# Olio-ohjelmointi 4

## 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.

```
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
```

# Iterables ja iterators

Iterable on olio, josta voidaan luoda iteraattoria. Toisin sanoen, iterable on mikä tahansa olio, joka tukee iterointia. Yleisimpiä iterable-olioita ovat listat, tuplet, sanakirjat ja merkkijonot. Teknisesti iterable-olioilla on `__iter__`-metodi, joka palauttaa iteraattorin, ja/tai `__getitem__`-metodi, joka toteuttaa *Sequence*-ominaisuuksia.

Iteraattorit mahdollistavat tietorakenteiden läpikäymisen yksi alkio kerrallaan. Iteraattorit ovat keskeinen osa Pythonin iterointiprotokollaa, joka koostuu kahdesta pääkomponentista: `__iter__` - ja `__next__`-metodeista.

`__iter__` -metodi iterable -oliosta palauttaa jonkun iterator -olio, joka voi olla samaa kuin iterable -olio tai ei. Käydään läpi olion elementit kutsumalla `__next__`-metodi.

```
class CustomClass:
    def __init__(self, data):
        self.data = data
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            result = self.data[self.index]
            self.index += 1
            return result
        else:
            raise StopIteration

my_object = CustomClass([1, 2, 3])
for item in my_object:
    print(item)
```

Voidaan sitä tehdä myös manuaalisesti:

```
my_object = CustomClass([1, 2, 3])
iterator_object = my_object.__iter__()
print(iterator_object.__next__()) # 1
print(iterator_object.__next__()) # 2
print(iterator_object.__next__()) # 3
iterator_object.__next__() # StopIteration
```

Pythonissa on myös sisäänrakennettuja funktioita, kuten `iter()` ja `next()`, jotka helpottavat iteraattorien käyttöä:

```
data = [1, 2, 3]
iterator = iter(data)

print(next(iterator))  # 1
print(next(iterator))  # 2
print(next(iterator))  # 3
print(next(iterator))  # StopIteration
```

Tässä esimerkissä `iter()`-funktio luo iteraattorin listasta, ja `next()`-funktio palauttaa seuraavan alkion iteraattorista.

Iteraattorit ovat hyödyllisiä, kun haluat käsitellä suuria tietomääriä tehokkaasti tai kun haluat luoda omia mukautettuja iteraattoreita.

# Duck typing

Duck typing tarkoittaa ohjelmointiparadigmaa, jossa olion tyyppiä tai luokkaa ei tarkisteta eksplisiittisesti, vaan sen **käyttäytymistä** (eli mitä metodeja ja ominaisuuksia sillä on) käytetään ratkaisevana tekijänä. Ajatus perustuu sanontaan:

> *"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."*

Mitä tämä käytännössä tarkoittaa?

- Pythonissa ei ole pakko määritellä tiukkoja tyyppirajoituksia.
- Jos olio tukee tarvittavia metodeja ja ominaisuuksia, sitä voidaan käyttää, riippumatta sen varsinaisesta luokasta.
- Tämä tekee koodista joustavaa ja helpottaa polymorfismia.

Esimerkki:

```
class Duck:
    def quack(self):
        print("Quack!")

class Person:
    def quack(self):
        print("I can imitate a duck!")

def make_it_quack(obj):
    obj.quack()  # Ei tarkisteta tyyppiä, vain että metodin voi kutsua

duck = Duck()
person = Person()

make_it_quack(duck)    # Quack!
make_it_quack(person)  # I can imitate a duck!
```

Tässä funktio `make_it_quack` ei välitä, onko olio `Duck` vai `Person`, kunhan sillä on `quack()`-metodi.

# Iterable vs Sequence

**Iterable** ja **Sequence** liittyvät toisiinsa, mutta eivät ole sama asia Pythonissa. Tässä selkeä ero:

## Iterable

- **Määritelmä**: Olio, jota voidaan iteroida (käydä läpi) `for`-silmukalla.
- Toteuttaa `__iter__()`-metodin (tai `__getitem__()` vanhemmassa mallissa).
- Ei välttämättä ole järjestetty eikä tue indeksointia.
- Esimerkkejä:
  - `list`, `tuple`, `str` (sekvenssejä, mutta myös iterableja)
  - `set`, `dict` (iterableja, mutta **ei** sekvenssejä)
  - Generaattorit (`generator`), tiedostokahvat

## Sequence

- **Määritelmä**: Erityinen iterable, joka on **järjestetty** ja tukee **indeksointia** ja **viipalointia**.
- Toteuttaa `__getitem__()` ja `__len__()`.
- Esimerkkejä:
  - `list`, `tuple`, `str`, `range`

Keskeinen ero:

- **Kaikki sekvenssit ovat iterableja**, mutta **kaikki iterablet eivät ole sekvenssejä**.
- Esim. `set` ja `dict` ovat iterableja, mutta eivät sekvenssejä, koska niillä ei ole kiinteää järjestystä eikä indeksointia.

Esimerkki:

```
# Iterable mutta ei sequence
my_set = {1, 2, 3}
for item in my_set:
    print(item)  # toimii
# my_set[0]  # Virhe! Ei indeksointia

# Sequence
my_list = [10, 20, 30]
print(my_list[0])  # toimii
```

# Olion luoja-metodi (new)

Pythonissa `__new__`-metodi on erityinen metodi, joka vastaa uuden olion luomisesta. Se on staattinen metodi, joka kutsutaan ennen `__init__`-metodia, ja sen avulla voidaan hallita olion luontiprosessia. `__new__`-metodia käytetään yleensä silloin, kun halutaan vaikuttaa olion luontiin, esimerkiksi singleton-mallin toteuttamisessa tai kun peritään muuttumattomia luokkia, kuten `tuple` tai `str`.

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        print("Creating instance")
        instance = super().__new__(cls)
        return instance

    def __init__(self, value):
        print("Initializing instance")
        self.value = value

obj = MyClass(10)
print(obj.value)
```

Tyyppillisesti uuden olion luodaan kutsumalla `super().__new__(cls)`. `__new__`-metodi pitää palauttaa luotun instanssin.
Instanssi voidaan alustaa tavallisella tavalla.

```
class PositiveInt(int):
    def __new__(cls, value):
        if value > 0:
            raise ValueError('Positive supports only positive integers.')
        return super().__new__(cls, value)

a = PositiveInt(5)
print(a)

b = PositiveInt(-1)    # Virhe!
```

# Singleton

Singleton on suunnittelumalli, joka varmistaa, että luokasta luodaan vain yksi instanssi koko ohjelman elinkaaren aikana. Tämä on hyödyllistä tilanteissa, joissa halutaan rajoittaa olion luonti yhteen instanssiin, kuten esimerkiksi konfiguraatioasetuksissa, lokituksessa tai tietokantayhteyksissä.

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

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

obj1 = Singleton(10)
obj2 = Singleton(20)

print(obj1.value)   # 20
print(obj2.value)   # 20
print(obj1 is obj2) # True
```

`Singleton`-luokka varmistaa, että vain yksi instanssi on mahdollista luoda. Jos instanssia ei ole vielä luotu, `__new__`-metodi luo sen ja tallentaa `_instance`-luokka-attribuuttiin. Seuraavilla kerroilla `__new__` palauttaa olemassa olevan instanssin.

# Rekursio

Rekursio ohjelmoinnissa tarkoittaa menetelmää, jossa **funktio kutsuu itseään** suorittaakseen tehtävän. Tämä jatkuu, kunnes saavutetaan **pohjaehto (base case)**, joka lopettaa rekursion. Ilman pohjaehtoa rekursio jatkuu loputtomasti ja aiheuttaa virheen (esim. `RecursionError` Pythonissa).

Perusidea:

- **Pohjaehto**: määrittää, milloin rekursio lopetetaan.
- **Rekursiivinen askel**: funktio kutsuu itseään pienemmällä tai yksinkertaisemmalla syötteellä.

Esimerkki:

```
def factorial(n):
    if n == 0:  # pohjaehto
        return 1
    else:
        return n * factorial(n - 1)  # rekursiivinen kutsu

print(factorial(5))  # Tulostaa 120
```

Tässä:

- `factorial(5)` kutsuu `factorial(4)`, joka kutsuu `factorial(3)` jne.
- Kun `n == 0`, rekursio pysähtyy.

Rekursiota käytetään:

- Läpikäynti puu- ja graafirakenteissa (esim. hakualgoritmit).
- Laskennalliset ongelmat, kuten faktoriaali, Fibonacci-luku.
- Tiedostorakenteiden läpikäynti.

**Harjoitus**

Toteutaa AoC 2024_11 käyttämällä rekursiota.

## Memoization

Memoization on optimointitekniikka, jossa tallennetaan aiemmin lasketut funktiokutsujen tulokset, jotta niitä ei tarvitse laskea uudelleen samoilla syötteillä. Tämä on erityisen hyödyllistä rekursiivisissa funktioissa, joissa samat laskelmat toistuvat usein.

```
from functools import cache
import time

def test_performance(func) -> None:
    print(f'Testing performance of {func.__name__}:')
    for n in range(1,41):
        print(f'{n = }: ', end='')
        start = time.time_ns()
        print(func(n), end='')
        end = time.time_ns()
        print(f' {(end-start)/1e9:.3f}s')
    print()

def naive_fibonacci(n: int) -> int:
    if n <= 1:
        return n
    else:
        return naive_fibonacci(n - 1) + naive_fibonacci(n - 2)

test_performance(naive_fibonacci)

@cache
def naive_fibonacci_cached(n: int) -> int:
    if n <= 1:
        return n
    else:
        return naive_fibonacci_cached(n - 1) + naive_fibonacci_cached(n - 2)

test_performance(naive_fibonacci_cached)
```

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