinen):
        return self.__add__(toinen)

a = Numero(5)
b = Numero(7)
a + b   # Numero(12)
a + 1   # Numero(6)
1 + a   # Numero(6)
```

Kaikki dunder -metodit: https://www.pythonmorsels.com/every-dunder-method/

Harjoitus 3

Toteuta Vector -luokka, joka hyväksyy kaksi lukua alustajassa: `Vector(a, b)`. Toteuta sille ainakin `__eq__`, `__ne__`, `__add__`, `__sub__`, `__neg__` ja `__len__`-metodit.

- `Vector(a, b) == Vector(c, d)` = `a == c and b == d`
- `-Vector(a, b)` = `Vector(-a, -b)`
- `Vector(a, b) + Vector(c, d)` = `Vector(a+c, b+d)`
- `Vector(a, b) - Vector(c, d)` = `Vector(a-c, b-d)`
- `len(Vector(a, b))` = `sqrt(a**2 + b**2)`

Toteuta myös Vectorin skalaaritulo:

- `Vector(a, b) * n` = `Vector(a*n, b*n)`