Tampereen yliopisto

Markus Meskanen

METAOHJELMOINTI PYTHON 3:SSA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Marraskuu 2020

TIIVISTELMÄ

Markus Meskanen: Metaohjelmointi Python 3:ssa
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Marraskuu 2020

Toistuva ohjelmakoodi mielletään usein paheeksi, sillä mahdolliset muutokset koodiin joudutaan toteuttamaan useaan eri paikkaan, mikä aiheuttaa enemmän työtä ja inhimillisiä virheitä. Tästä syystä ohjelmointikielet tarjoavat useita työkaluja ohjelmakoodin rakentamiseksi, joiden avulla päästään eroon toistuvasta koodista ja vähennetään ohjelmoijan työmäärää.

Perinteisiä ohjelmoinnissa käytettyjä työkaluja ovat luokat ja funktiot, jotka mahdollistavat saman koodin käyttämisen useaan otteeseen. Ne hyödyntävät abstraktioperiaatetta, jonka mukaan yksittäisiä ohjelman osia ei kirjoiteta yksi kerrallaan, vaan kirjoitetaan laajempia abstraktioita kuvaamaan useita samankaltaisia osia kerrallaan. Joskus nämä työkalut eivät kuitenkaan riitä, vaan eri luokat tai funktiot jakavat keskenään toistuvaa koodia.

Yhtenä ratkaisuna toistuvaan koodiin abstrakteissa kuvauksissa toimii metaohjelmointi, joka vie abstraktioperiaatteen korkeammalle tasolle, mahdollistaen useiden samankaltaisten funktioiden tai luokkien kuvaamisen yhdellä kertaa. Toteutuksen tasolla metaohjelmointi mahdollistaa ohjelmakoodin muokkauksen ohjelman suorituksen aikana. Näin luokissa ja funktioissa toistuva koodi voidaan määritellä ohjelmoijan toimesta vain kertaalleen, jonka jälkeen se voidaan kopioida ohjelman toimesta kaikille sitä tarvitseville luokille tai funktioille ohjelman ajon aikana.

Tässä työssä esitellään Pythonin tärkeimpiä metaohjelmointityökaluja, kuten dekoraattorit, deskriptorit, sekä metaluokat. Työssä käydään läpi esimerkkejä tyypillisistä luokkien ja funktioiden käytössä syntyvistä ongelmista, sekä metaohjelmoinnin tarjoamia ratkaisuja niihin. Työssä käsitellään myös metaohjelmointityökalujen hyödyntämiä kielen ominaisuuksia, jotta työkalujen toimintaa voidaan pohjustaa ja ymmärtää syvemmin.

Avainsanat: python, metaohjelmointi, abstraktioperiaate

Sisällysluettelo

1 Johdanto

Helposti ylläpidettävän ja jatkokehitettävän ohjelmiston tärkeänä kulmakivenä toimii lähdekoodi, joka ei toista itseään [1]. Abstraktioperiaatteen mukaan lähdekoodi tulisi rakentaa siten, että toistuva toiminnallisuus tarvitsee kirjoittaa vain kertaalleen [2]. Tätä periaatetta noudattaessa mahdollisia tulevaisuuden muutoksia ei tarvitse tehdä useaan paikkaan, ja koodin rakenne pysyy ymmärrettävänä.

Modernit ohjelmointikielet tarjoavat useita eri työkaluja abstraktioperiaatteen toteuttamiseen. Yleisimpiä näistä ovat funktiot, luokat ja luokkien periyttäminen. Joskus nämä perinteiset työkalut eivät kuitenkaan riitä, vaan eri funktioiden tai luokkien välillä tulee keskenään toistoa [1]. Oikeaoppinen periyttäminen ratkaisee suuren osan ongelmista, mutta usein joudutaan silti toteuttamaan kymmeniä, joskus jopa satoja samankaltaisia luokkia, joista jokainen mahdollisesti omaan tiedostoonsa. Tämä luo suuria määriä *boilerplate*-koodia, eli rutiininomaisesti useassa paikassa toistettavaa koodia, joka toimii vain rakenteena varsinaisesti tärkeille toiminnallisuuksille ja algoritmeille [3]. Boilerplate-koodi paisuttaa ohjelman kokoa, ja siihen tehtävät muutokset vaativat laajoja, virhealttiita toimenpiteitä.

Tämä tutkielma esittelee, miten Python 3 -ohjelmointikielessä voidaan parantaa koodin rakennetta ja vähentää boilerplate-koodin määrää metaohjelmoinnin avulla. Metaohjelmointi vie abstraktioperiaatteen luokkia ja funktioita korkeammalle tasolle käsittelemällä muuta ohjelmaa tai sen osia datanaan. Tämä mahdollistaa ohjelman osien analysoinnin ja muokkauksen, tai jopa uuden ohjelman dynaamisen luomisen. [4] Sen avulla perinteiset toistuvasta koodista aiheutuvat ongelmat saadaan ratkaistua elegantisti [1]. Tutkielma ei pyri tarjoamaan yksikäsitteisiä ratkaisuja tiettyihin ongelmiin, vaan perehtyy monimuotoisiin työkaluihin, joita voidaan hyödyntää usein eri tavoin.

Työssä perehdytään ensin metaohjelmoinnin kannalta tärkeisiin Pythonin kielenrakenteisiin ja ominaisuuksiin, sekä Python-olioiden toimintaan. Myöhemmissä luvuissa käydään läpi Pythonin metaohjelmointityökaluista dekoraattorit, metaluokat ja deskriptorit. Luvuissa perehdytään työkaluihin määritelmien, ongelmien ja esimerkkien kautta.

2 Python 3 -ohjelmointikielen ominaisuuksia

Tässä luvussa tarkastellaan Python-kielen tärkeitä ominaisuuksia ja havainnollistetaan niiden toiminnallisuutta esimerkeillä. Myöhempien lukujen kannalta on tärkeä ymmärtää luvussa esitellyt toiminnallisuudet, sillä ne toimivat vankkana pohjana useille Pythonin metaohjelmointityökaluille.

2.1 Ensimmäisen luokan kansalaisuus

Pythonissa, ja usein metaohjelmoinnissa yleisesti, keskeisenä ohjelman datana toimivat luokat ja funktiot. Niitä voidaan välittää argumentteina ja paluuarvoina eteenpäin tai tallentaa muuttujiin minkä tahansa muun olion tavoin. Tällaisia rakenteita kutsutaan *ensimmäisen luokan kielen rakenteiksi* tai *ensimmäisen luokan kansalaisiksi*. [5] [6]

Pythonissa voidaan myös luoda funktioita dynaamisesti toisten funktioiden sisällä, kuten mitä tahansa muita olioita [6]. Tämä on metaohjelmointia yksinkertaisimmillaan: ohjelmaan ei tarvitse luoda useita samankaltaisia funktioita käsin, vaan voidaan toteuttaa yksi funktio, jota voi kutsua luodakseen uusia funktioita tarvittaessa.

```
1  def luo_funktio(tuloste):
2      def funktio():
3          print(tuloste)
4      return funktio
5
6  tulosta_kissa = luo_funktio('kissa')
7  tulosta_koira = luo_funktio('koira')
8
9  tulosta_kissa()  # tulostaa: 'kissa'
10 tulosta_koira()  # tulostaa: 'koira'
```

Ohjelma 1. Funktioita luova funktio

Ohjelmassa 1 on toteutettu esimerkkifunktio, jonka kutsuminen luo uuden funktion. Esimerkkifunktio vastaanottaa `tuloste`-argumentin, jonka avulla luoduista funktioista saadaan keskenään erilaisia. Lopulta se palauttaa uuden dynaamisesti luodun funktion kutsujalle, joka voi nyt kutsua uutta funktiota tulostaakseen alkuperäisen syötteensä.

2.2 Sulkijafunkio

Sulkijafunktio (engl. *closure*) on toisen funktion sisällä dynaamisesti luotu funktio, joka sitoo luontiympäristössään olleet muuttujat suoritukseensa. Se siis muistaa ulkopuolellaan olleiden muuttujien arvot eri kutsujen välillä. Sulkijafunktion sitomat muuttujat voivat myös olla sen luojafunktion vastaanottamia argumentteja. [5] [7] Muuttujan sitominen itsessään ei ole metaohjelmointia, mutta usein muut metaohjelmointityökalut hyödyntävät sulkijafunktioita.

```
1  def luo_laskuri():
2      luku = 0
3      def seuraava_luku():
4          luku += 1
5          return luku
6      return seuraava_luku
```

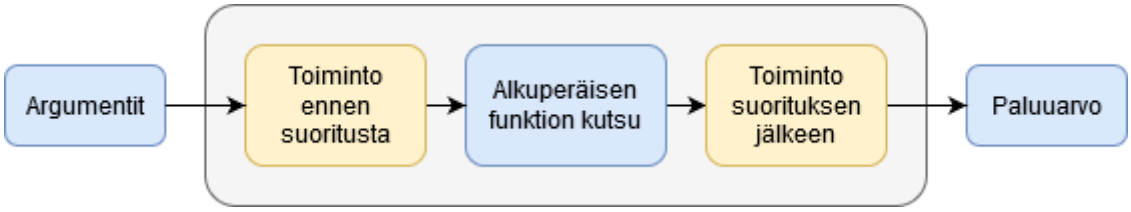
Ohjelma 2. Sulkijafunktio

Ohjelmassa 2 on havainnollistettu sulkijafunktion luontia ja käyttöä. Luojafunktio `luo_laskuri` luo jokaisella kutsullaan uuden `luku`-muuttujan sekä uuden funktion `seuraava_luku`. Jokainen `seuraava_luku()`-funktiokutsu muistaa `luku`-muuttujan arvon ja kasvattaa sitä yhdellä. Funktiota `seuraava_luku` kutsutaan sulkijafunktioksi, sillä sen ulkopuolella määritelty muuttuja `luku` on sidottu sen suoritukseen.

2.3 Kietoutujafunktio

Kietoutujafunktio (engl. w*rapper*) kietoutuu jonkin toisen funktion ympärille kutsuen sitä oman toteutuksensa sisällä. Kietoutujafunktiot hyväksyvät tyypillisesti samat argumentit

kuin kiedottava funktio ja palauttavat kiedotun funktion palauttaman paluuarvon eteenpäin kutsujalle. Kietoutujafunktio voi myös määrittää hyväksyvänsä mitä tahansa argumentteja. Tällöin se välittää kaikki argumentit eteenpäin alkuperäiselle funktiolle siirtäen vastuun argumenttien oikeellisuudesta kutsujalle. [1] Mikään ei teknisesti sido kietoutujaa toteuttamaan yllä mainittuja ehtoja, vaan se voi muokata annettuja argumentteja ja paluuarvoa haluamallaan tavalla.



Kuva 1. Kietoutujafunktion toiminnallisuus.

Kietoutumista on hahmoteltu kuvassa 1, jossa harmaa tausta kuvaa kietoutujafunktion suoritusta. Suorituksen keskeisimpänä osana on alkuperäisen funktion kutsu, mutta ennen ja jälkeen kutsua voidaan suorittaa vapaaehtoisia lisätoimintoja. Tätä havainnollistetaan ohjelmassa 3.

```
1  def funktio(n):
2      ...
3
4  def funktion_kesto(*args, **kwargs):
5      alku = time.time()
6      paluuarvo = funktio(*args, **kwargs)
7      loppu = time.time()
8      print('Funktion suoritus kesti', loppu - alku, 'sekuntia')
9      return paluuarvo
```

Ohjelma 3. Suorituksen kestoa mittaava kietoutujafunktio

Ohjelman 3 kietoutujafunktio `funktion_kesto` määrittää vastaanottavansa mitä tahansa argumentteja ja avainsana-argumentteja Pythonin `*args` ja `**kwargs` -kieliopeilla. Rivillä 5 suoritetaan alustava toiminto, jossa tallennetaan lähtöaika ennen alkuperäisen funktion suoritusta. Seuraavaksi rivillä 6 suoritetaan alkuperäinen funktio ja tallennetaan sen paluuarvo muuttujaan. Suorituksen jälkeisenä toimintona tallennetaan loppuaika ja tulostetaan alku- ja loppuaikojen erotus. Lopulta kietoutujafunktio palauttaa vastaanottamansa paluuarvon eteenpäin. Ohjelman kietoutujafunktio siis vastaanottaa samat argumentit kuin alkuperäinen funktio ja palauttaa saman paluuarvon kuin alkuperäinen funktio imitoiden täysin sen suoritusta kutsujan näkökulmasta.

3 Python-olioiden toiminta

3.1 Olioiden luonti

Pythonin oliot instansoidaan kirjoittamalla luokan nimen perään sulkeet syntaksilla `Luokka(...)`. Tämä suorittaa funktiokutsun luokan `__new__`-metodiin, joka vastaan-

ottaa kutsuvan luokan argumenttinaan. Mikäli `__new__()` palauttaa jonkin olion, kutsutaan tälle oliolle automaattisesti metodia `__init__()`, jossa tyypillisesti alustetaan olion instanssiattribuutit. Tästä syystä `__init__()` ei saa palauttaa mitään arvoa. [8]

Luokkametodi `__new__()` luo tyypillisesti uuden olion `object.__new__`-metodilla, yleensä kutsumalla `super().__new__()`:tä omassa toteutuksessaan. Ohjelmassa 4 on kuitenkin havainnollistettu, kuinka `__new__` voi myös palauttaa jonkin jo olemassa olevan olion. Ohjelma luo singleton-luokan eli luokan, josta voidaan luoda vain yksi instanssi.

```
1  class Singleton:
2      _instance = None
3
4      def __new__(cls, *args, **kwargs):
5          if cls._instanssi is None:
6              cls._instanssi = super().__new__(cls)
7          return cls._instanssi
```

Ohjelma 4. Singleton-toteutus `__new__`-metodilla

Metodin `__new__` tulee vastaanottaa samat argumentit kuin vastaavan luokan `__init__`-metodin. On myös tärkeää huomata, että `__new__` on *staattinen metodi* eli se ei vastaanota metodin kutsujaa automaattisesti argumenttinaan. Tästä syystä Python ei välitä kantaluokan `super().__new__()` kutsulle `cls` argumenttia implisiittisesti, vaan kutsujan tulee välittää se manuaalisesti eteenpäin. [8]

3.2 Attribuuttien toiminta

Tavallisilla Python olioilla on `__dict__` niminen instanssiattribuuttikirjasto, joka on oletuksena tyhjä `dict`-tyyppinen olio. Kun Python-oliolle lisätään uusi attribuutti, se tallennetaan instanssiattribuuttikirjastoon avain-arvo-parina siten, että avaimeksi tallennetaan attribuutin nimi ja arvoksi attribuutin arvo. [1] Esimerkiksi `oppilas.arvosana = 5` lisäisi `('arvosana', 5)`-avain-arvo-parin oppilaan instanssiattribuuttikirjastoon.

Pythonin standardikirjasto tarjoaa myös `getattr(olio, attr)` ja `setattr(olio, attr, arvo)` -funktiot, joilla voidaan hakea ja asettaa olion attribuutti `string`-tyyppisen `attr`-argumentin avulla [9]. Attribuuttifunktioiden toimintaa on verrattu tavalliseen attribuuttien päivittämiseen toiminnaltaan identtisissä ohjelmissa 5a ja 5b. Attribuuttifunktiot ovat erityisen käteviä metaohjelmoinnin yhteydessä, kun attribuutteja halutaan käsitellä dynaamisesti jonkin merkkijonon perusteella.

a.
```
1  hinta = tuote.hinta
2  tuote.hinta = hinta + 5
```

b.
```
1  hinta = getattr(tuote, 'hinta')
2  setattr(tuote, 'hinta', hinta + 5)
```

Ohjelma 5. Attribuutin käyttäminen a. piste-operaattorilla ja b. attribuuttifunktioilla

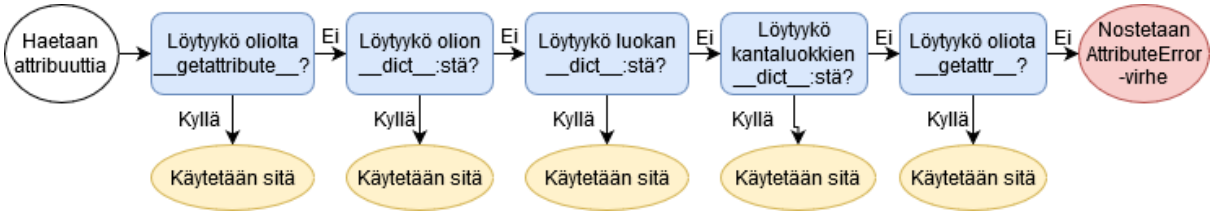
Olion attribuuttien haku- ja asettamisoperaatioiden toimintaa voidaan muokata, tapahtui operaatio sitten piste-operaattorin tai `getattr` ja `setattr` -funktioiden kautta. Mikäli asettamis-operaatiota halutaan muokata, voidaan oliolle toteuttaa `__setattr__`-metodi. [10]

```
1  class Luokka:
2      def __setattr__(self, attr, value):
3          if not isinstance(value, int):
4              raise ValueError('ei kokonaisluku')
5          super().__setattr__(attr, value)
```

Ohjelma 6. Metodin `__setattr__` toteuttaminen

Ohjelman 6 luokka varmistaa, että luokan instansseille voi asettaa vain kokonaislukuarvoja attribuuteiksi. Erityisen tärkeää on huomata, että attribuutin arvo tulee lopulta asettaa kutsumalla kantaluokan metodia `super().__setattr__()`, eikä asettamalla attribuuttia suoraan `self`-oliolle, sillä tämä aiheuttaisi ikuisen `__setattr__()`-funktiokutsurekursion. Attribuuttien asettamiseen liittyy lisäksi *deskriptorit*, joihin perehdytään tarkemmin luvussa kuusi.

Attribuuttien hakuoperaatio on asettamista monivaiheisempi prosessi. Siinä missä asettaminen tapahtuu aina instanssilta haetulla `__setattr__`-funktiolla, hakuun liittyy kaksi eri metodia, `__getattribute__` ja `__getattr__`, sekä olion luokan ja kantaluokkien instanssiattribuuttikirjastot. [10] Kuten attribuuttien asettamiseen, myös niiden hakuun liittyy lisäksi deskriptorit, joihin perehdytään tarkemmin luvussa 6.



Kuva 2. Attribuutin hakuprosessi

Kuvasta 2 nähdään, että `__getattribute__`-metodi on ensimmäisellä prioriteetilla, mikäli sellainen löytyy. Tällä metodilla voidaan siis muokata attribuutin haun toiminnallisuutta riippumatta siitä, onko kyseinen attribuutti olemassa. Tämän jälkeen attribuuttia etsitään olion, luokan, sekä kantaluokkien `__dict__`-instanssiattribuuttikirjastoista. Mikäli attribuuttia ei löydy, etsitään oliolta vielä metodia `__getattr__`, joka määrittää toiminnallisuuden puuttuvan attribuutin tapauksessa. Mikäli oliolle ei ole määritelty myöskään `__getattr__`-metodia, Python nostaa `AttributeError`-virheen. [10]

3.3 Luokkametodit ja staattiset metodit

Pythonin standardikirjasto tarjoaa `classmethod` ja `staticmethod` -funktiot, joilla voi luoda metodeille poikkeuksellista toiminnallisuutta. `classmethod`:in avulla voidaan toteuttaa luokkametodi, joka vastaanottaa ensimmäisenä argumenttinaan kutsuvan luokan, eikä luokan instanssia. Tästä syystä luokkametodien ensimmäinen argumentti on tyypillisesti nimeltään `cls`, eikä `self`. [9]

```
class Laatikko:

    def __init__(self, leveys, korkeus, syvyys):
        self.leveys = leveys
        self.korkeus = korkeus
        self.syvyys = syvyys

    @classmethod
    def luo_kuutio(cls, leveys):
        return cls(leveys, leveys, leveys)
```

Ohjelma 7. Luokkametodi

Ohjelmassa 7 käytetään luokkametodia vaihtoehtoisena rakentajana `Laatikko`-luokalle. Luokkametodin avulla aliluokat voivat käyttävää samaa `aliluokka.luo_kuutio()` rakentajaa luodakseen oman instanssin. Koodista huomataan, että `classmethod` käyttää erityistä *dekoraattori*syntaksia, `@`-merkkiä. Dekoraattorisyntaksin toimintaan perehdytään tarkemmin luvussa neljä. Tärkeää on ymmärtää, että dekoraattorit operoivat niiden alapuolella olevaa funktiota tai luokkaa.

`staticmethod` toimii `classmethod`in tavoin, mutta se luo staattisia metodeja. Staattiset metodit eivät vastaanota kutsujaan liittyvää argumenttia, eikä niiden toiminnallisuus siis riipu kutsujasta. Myös `staticmethod`in kanssa käytetään tyypillisesti dekoraattorisyntaksia, `@staticmethod`. [9]

3.4 Propertyt

Mikäli jonkin tietyn attribuutin haku-, asettamis- ja/tai tuhoamisoperaatioiden toiminnallisuutta halutaan muokata, voidaan käyttää `property`-funktiota. Se muokkaa attribuuttisyntaksin kutsumaan instanssimetodeja tavallisen attribuutin asettamisen sijaan. [9]

```
class Laatikko:
    ...

    @property
    def tilavuus(self):
        return self.leveys * self.korkeus * self.syvyys
```

Ohjelma 8. Tilavuuden laskeminen `property`:n avulla

Ohjelmassa 8 toteutetaan metodi, joka laskee laatikon tilavuuden muista attribuuteista. Metodi on merkattu `property`-dekoraattorilla, jolloin sitä tulee käyttää niin kuin se olisi attribuutti, eli funktiokutsusulkeita ei kirjoiteta. Esimerkiksi tilavuuden tulostaminen hoituisi koodilla `print(laatikko.tilavuus)`. Tämä mahdollistaa muun muassa attribuutin toiminnallisuuden muokkaamisen, ilman että kutsujan tarvitsee muuttaa syntaksia. [10]

Toinen yleinen käyttötarkoitus `property`:lle on attribuutin piilottaminen vain-luku-attribuutiksi. Tällöin varsinaisen attribuutin nimi aloitetaan käytännön mukaisesti alaviivalla, ja property nimetään vastaavaksi ilman alaviivaa [9]. Tämä on erityisen hyödyllistä

sillon, kun jonkin attribuutin arvo halutaan pitää samana koko olion eliniän ajan, mutta sen tulee silti olla julkisesti luettava arvo.

```
1  class Opiskelija:
2
3      def __init__(self, opnro):
4          self._opnro = opnro
5
6      @property
7      def opnro(self):
8          return self._opnro
```

Ohjelma 9. Vain-luku-attribuutti

Ohjelmassa 9 piilotetaan varsinainen instanssiattribuutti alaviivan taakse, ja käytetään `property`:ä luodaksemme vain-luku-attribuutti `opnro`. Kielen toiminnan kannalta mikään ei estä muuttamasta `self._opnro`-attribuutin arvoa, mutta Pythonin vahvasti juurtuneet käytännöt merkkaavat alaviivalla alkavat attribuutit yksityisiksi [11].

Luodulla `property`:llä on lisäksi olemassa `setter` ja `deleter` -metodit, jotka määrittelevät toiminnallisuuden, kun attribuutin arvoa yritetään vaihtaa yhtäsuuruusmerkillä, tai kun attribuutti yritetään tuhota Pythonin `del`-avainsanalla. [9]

```
1  class Tentti:
2
3      def __init__(self):
4          self._arvosana = 0
5
6      @property
7      def arvosana(self):
8          return self._arvosana
9
10     @arvosana.setter
11     def arvosana(self, arvo):
12         if not isinstance(arvo, int):
13             raise ValueError('ei ole kokonaisluku')
14         if arvo < 0 or arvo > 5:
15             raise ValueError('tulee olla väliltä 0-5')
16         self._avosana = arvo
```

Ohjelma 10. Attribuutin asettaminen `property setter`:illä.

Ohjelman 10 operaatio käyttää `setter`:iä varmistuakseen arvosanan oikeellisuudesta, jonka jälkeen se muokkaa varsinaista `_arvosana`-attribuuttia. Ohjelmasta selviää myös `setter`-metodin syntaksi. Metodi vastaanottaa ainoana argumenttinaan yhtäsuuruusoperaattorilla asetetun arvon.

3.5 Erityismetodit

Pythonin *erityis-* eli *taikametodeiksi* kutsutaan virallisesti kaikkia niitä metodeja, jotka alkavat ja loppuvat kahteen alaviivaan. Niiden päätarkoitus on muokata olion toimintoja

kielen operaattoreiden toimesta. Esimerkiksi `__getattr__()` ja `__setattr_()` muokkaavat olion piste-operaattorin toimintaa, ja `__init__()` ja `__new__()` muokkaavat olion instansointia. [8] Taikametodeja on liikaa läpikäytäväksi tässä työssä, mutta käydään seuraavaksi läpi muutamia esimerkkejä.

Vertailuoperaattoreiden toimintaa on usien järkevää muokata, mikäli luokan olioita voidaan verrata toisiinsa yksikäsitteisesti. Näiden vertailuoperaattori-erityismetodien nimet juontuvat englanninkielestä. Esimerkiksi suurempi kuin -operaattorin (>) toimintaa muokataan englannin kielen "greater than" sanoista juontuneella `__gt__`-erityismetodilla. Vastaavasti __eq__ tulee sanasta "equals", joka määrittää yhtäsuuruusvertailun. Metodit vastaanottavat argumenttina olion, johon `self`-oliota ollaan vertaamassa. [8] Muita yleisiä erityismetodeja on matemaattiset operaatiot kuten yhteenlasku ja kertolasku, `__add__` ja `__mul__`, sekä tyyppimuunnokset, kuten `__str__` ja `__bool__`.

4 Dekoraattorit

4.1 Dekoraattorien toimintaperiaate

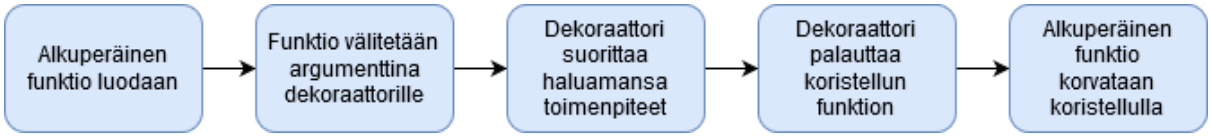
Dekoraattorit eli *koristelijat* ovat funktioita tai luokkia, jotka hyväksyvät toisen funktion tai luokan argumenttinaan ja palauttavat jonkin funktion tai luokan paluuarvonaan. Niiden käytölle on olemassa oma syntaksi, `@dekoraattori`-koodirivi. Erityissyntaksin käyttäminen suorittaa dekoraattorifunktiokutsun välittömästi, korvaten alkuperäisen funktion tai luokan paluuarvollaan. [1] Ohjelmassa 11a dekoroidaan eli koristellaan funktio perinteisesti, ja ohjelmassa 11b käytetään dekoraattorisyntaksia. Toiminnallisuudeltaan tavat ovat identtiset.

a.
```
1  def funktio(n):
2      ...
3  funktio = ajastin(funktio)
```

b.
```
1  @ajastin
2  def funktio(n):
3      ...
```

Ohjelma 11. a. Perinteinen dekoroiti ja b. @-syntaksilla dekoroiti

Ohjelmat siis kutsuvat kuvitteellista dekoraattorifunktiota `ajastin`, välittäen tälle argumenttina funktion `funktio`. Dekoraattori palauttaa uuden funktion, jolla korvataan alkuperäinen `funktio`-muuttujan arvo. Tämä toiminnallisuus on hahmoteltu kuvassa 3.



Kuva 3. Dekoraattorin toiminnan elinkaari

Dekoraattoreita käytetään siis toisten luokkien tai funktioiden koristeluun niiden luonnin jälkeen. Ne mahdollistavat periytymishierarkiasta irrallisen funktioiden tai luokkien toiminnallisuuden muokkaamisen [1]. Dekoraattorien käyttö toteuttaa abstraktioperiaatetta,

siirtäen yhteisen koodin esimerkiksi useassa funktiossa käytettävän ajastimen toteutuksesta yhteen dekoraattoriin.

Koristelun yhteydessä valmiille funktiolle tai luokalle voidaan tehdä jokin toiminto, kuten tallentaa se listaan, muokata sen nimeä, tai luokkien tapauksessa jopa muokata sen metodeja. Kietoutujafunktioihin yhdistettynä dekoraattoreilla voidaan jopa muokata funktion toimintaan ennen sen suoritusta tai suorituksen jälkeen.

```
1  funktio_lista = []
2
3  def tallenna_funktio(koristeltava):
4      funktio_lista.append(koristeltava)
5      return koristeltava
6
7  @tallenna_funktio
8  def funktio(n):
9      ...
```

Ohjelma 12. Itse toteutettu dekoraattorifunktio

Ohjelmassa 12 on toteutettu dekoraarifunktio, jonka ainoa tehtävä on tallentaa koristeltava funktio listaan `funkio_lista`. Se vastaanottaa koristeltavan olion argumenttina, lisää sen listaan, ja palauttaa koristeltavan olion sellaisenaan, eli koristeltavan funktion toimintaa ei muokata mitenkään. Python kielen toiminnan kannalta on kuitenkin mahdollista palauttaa `tallenna_funktio`-dekoraattorista mikä tahansa olio. Tätä hyödynnetään lähinnä kietoutujafunktioiden palauttamisessa.

4.2 Dekoraattorin toteuttaminen funktioilla

Toteutetaan seuraavaksi oma dekoraattori, jonka tehtävänä on helpottaa ohjelman debuggausta, eli virheiden paikantamista. Dekoraattorin tehtävänä on tulostaa funktiokutsun yhteydessä funktion vastaanottamat argumentit, sekä palautettu paluuarvo. Alla on esitelty esimerkki dekoraattorin käytöstä. Nuolella (>) merkatut rivit ovat syötteitä ohjelmalle, loput rivit ovat ohjelman tulosteita.

```
> @debuggaa
> def esimerkki(x, y):
>     return x + y
>
> esimerkki(2, 3)
Kutsu: esimerkki, *args=(2, 3), **kwargs={}
Paluu: esimerkki, paluuarvo=5
5
```

Esimerkistä huomataan, että toteutettavan dekoraattorin ei tule muokata funktion varsinaista toiminnallisuutta, vaan ainoastaan tulostaa sille annetut argumentit ja sen paluuarvo. Esimerkkiä noudattava `debuggaa`-dekoraattori on toteutettu ohjelmassa 13 käyttäen funktioita.

```
def debuggaa(funktio):
    nimi = funktio.__nimi__

    def koristeltu(*args, **kwargs):
        print(f'Kutsu: {nimi}, *args={args}, **kwargs={kwargs}')
        paluuarvo = funktio(*args, **kwargs)
        print(f'Paluu: {nimi}, paluuarvo={paluuarvo}')
        return paluuarvo

    return koristeltu
```

Ohjelma 13. Esimerkki funktiodekoraattori, `debuggaa`

Ohjelmasta nähdään, että dekoraattori saa argumenttina koristeltavan funktion, ja palauttaa uuden `koristeltu`-kietoutujafunktion. Palautettava funktio tulostaa saamansa argumentit, kutsuu alkuperäistä funktiota kyseisillä argumenteilla, tulostaa saamansa paluuarvon, ja lopulta palauttaa kyseisen arvon eteenpäin. Se siis imitoi täysin alkuperäisen funktion toiminnallisuutta, mutta koristelee sitä tulostuksilla ennen alkuperäisen funktion suoritusta ja sen jälkeen.

4.3 Dekoraattorin toteuttaminen luokalla

Dekoraattoreita voidaan toteuttaa myös luokkien avulla, mikä on erityisen kätevää, kun tarvitaan paljon välimuuttujia – kuten luvun 4.2 `debuggaa`-esimerkin `nimi`-muuttuja – tai kun dekoraattorin koko meinaa paisua liian suureksi. Uuden olion luonti on syntaksiltaan identtinen funktiokutsun kanssa, lukuunottamatta käytännöksi muodostunutta isoa alkukirjainta, jolloin dekoraattorifunktiokutsu voidaan korvata suoraan luokan instansoinnilla. Lisäksi palautettava instanssi pitää muokata kutsuttavaksi olioksi, jotta se voi imitoida alkuperäisen funktion toimintaa. Tämä onnistuu helposti erityismetodilla `__call__`, joka määrittelee mitä tapahtuu, kun kirjoitetaan sulkeet instanssin perään [8].

```
class Debuggaa:

    def __init__(self, funktio):
        self.funktio = funktio
        self.nimi = funktio.__nimi__

    def __call__(self, *args, **kwargs):
        print(f'Kutsu: {self.nimi}, *args={args}, **kwargs={kwargs}')
        paluuarvo = self.funktio(*args, **kwargs)
        print(f'Paluu: {self.nimi}, paluuarvo={paluuarvo}')
        return paluuarvo
```

Ohjelma 14. Esimerkki luokkadekoraattori, `debuggaa`

Ohjelmassa 14 on toteutettu toiminnaltaan luvun 4.2 `debuggaa`-dekoraattoria vastaava luokkadekoraattori. Luokalla toteutetun dekoraattorin käyttö ei eroa mitenkään funktioilla toteutetun dekoraattorin käytöstä, sillä molemmat ottavat vastaan yhden argumentin ja palauttavat kutsuttavan olion. Yksinkertaisen esimerkin tapauksessa luokalla toteutettu dekoraattori on usein monimutkaisempi ja vaikeampi hahmottaa kuin vastaava funktio,

mutta mitä monimutkaisemmaksi dekoraattorin toteutus paisuu, sitä selvemmäksi luokka toteutus usein muuttuu suhteessa funktiolla toteutettuun versioon.

4.4 Argumentilliset dekoraattorit

Joskus dekoraattorit vaativat lisäargumentteja niiden toiminnallisuuden tarkentamiseksi. Kielen määritelmän mukaan @-syntaksi ei kuitenkaan vastaanottaa koristeltavan olion lisäksi mitään argumentteja. Tämä ongelma voidaan kiertää luomalla argumentteja vastaanottava funktio, joka luo uuden dekoraattorin ja palauttaa sen @-operaattorille. [1]

```
1  def tallenna_funktio(kohdelista):
2      def dekoraattori(koristeltava):
3          kohdelista.append(koristeltava)
4          return koristeltava
5      return dekoraattori
6
7  @tallenna_funktio(funktio_lista)
8  def funktio(n):
9      ...
```

Ohjelma 15. Argumentillinen dekoraattori

Ohjelmassa 15 on toteutettu yksinkertainen `tallenna_funktio` dekoraattori siten, että se vastaanottaa halutun kohdelistan argumenttina. Nyt funktio `tallenna_funktio` ei itsessään ole varsinaisesti dekoraattori, vaan sen funktiokutsu palauttaa todellisen dekoraattorin. Käytännössä tämä mahdollistaa argumentillisen dekoraattorisyntaksin.

Argumentilliset dekoraattorit voidaan toteuttaa myös luokkien avulla, jolloin luokan `__init__`-metodi vastaanottaa dekoraattorin argumentit, palautettu instanssi on varsinainen dekoraattori, ja `__call__` vastaanottaa koristeltavan olion [6].

```
1  class Debuggaa:
2
3      def __init__(self, tiedosto):
4          self.tiedosto = tiedosto
5
6      def __call__(self, koristeltava):
7          nimi = koristeltava.__name__
8
9          def kietoutuja(*args, **kwargs):
10             with open(self.tiedosto) as f:
11                 f.write(f'Kutsu: {nimi}, *args={args}, **kwargs={kwargs}')
12             paluuarvo = koristeltava(*args, **kwargs)
13             with open(self.tiedosto) as f:
14                 f.write(f'Paluu: {nimi}, paluuarvo={paluuarvo}')
15             return paluuarvo
16
17         return kietoutuja
```

Ohjelma 16. Argumentillinen dekoraattoriluokka

Ohjelmassa 16 on toteutettu luvun 4.3 `Debuggaa`-esimerkki siten, että se vastaanottaa argumenttina tiedoston, johon tulosteet tulisi tallentaa. Nyt syntaksi `@Debuggaa('tiedosto.txt')` luo ensin uuden `Debuggaa`-instanssin argumentilla `'tiedosto.txt'`, jonka jälkeen palautettua instanssia käytetään dekoraattorina, kutsuen sen metodia `__call__`. Tästä eteenpäin prosessi toimii tavallisen funktioilla toteutetun dekoraattorin tavoin.

5 Metaluokat

5.1 Luokkien luokka

Pythonin luokat ovat ensimmäisen luokan kansalaisia, eli tavallisia Python-olioita, joten myös niillä on oltava jokin tyyppi. Siinä missä kokonaisluvun `5` tyyppi on `int`, on jokaisen luokan tyyppi oletuksena `type`. Tästä syystä luokkia voidaan instansoida dynaamisesti `type`-luokasta. Sen rakentaja vastaanottaa argumentteina uuden luokan nimen, sen kantaluokat, sekä instanssiattribuuttikirjaston. [9] Ohjelmissa 17a ja 17b on esitelty kaksi toiminnallisuudeltaan identtistä tapaa luoda uusi luokka; Pythonin perinteinen `class`-syntaksi, sekä `type`-luokan instansointi.

a.
```
class Luokka(Kantaluokka):
    x = 5
    def f(self):
        return self.x + 3
```

b.
```
def f(self):
    return self.x + 3

Luokka = type('Luokka', (Kantaluokka,), {'x': 5, 'f': f})
```

Ohjelma 17. Uusi luokka a. perinteisesti ja b. `type`:stä instansoimalla

Pythonin `type` on siis luokka, jonka instanssit ovat toisia luokkia. Tällaista luokkaa kutsutaan *metaluokaksi*. Metaluokka määrittelee muiden luokkien toiminnan, samalla tavoin kuin luokka määrittele omien instanssiensa toiminnan. [8]

5.2 Oman metaluokan periyttäminen

Metaluokasta `type` voidaan myös periyttää toisia metaluokkia, joiden avulla voidaan muokata sen instassiluokkien toimintaa. Oma metaluokka periytetään `type`:stä normaalilla periyttämissyntaksilla. Metaluokan ja luokan suhde toisiinsa on sama kuin tavallisen luokan ja instanssin suhde, joten metaluokalla voi toteuttaa luokille samat asiat kuin luokalla voi olioille. Metaluokat siis tukevat kaikkia erityismetodeja, niillä voi muokata luokkien instansointia, tai vaikkapa attribuuttien toiminnallisuutta. [8]

```
class NimiMeta(type):

    @property
    def nimi(cls):
        return cls.__name__
```

Ohjelma 18. Metaluokan periyttäminen

Ohjelmassa 18 on toteutettu yksinkertainen metaluokka, joka lisää luokilleen `nimi`-propertyn. Metaluokkien instanssimetodeissa on usein järkevää korvata `self`-argumentti `cls`-argumentilla, sillä tämä helpottaa hahmottamaan metaluokan toiminnallisuutta. Vastaavasti metaluokan metodissa `__new__` on tyypillistä korvata `cls`-argumentti `meta`-argumentilla. Tästä hieman monimutkaisempi metaluokka esimerkki ohjelmassa 19, jossa metaluokka tallentaa kaikki instanssinsa listaan.

```
class LuokkaTallennin(type):
    _tallennetut_luokat = []

    def __new__(meta, name, bases, attrs):
        cls = super().__new__(meta, name, bases, attrs)
        meta._tallennetut_luokat.append(cls)
        return cls
```

Ohjelma 19. Metaluokan `__new__`-metodi

Ohjelmassa metaluokka `LuokkaTallennin` lisää kaikki instanssinsa luomisen yhteydessä `_tallennetut_luokat`-listaan. Metaluokan `__new__()` vastaanottaa samat argumentit kuin `type`-luokka. Varsinainen luokka instansoidaan kantaluokan kutsulla `super().__new__(...)`. Metaluokka voi tarvittaessa vastaanottaa ylimääräisiä avainsana-argumentteja yllä olevien neljän pakollisen argumentin lisäksi. [8]

5.3 Metaluokan käyttäminen

Oma luokka voidaan instansoida metaluokasta kahdella tavalla. Ensimmäinen ja harvinaisempi tapa on tavallinen instansointisyntaksi, joka kutsuu metaluokan rakentajaa. Yleisempi ja usein mielekkäämpi tapa on Pythonin `class`-syntaksi, jolloin metaluokka annetaan `metaclass`-argumenttina periytymissulkeiden sisälle, muodossa `class Luokka(metaclass=OmaMetaluokka)`.

Toisin kuin dekoraattorit, metaluokat ovat sidottu periytymishierarkiaan. Riittää siis määritellä metaluokka hierarkian ylimmän tason luokalle, ja kaikki aliluokat perivät saman metaluokan automaattisesti. [1] Näin voidaan edistää abstraktioperiaatetta viemällä useassa luokassa toistettava koodi yhteiseen metaluokkaan. Tätä on havainnollistettu ohjelmassa 20, joka käyttää ohjelman 19 `LuokkaTallennin`-metaluokkaa.

```
class EsimerkkiLuokka(metaclass=LuokkaTallennin):
    ...

class ToinenLuokka(EsimerkkiLuokka):
    ...

print(LuokkaTallennin._tallennetut_luokat)
# tulostus: [<class 'EsimerkkiLuokka'>, <class 'ToinenLuokka'>]
```

Ohjelma 20. Metaluokan käyttö ja periytyminen

Tästä syystä metaluokat ovat erityisen hyödyllinen työkalu esimerkiksi Python-kirjastoille, joiden tarvitsee muokata kaikkien aliluokkien toimintaa. [1] Instansoitava luokka voi lisäksi välittää ylimääräisiä avainsana-argumentteja metaluokalle periytymissulkeissa, edellyttäen että metaluokan rakentaja hyväksyy kyseiset argumentit. [8]

5.4 `__init_subclass__`-luokkametodi

Usein metaluokan ainoa tarkoitus on suorittaa jokin toiminto uutta aliluokkaa luotaessa, kuten ohjelman 19 `LuokkaTallennin`-metaluokan tapauksessa. Tämä monimutkaistaa luokkahierarkiaa melko paljon suhteessa saavutettuun hyötyyn. Tästä syystä Pythonin versio 3.6 esitteli uuden luokkametodin, `__init_subclass__` [12]. Metodi tarjoaa *kantaluokalle* mahdollisuuden suorittaa toimintoja sen aliluokille, jolloin usein on helpompi jättää metaluokka kokonaan toteuttamatta.

```
class TallennettavaLuokka:  # Ei periydy type:stä!
    _tallennetut_luokat = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        TallennettavaLuokka._tallennetut_luokat.append(cls)
```

Ohjelma 21. Luokkametodin `__init_subclass__` käyttö

Ohjelmassa 21 on toteutettu ohjelmassa 19 esitellyn metaluokan `LuokkaTallennin` toiminnallisuus, käyttäen nyt metodia `__init_subclass__`. Uusi `TallennettavaLuokka` ei käytä metaluokkaa, vaan luokkien tallentaminen hoituu periyttämällä tallennettavat luokat kyseisestä luokasta. Tämä on erityisen hyödyllistä silloin, kun periyttäminen vaaditaan joka tapauksessa jonkin toisen ominaisuuden vuoksi.

6 Deskriptorit

Deskriptorit ovat hyödyllinen työkalu Pythonin metaohjelmoinnissa, ja niiden ymmärtäminen avaa paremmin Pythonin sisäistä toimintaa. Useat Pythonin tarjoamat työkalut, kuten aiemmin esitellyt `property` ja `classmethod`, eivät toimisi ilman deskriptoreita. Myös jokainen olioille suoritettu metodikutsu, `olio.metodi(…)`, muokataan deskriptorien avulla lopulliseen `Luokka.metodi(self=olio, …)`-muotoon. Tässä luvussa tutustutaan yleisellä tasolla siihen mitä deskriptorit ovat, ja miten niitä voi itse toteuttaa.

6.1 Deskriptorien toimintaperiaate

Deskriptorien tarkoitus on muokata olion attribuuttien toiminnallisuutta. Tavallisesti pisteoperaattori `obj.attr` hakee, asettaa tai poistaa attribuutin instanssiattribuuttikirjastosta, mutta mikäli instanssin luokalla on vastaavan niminen deskriptori-attribuutti, voi se muokata jotakin tai useampaa näistä toiminnoista. [13]

Deskriptori-olio luodaan luokan nimiavaruuteen tavalliseksi luokka-attribuutiksi, kuten alla olevassa esimerkissä.

```
class Luokka:
    esimerkki = OmaDeskriptori()
```

Mikäli kuvitteellinen `OmaDeskriptori`-luokka nimensä mukaisesti toteuttaa Pythonin *deskriptoriprotokollaa*, se voi muokata luokan instanssien `esimerkki`-attribuutin toiminnallisuutta. Deskriptoriprotokolla vaatii `__get__`, `__set__` tai `__delete__` -metodien toteuttamisen luokalle. Metodi `__get__` muokkaa attribuutin haku-, `__set__` asettamis- ja `__delete__` tuhoamistoiminnallisuutta. [13] Luokan `OmaDeskriptori` tulisi siis toteuttaa jokin tai useampia näistä metodeista ollakseen deskriptori.

6.2 Oman deskriptorin toteuttaminen

Toteutetaan oma deskriptori, joka hakee opiskelijan kouluarvosanat kuvitteellisesta tietokannasta opiskelijan `opnro`-instanssiattribuutin perusteella. Deskriptorille annetaan argumenttina tietokanta ja tarvittavat hakuavaimet, mutta esimerkissä ei perehdytä oikean tietokannan toimintaan. Ohjelma 22 havainnollistaa vielä toistaiseksi toteuttamattoman `TietokantaAttr`-deskriptorin käyttöä.

```
1  class Opiskelija:
2      arvosanat = TietokantaAttr(
3          opiskelija_tietokanta,
4          taulu='arvosana',
5      )
6
7      def __init__(self, opnro):
8          self.opnro = opnro
9
10     def laske_keskiarvo(self):
11         return sum(self.arvosanat) / len(self.arvosanat)
```

Ohjelma 22. Deskriptorin käyttöesimerkki

Ohjelmasta nähdään, kuinka tietokanta-attribuuttia voi käyttää minkä tahansa instanssiattribuutin tavoin. Erona on sen toiminnallisuus; deskriptori on voinut toteuttaa attribuutin taustalle tietokannasta haun ja tietokantaan tallentamisen, jota ei näe päällepäin.

Ohjelmassa 23 on implementoitu deskriptorprotokollan `__get__`-metodi palauttamaan tietokannasta haettu arvo, `__set__`-metodi kirjoittamaan tietokantaan uutta dataa, ja `__delete__`-metodi tuhoamaan tietokannasta opiskelijan arvosanat.

```
class TietokantaAttr:

    def __init__(self, tietokanta, taulu):
        self.tietokanta = tietokanta
        self.taulu = taulu

    def __get__(self, obj, type_=None):
        return self.tietokanta.hae(self.taulu, obj.opnro)

    def __set__(self, obj, value):
        self.tietokanta.kirjoita(self.taulu, obj.opnro, value)

    def __delete__(self, obj):
        self.tietokanta.poista(self.taulu, obj.opnro)
```

Ohjelma 23. Tietokanta-attribuutti-deskriptorin esimerkkitoteutus

Tärkeää on huomata, kuinka deskriptoriprotokollan metodeissa saadaan parametrinä deskriptoria käyttävä instanssi [13]. Tämän avulla voidaan käsitellä oikean olion tietoja, tässä tapauksessa olion opiskelijanumeron avulla. Esimerkkiin on yksinkertaisuuden vuoksi kovakoodattu `obj.opnro`-attribuutin käyttö, mutta todelliselle deskriptorille olisi suotavaa antaa attribuutin nimi argumenttina, ja hakea ja asettaa attribuutin arvoa annetun argumentin perusteella `getattr` ja `setattr` -funktioilla. Tämä mahdollistaisi `TietokantaAttr`-deskriptorin käytön muissakin luokissa, eikä sitoisi sitä `opnro`-attribuuttiin.

6.3 Data- ja ei-datadeskriptorit

Deskriptorit voidaan jakaa kahteen kategoriaan, datadeskriptoreihin ja ei-datadeskriptoreihin (engl. *non-data* descriptor). Ei-datadeskriptorit ovat deskriptoreja, jotka määrittelevät vain protokollan `__get__`-metodin. Datadeskriptorit puolestaan ovat kaikki deskriptorit, joilla on määriteltynä `__set__` tai `__delete__` -metodi, riippumatta niiden `__get__`-metodin olemassaolosta. [13]

Deskriptorit myös monimutkaistavat luvussa 3.2 läpikäytyä tavallisen attribuutin hakuprioriteettia. Ei-datadeskriptorit ovat hakuprioriteetiltään instassiattribuuttikirjaston *jälkeen*, jolloin saman nimisen instanssiattribuutin luonti piilottaa deskriptorin. Datadeskriptorit ovat hakuprioriteetiltään *ennen* instanssiattribuutteja. Ne siis kykenevät muokkaamaan tai jopa estämään instanssiattribuutin asettamisen. [13]

6.4 `__set_name__`-metodi

Joskus deskriptorin on tärkeää tietää minkä nimistä attribuuttia se käsittelee. Tarpeellisuuden hahmottamiseksi ohjelmassa 24 on toteutettu epäonnistunut esimerkki arvosana-deskriptorista.

```
class Arvosana:

    def __init__(self):
        self._arvosana = 0

    def __get__(self, obj, type_=None):
        return self._arvosana

    def __set__(self, obj, value):
        if value < 0 or value > 5:
            raise ValueError('arvosanan tulee olla väliltä 0-5')
        self._arvosana = value

class Oppilas:
    matematiikan_arvosana = Arvosana()

    def __init__(self, nimi, matematiikan_arvosana):
        self.nimi = nimi
        self.matematiikan_arvosana = matematiikan_arvosana

o1 = Oppilas('Antti', 4)
o2 = Oppilas('Matti', 2)
print(o1.keskiarvo, o2.keskiarvo)
# tulostus: 2 2
```

Ohjelma 24. Virheellinen `Arvosana`-deskriptori

Kuten lopun tulostuksesta huomataan, on Matin arvosanan asettaminen muokannut myös Antin arvosanaa. Tämä johtuu siitä, että `Arvosana`-instansseja on luotu vain yksi kappale `Oppilas`-luokan sisällä, jolloin myös attribuutteja `_arvosana` on vain yksi, kaikille oppilaille yhteinen kappale. Tämä voidaan ratkaista tallentamalla attribuutin arvo *oppilaan* instanssiattribuutteihin, kuten ohjelmassa 25.

```
class Arvosana:

    def __get__(self, obj, type_=None):
        return obj._arvosana

    def __set__(self, obj, value):
        if value < 0 or value > 5:
            raise ValueError('arvosanan tulee olla väliltä 0-5')
        obj._arvosana = value
```

Ohjelma 25. Paranneltu, joskin yhä virheellinen `Arvosana`-deskriptori

Ongelmaksi kuitenkin jää usean deskriptorin käyttö saman luokan sisällä. Mikäli opiskelijalle lisätään toinen samanlainen attribuutti, `fysiikan_arvosana = Arvosana()`, niin molemmat deskriptorit yrittävät käyttää samaa opiskelijan `_arvosana` instanssiattribuuttia, jolloin ne korvaavat toistensa asettamat arvot.

Python tarjoaa ongelmaan ratkaisuksi `__set_name__`-metodin. Kun `type`-metaluokka luo uuden luokan, se etsii kyseisen luokan attribuuteista deskriptoreja, jotka sisältävät metodin `__set_name__`. Seuraavaksi `type` kutsuu löytyneitä metodeja, välittäen argumenteiksi käsiteltävän luokan, sekä sen attribuutin nimen, johon deskriptor on kyseisen luokan sisällä asetettu. [13] Ohjelma 26 havainnollistaa tätä toiminnallisuutta.

```
1  class OmaDeskriptori:
2
3      def __set_name__(self, owner, name):
4          print('Luokka:', owner, ', Attribuutti:', name)
5
6  class Esimerkki:
7      esimerkki = OmaDeskriptori()
8
9  # tulostus: Luokka: <class 'OmaDeskriptori'>, Attribuutti: esimerkki
```

Ohjelma 26. Esimerkki metodin `__set_name__` käytöstä

Nyt ohjelman 25 `Arvosana`-deskriptori voidaan toteuttaa tallentamaan arvosana oppilaan instanssiattribuutteihin, vastaavan attribuutin nimen taakse. Tämä on toteutettu ohjelmassa 27.

```
1  class Arvosana:
2
3      def __set_name__(self, owner, name):
4          self.attr = '_' + name
5
6      def __get__(self, obj, type_=None):
7          return getattr(obj, self.attr)
8
9      def __set__(self, obj, value):
10         if value < 0 or value > 5:
11             raise ValueError('arvosanan tulee olla väliltä 0-5')
12         setattr(obj, self.attr, value)
```

Ohjelma 27. Toimiva `Arvosana`-deskriptori

Koska attribuutin nimi vastaanotetaan ohjelmassa merkkijonona, deskriptorin täytyy käyttää `getattr` ja `setattr` -funktioita. Lisäksi attribuutin nimen eteen on helppo lisätä yksityistä attribuuttia kuvaava alaviiva. Ohjelmaa on pelkistetty esimerkin vuoksi, todellisessa ohjelmassa olisi hyvä olla muun muassa `__init__`-metodi, joka alustaisi `self.attr`-instanssiattribuutin.

7 Yhteenveto

Työssä perehdyttiin metaohjelmointiin, jonka avulla voidaan viedä abstraktioperiaate luokkia ja funktioita korkeammalle tasolle. Työssä esiteltiin tarkemmin Pythonin tärkeimpiä metaohjelmointityökaluja, joiden avulla voidaan ratkaista useita ohjelman rakenteeseen liittyviä ongelmia elegantisti. Toistuva koodi yksittäisissä luokissa tai funktioissa voidaan eliminoida dekoraattorien avulla, ja kokonaisista periytymishierarkioista saadaan toistuva koodi koottua yhteen metaluokkaan. Lisäksi deskriptoreilla voidaan toteuttaa elegantisti kaikkea toistuvista propertyistä aina räätälöityihini metodikutsuihin.

Jokainen työkalu on kuitenkin vain yhtä hyvä kuin käyttäjänsä, ja tästä syystä on tärkeä tutustua myös metaohjelmointityökalujen pohjalla toimiviin ominaisuksiin. Kun ohjelmoija ymmärtää miten työkalut toimivat, niitä on helpompi soveltaa erilaisissa tilanteissa. Tärkeimpiä työssä esiteltyjä konsepteja metaohjelmoinnin kannalta ovat Pythonin luokkien ja funktioiden ensimmäisen luokan kansalaisuus sekä kietoutujafunktiot. Näitä kielen ominaisuuksia käytetään jatkuvasti lähes kaikissa laajalti käytössä olevissa Pythonkirjastoissa.

Lopuksi on tärkeää muistaa, että koodia luetaan useammin kuin sitä kirjoitetaan. Liiallinen metaohjelmointi ja abstraktioperiaatteen ehdoton noudattaminen saattavat pahimmassa tapauksessa monimutkaistaa ohjelman toteutusta ja tehdä koodista lähes lukukelvotonta. Vaikka metaohjelmointi on erittäin hyödyllinen työkalu, johon jokaisen aktiivisesti Pythonia käyttävän ohjelmoijan tulisi perehtyä, tulisi sitä silti käyttää harkiten. Lopullinen vastuu ohjelman toteutuksesta ja luettavuudesta on ohjelmoijan itsensä harteilla.

8 Lähdeluettelo

[1] Beazley, David M. ja Brian K. Jones, ”Chapter 9. Metaprogramming,” tekijä: *Python Cookbook*, Third toim., Sebastopol, California: O’Reilly Media Inc., 2014.

[2] Pierce, Benjamin C, ”Polymorphism,” tekijä: *Types and Programming Languages*, The MIT Press, 2019, p. 339.

[3] R. Lämmel ja S. P. Jones, ”Scrap your boilerplate: a practical design pattern for generic programming,” tekijä: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, New Orleans, Louisiana, USA, 2003.

[4] Lilis, Yannis ja Anthony Savidis, ”A Survey of Metaprogramming Languages,” ACM computing surveys, 2020.

[5] Aaron C. Erickson, Ted Neward, Richard Minerich ja Talbott Crowell, ”13. Functions,” tekijä: *Professional F# 2.0*, Indianapolis, Wiley Publishing, 2010, pp. 207-203.

[6] David Beazley, ”Python 3 Metaprogramming,” tekijä: *PyCon US*, 2013.

[7] Beazley, David M. ja Brian K. Jones, ”Functions,” tekijä: *Python Cookbook*, Third toim., Sebastopol, California: O’Reilly Media Inc., 2014.

[8] Python Software Foundation, ”Datamodel, Python Reference Documentation,” 2020. [Online]. Available: https://docs.python.org/3/reference/datamodel.html. [Haettu 30 10 2020].

[9] P. S. Foundation, ”Built-in Functions,” 2020. [Online]. Available: https://docs.python.org/3/library/functions.html. [Haettu 31 10 2020].

[10] Lott, Steven F. ja Duraid Fatouhi, Mastering Object-Oriented Python: Grasp the Intricacies of Object-Oriented Programming in Python in Order to Efficiency Build Powerful Real-World Applications, Birmingham: Packt Publishing, 2015.

[11] Guido van Rossum, Barry Warsaw ja Nick Coghlan, ”PEP 8 Style Guide for Python Code,” 2020. [Online]. Available: https://www.python.org/dev/peps/pep-0008/. [Haettu 1 11 2020].

[12] Python Software Foundation, ”What's New In Python 3.6,” 2020. [Online]. Available: https://docs.python.org/3/whatsnew/3.6.html. [Haettu 30 10 2020].

[13] R. Hettinger, ”Descriptor HowTo Guide,” 2020. [Online]. Available: https://docs.python.org/3/howto/descriptor.html. [Haettu 30 10 2020].