Toisto ja modulaarisuus

Turun yliopisto
University of Turku

Päivän sisältö

- Ohjausrakenne: toisto
  - Määrätty toisto (Definiitti)
  - Epämääräinen toisto (Indefiniitti)
- Modulaarisuus

Toisto

Toisto?

- Mihin tarvitaan toistoa?

```
print ”2 * 1 = 2”
print ”2 * 2 = 4”
print ”2 * 3 = 6”
print ”2 * 4 = 8”
print ”2 * 5 = 10”
        .
        .
        .
```

Toisto

- Peräkkäisyys ja valinta eivät riitä algoritmien tarvittavan yleisyyden saavuttamiseksi
- Riittävä yleisyys saavutetaan ***toiston*** *(repetition)* eli ***iteraation*** avulla
  - Toistoa kutsutaan myös ***silmukaksi*** *(loop)* tai ***toistorakenteeksi***

Toistorakenteet

- Definiitti toisto
  - Toistokertojen määrä tunnetaan etukäteen
  - Toistokertojen määrä ei voi muuttua toistorakenteen suorituksen aikana
  - Joskus lyhennysmerkintä: korvaa “kopioi-liitä –menetelmän”
- Indefiniitti toisto
  - Dynaaminen toistorakenne, jossa toistokertojen määrä määräytyy toiston aikana
  - Voi määräytyä *toistoehdon* avulla niin, ettei määrää tiedetä ennen toiston suorituksen aloittamista
  - Lisää aidosti kielen ilmaisuvoimaisuutta
  - Huolehdittava siitä, että toisto päättyy joskus!

Definiitti toisto (“for”)

Definiitti toisto on muotoa

```
for x in {0,1,2,3,4}:
    toiminto
```

- Suoritetaan *toiminto* kerran kutakin listan *L* alkiota kohden
- Voidaan viitata vuorossa olevaan listan alkioon
  - Toiminto voi siis kohdistua eri objektiin eri toistokerroilla

Mikä “lista”? Esittelyssä: range

- Usein käydään läpi lukuja joltain väliltä
- Tähän voidaan käyttää **range**-funktiota
  - Funktiot käsitellään tarkemmin ensi viikolla
- Esimerkiksi

```
for i in range(N):
    toiminto
```

Tässä toiminto suoritetaan N kertaa siten, että

- Ensimmäisellä toistokerralla i:n arvo on 0, toisella kerralla 1,
  - Viimeisellä N-1, ei siis N!

Mikä “lista”? Esittelyssä: range

- Voidaan myös aloittaa jostain tietystä luvusta ja lopettaa toiseen lukuun

```
for i in range(M, N):
    toiminto
```

- Tässä toiminto suoritetaan aloittaen arvosta M ja lopettaen N-1:een

- Esim. luvut 1,2,3,…,8,9,10

range(1,11)

Mikä “lista”? Esittelyssä: range

- Jos alku- ja loppuarvot ovat samat, silmukkaa ei suoriteta ollenkaan:

```
for i in range(5,5):
    print “tätä ei suoriteta”
```

Askeltava toisto - esimerkki

```
summa = 0
for i in range(1,6):
  summa = summa + i
```

Muuttuja i saa siis arvot 1, 2, 3, 4, 5 ja suorittaa toiminnot kullakin i:n arvolla

Askeltava toisto - esimerkki

```
summa = 0
for i in range(1,6):
  summa = summa + i
```

Muuttuja i saa siis arvot 1, 2, 3, 4, 5 ja suorittaa toiminnot kullakin i:n arvolla

- 1. toisto, i = 1: sijoitetaan summaan arvo 0 + 1

Askeltava toisto - esimerkki

```
summa = 0
for i in range(1,6):
  summa = summa + i
```

Muuttuja i saa siis arvot 1, 2, 3, 4, 5 ja suorittaa toiminnot kullakin i:n arvolla

- 1. toisto, i = 1: sijoitetaan summaan arvo 0 + 1
- 2. toisto, i = 2: sijoitetaan summaan arvo 1 + 2

Askeltava toisto - esimerkki

```
summa = 0
for i in range(1,6):
  summa = summa + i
```

Muuttuja i saa siis arvot 1, 2, 3, 4, 5 ja suorittaa toiminnot kullakin i:n arvolla

- 1. toisto, i = 1: sijoitetaan summaan arvo 0 + 1
- 2. toisto, i = 2: sijoitetaan summaan arvo 1 + 2
- 3. toisto, i = 3: sijoitetaan summaan arvo 3 + 3

Askeltava toisto - esimerkki

```
summa = 0
for i in range(1,6):
  summa = summa + i
```

Muuttuja i saa siis arvot 1, 2, 3, 4, 5 ja suorittaa toiminnot kullakin i:n arvolla

- 1. toisto, i = 1: sijoitetaan summaan arvo 0 + 1
- 2. toisto, i = 2: sijoitetaan summaan arvo 1 + 2
- 3. toisto, i = 3: sijoitetaan summaan arvo 3 + 3
- 4. toisto, i = 4: sijoitetaan summaan arvo 6 + 4

Askeltava toisto - esimerkki

```
summa = 0
for i in range(1,6):
  summa = summa + i
```

Muuttuja i saa siis arvot 1, 2, 3, 4, 5 ja suorittaa toiminnot kullakin i:n arvolla

- 1. toisto, i = 1: sijoitetaan summaan arvo 0 + 1
- 2. toisto, i = 2: sijoitetaan summaan arvo 1 + 2
- 3. toisto, i = 3: sijoitetaan summaan arvo 3 + 3
- 4. toisto, i = 4: sijoitetaan summaan arvo 6 + 4
- 5. toisto, i = 5: sijoitetaan summaan arvo 10 + 5
- Silmukka suoritetaan 5 kertaa i:n arvoilla 1, 2, 3, 4 ja 5

Askeltava toisto - esimerkki

```
for i in range(1,11):
        print ”2 *”, i, ” = ”, (i * 2)
```

Tulostaa:

2 * 1 = 2

2 * 2 = 4

2 * 3 = 6

.

.

2 * 10 = 20

Askeltava definiitti toisto

- Jos askel poikkeaa yhdestä, se voidaan antaa **range**lle kolmanneksi parametriksi
- range(alkuarvo, loppuarvo, askel)
- Esimerkiksi

```
for i in range(1,10, 2):
        toiminto
```

Suorittaa toiminnon i:n arvoille 1,3,5,7,9

Edellinen esimerkki toisella tavalla

```
for i in range(2, 21, 2):
        print ”2 *”, (i/2), ”=”, i
```

Tulostaa:

2 * 1 = 2

2 * 2 = 4

2 * 3 = 6

.

.

2 * 10 = 20

Viime esimerkissä:

```
for i in range(1,11):
  print ”2 * ”, i, ”=”, (i * 2)
```

Askeltava definiitti toisto

- Askel voi olla myös negatiivinen
- Esimerkiksi

```
for i in range(10,0,-1):
    print i
```

- Nyt *i* saa arvot 10, 9, 8, …, 1
  - Loppuarvo 0 jää jälleen pois

Sisäkkäiset toistolauseet

- Kuten ehtolauseita, myös toistolauseet voivat sisältää toisia toistolauseita
- Esimerkiksi:

```
for i in range(5):
  for j in range(5):
    print i, j
```

Tulostaa 0 0 0 1 0 2 0 3 0 4 1 0 1 1 1 2 1 3 1 4 ….

(eri riveille)

- Sisemmän silmukan laskuri (eli j yllä) kannattaa nimetä eri tavalla kuin ulomman

Sisäkkäiset toistolauseet

- Sisempi toistolause voi käyttää myös ulomman laskurimuuttujaa
- Esimerkiksi:

```
for i in range(3):
    for j in range(i):
        print "sisempi silmukka"
```

Indefiniitti toisto (toistojen lukumäärää ei tiedetä)

```
while ehto:
    toiminto
```

- Silmukan *toiminnot* suoritetaan niin kauan kuin *ehto* on tosi
- Jos ehto heti epätosi, ei suoriteta kertaakaan

Indefiniitti toisto (toistojen lukumäärää ei tiedetä)

```
while ehto:
    toiminto
```

- *Toiminnon* suorituksen jälkeen *ehto* testataan jälleen
- Jos *ehto* edelleen voimassa, suoritetaan *toiminnot* uudestaan
- Toisto loppuu, kun *ehto* on epätosi

While-toisto - esimerkki

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
    i = i + 1
```

While-toisto - esimerkki

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
    i = i + 1
```

- i = 5 < 10 – suoritetaan silmukka; summa 5; i 6

While-toisto - esimerkki

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
    i = i + 1
```

- i = 5 < 10 – suoritetaan silmukka; summa 5; i 6
- i = 6 < 10 – suoritetaan silmukka; summa 11; i 7

While-toisto - esimerkki

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
    i = i + 1
```

- i = 5 < 10 – suoritetaan silmukka; summa 5; i 6
- i = 6 < 10 – suoritetaan silmukka; summa 11; i 7
- i = 7 < 10 – suoritetaan silmukka; summa 18; i 8

While-toisto - esimerkki

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
    i = i + 1
```

- i = 5 < 10 – suoritetaan silmukka; summa 5; i 6
- i = 6 < 10 – suoritetaan silmukka; summa 11; i 7
- i = 7 < 10 – suoritetaan silmukka; summa 18; i 8
- i = 8 < 10 – suoritetaan silmukka; summa 26; i 9

While-toisto - esimerkki

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
    i = i + 1
```

- i = 5 < 10 – suoritetaan silmukka; summa 5; i 6
- i = 6 < 10 – suoritetaan silmukka; summa 11; i 7
- i = 7 < 10 – suoritetaan silmukka; summa 18; i 8
- i = 8 < 10 – suoritetaan silmukka; summa 26; i 9
- i = 9 < 10 – suoritetaan silmukka; summa 34; i 10

While-toisto - esimerkki

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
    i = i + 1
```

- i = 5 < 10 – suoritetaan silmukka; summa 5; i 6
- i = 6 < 10 – suoritetaan silmukka; summa 11; i 7
- i = 7 < 10 – suoritetaan silmukka; summa 18; i 8
- i = 8 < 10 – suoritetaan silmukka; summa 26; i 9
- i = 9 < 10 – suoritetaan silmukka; summa 34; i 10
- i = 10 ei ole < 10, lopetataan

While-toisto – ehdon muotoilu

- **Mikä tässä on vikana?**

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
```

While-toisto– ehdon muotoilu

- Toiston rungossa on tärkeä tehdä operaatioita jotka lopulta muuttavat ehdon epätodeksi
- Ehdon muotoon kiinnitettävä huomiota
- Yleinen virhe: ehto joka pysyy totena
- ViLLEssä tällaiset virheet ilmenevät aikakatkaisuvirheenä (Operation timed out)

```
i = 5
summa = 0
while i < 10:
    summa = summa + i
```

While-toisto – ehdon muotoilu

- Esimerkki:

```
i = 1
summa = 0
while i != 10:
    summa = summa + i
    i = i + 2
```

- Nyt i käy läpi parittomia arvoja 1, 3, 5, 7, 9, 11
- Ehto ei siis ikinä tule epätodeksi
- Korjaus: ehto muotoon i < 10

Indefiniitti vs definiitti toisto

- Definiitti toisto voidaan aina korvata indefiniitillä toistolla
  - Päinvastoin ei päde!
- **Indefiniitti toisto riittäisi kaikkien toistojen esittämiseen**
- Selvyyden vuoksi valitaan ongelmaan parhaiten sopiva toistomuoto
  - Jos tarvitaan silmukkamuuttujaa, definiitti yleensä soveltuu paremmin

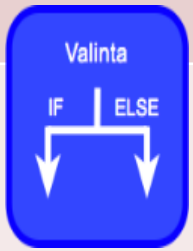
Indefiniitti vs definiitti toisto

- Definiitti toiston korvaus indefiniitillä toistolla esimerkki:

Definiitti toisto	Indefiniitti toisto

summa = 0
for i in range(1,6):
    summa = summa + i
	summa = 0
	i = 1
	while i < 6:
	    summa = summa + i
	    i = i + 1

Yhteenveto kontrollirakenteista

Ohjausrakenne	Selitys
	• Iteratiivisten ohjelmien perusrakenne • Koodirivit suoritetaan ylhäältä alas algoritmiin kirjoitetussa järjestyksessä • Tästä järjestyksestä voidaan poiketa muilla rakenteilla
	• Kontrollirakenne jolla valitaan kahden toiminnon välillä • Ehdon ollessa tosi suoritetaan if-lohkon koodi • Ehdon ollessa epätosi suoritetaan else-lohkon koodi
	• Mahdollistaa toiminnon suorittamisen monta kertaa • Eri tyyppejä sen mukaan tiedetäänkö toistojen määrä • Rakenteet for, while

Modulaarisuus

Abstraktiot

- Ihminen hankkii tietämystä kehittämällä asiasta ensin epätarkan käsitteellisen mallin
- Malli tarkentuu (ja korjaantuu) tietämyksen karttuessa
- Luotua mallia kutsutaan ***abstraktioksi***
- Mallin luomisen henkistä prosessia ***abstrahoinniksi***
- Epätarkatkin mallit tärkeitä, ne ohjaavat uuden tietämyksen hankinnassa
- Abstraktin ajattelun kyky on erilainen eri ihmisillä
  - Abstraktia ajattelua voi kuitenkin kehittää!

Abstraktiot

- Abstraktion tulee olla riittävä, muttei välttämättä täydellinen
- Malli saa olla puutteellinen
- Kaikki merkitykselliset yksityiskohdat pystyttävä kuvaamaan mallilla
- On hyvä, jos irrelevantit yksityiskohdat on karsittu mallista pois
- Merkitykselliset yksityiskohdat riippuvat näkökulmasta
- Esimerkiksi yrityksen työntekijää (siis ihmistä) voitaisiin kuvata mallilla jossa on
  - Nimi
  - Virka-asema
  - Palkkaluokka
  - Ikä
  - jne..

Abstraktiotaso

- ***Abstraktiotaso*** on tärkeä abstraktion ominaisuus
  - Mitkä yksityiskohdat on piilotettu mallin sisälle? Eli mallin sisäinen esitys.
  - Mitkä yksityiskohdat näkyvät ulospäin? Eli mallin ulkoinen esitys.
- Ulkomaailman kannalta liittymät malliin muodostavat abstraktion
- Liittymien määrittely tärkeä osa abstrahointia
- Olellista, että ulkomaailman ei tarvitse tietää mallin sisäisiä yksityiskohtia, ulkoinen esitys riittää

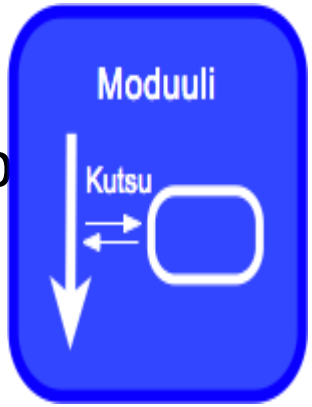
Algoritmien reduktiivinen suunnittelu

- Asteittain tarkentava suunnittelumenetelmä
- Aloitetaan algoritmin abstraktista kuvauksesta
- Tarkennusaskeleissa algoritmi jaetaan pienempiin ja tarkempiin komponentteihin, *alialgoritmeihin*
- Alialgoritmeja voidaan edelleen tarkentaa, kunnes algoritmi on jaettu ***triviaaleihin*** komponentteihin
  - Eli siten että prosessori osaa ne suorittaa

Algoritmien reduktiivinen suunnittelu

- Edellä kuvattua ongelmaratkaisuperiaatetta kutsutaan ***reduktiiviseksi***
  - Jaetaan ongelma osiin
  - Ratkaistaan osaongelmat
  - Koostetaan alkuperäisen ongelman ratkaisu osaongelmien ratkaisuista
- Kokonaisuuden osittamisessa tehdyt rajauspäätökset tärkeitä
- Pyrittävä löytämään luonteva ja hyvin määritelty jako osaongelmiin

Moduuli



- ***Moduuli*** on itsenäinen algoritmikomp
  - Moduuli eli metodi (method)
    eli rutiini (routine)
    eli alirutiini (subroutine)
    eli aliohjelma (subprogram)
- Moduuli ratkaisee yhden osaongelman
- Moduuli voidaan suunnitella käyttöyhteydestä riippumattomasti
- Moduulia voidaan käyttää missä tahansa algoritmissa jossa esiintyy ko. Osaongelma

Moduulin kutsu ja modulaarisuus

- Moduulia käyttävän algoritmin sanotaan ***kutsuvan*** moduulia
- Useista moduuleista koostuvaa algoritmia sanotaan ***modulaariseksi***
- Modulaarisessa algoritmissa jokainen ei-triviaali moduuli koostuu pienemmistä moduuleista

Modulaarisuus ja modulaarisuusperiaate

- Modulaarisuusperiaate:
  - Yleinen suunnitteluperiaate algoritmien suunnitteluun
    - Myös muuhun tietojenkäsittelyalaan, tietokoneen rakenteen ja toiminnan suunnitteluun
    - Myös yleiseen ongelmanratkaisuun

Moduulin määrittely

Yksinkertaisen moduulin määrittely on muotoa:

```
def moduulinNimi():
    moduulin runko
```

- *moduulinNimi* on moduulille annettu nimi
  - Nimen tulee kuvata moduulin tehtävää
  - Käytetään moduulia kutsuttaessa
- Moduulin runko on tehtävän toteuttavat toiminnot

Moduulin kutsu

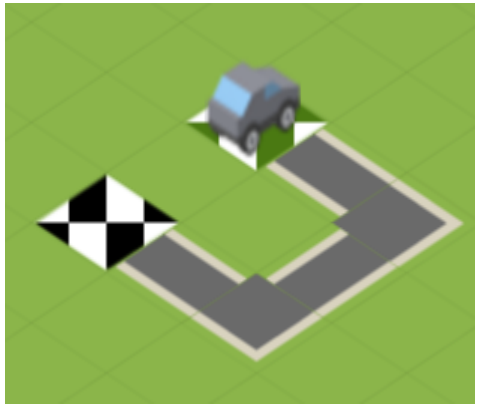
```
def moduulinNimi():
    moduulin runko
```

- Moduulia kutsutaan sen nimeä käyttäen
- Esimerkiksi yo. moduulin kutsu:

  moduulinNimi()
- Monimutkaisempia määrittelyjä ja kutsuja ensi viikolla

Moduulin käyttö - esimerkki

- Tehtävä: kuljeta auto maaliin teitä pitkin

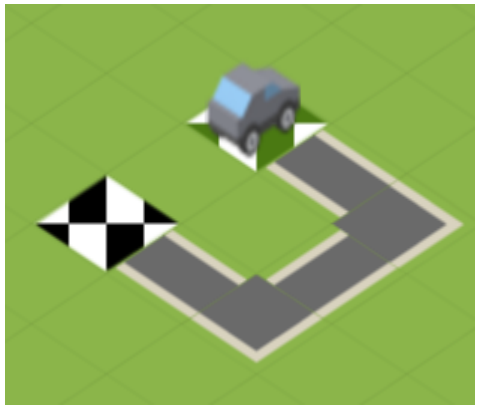


- Käytössä moduulit turnRight(), turnLeft(), moveForward(), moveBackward()

Moduulin käyttö - esimerkki

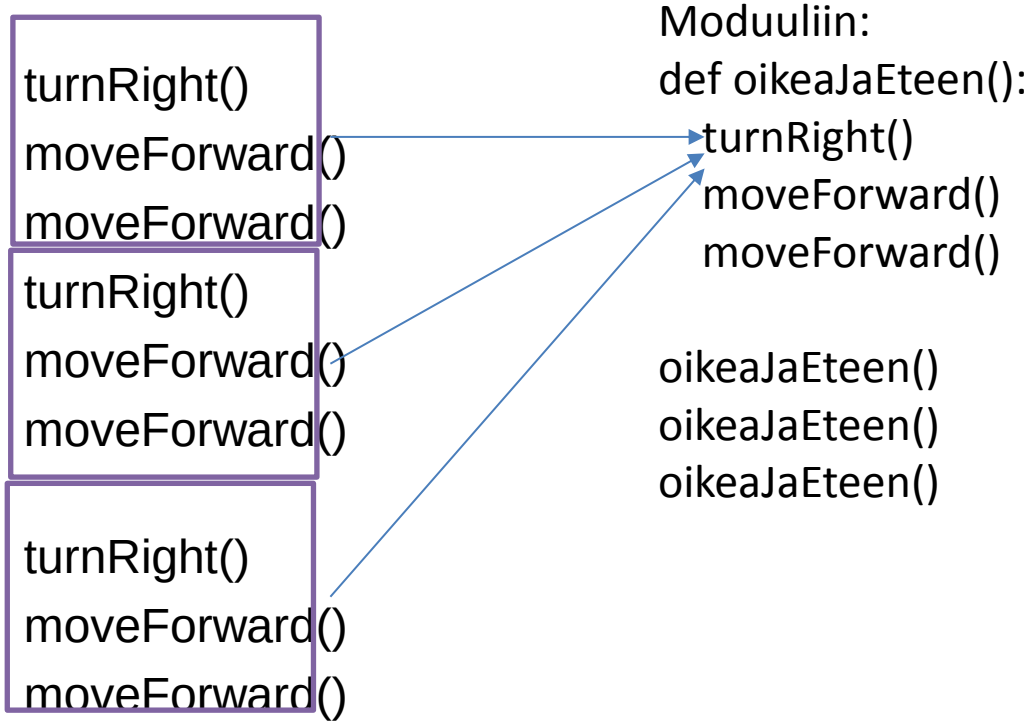
- Käytössä moduulit turnRight(), turnLeft(), moveForward(), moveBackward()
- Naiivi tapa:

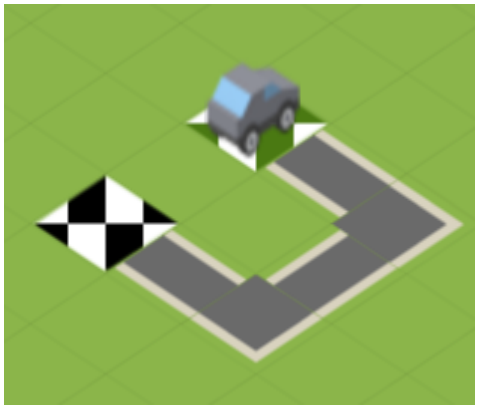
```
turnRight()
moveForward()
moveForward()
turnRight()
moveForward()
moveForward()
turnRight()
moveForward()
moveForward()
```



Moduulin käyttö - esimerkki

Käytössä moduulit turnRight(), turnLeft(), moveForward(), moveBackward()
Naiivi tapa:



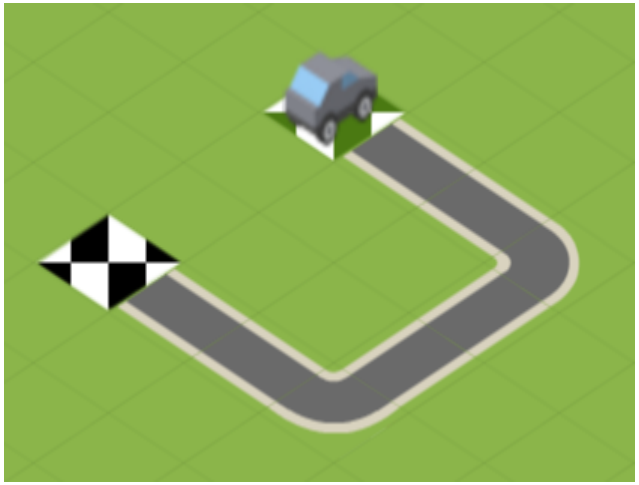


Moduulin käyttö - esimerkki

- Miten algoritmi voidaan korjata kuvan uuteen tilanteeseen?

```
# Ei toimi!
def oikeaJaEteen():
    turnRight()
    moveForward()
    moveForward()

oikeaJaEteen()
oikeaJaEteen()
oikeaJaEteen()
```

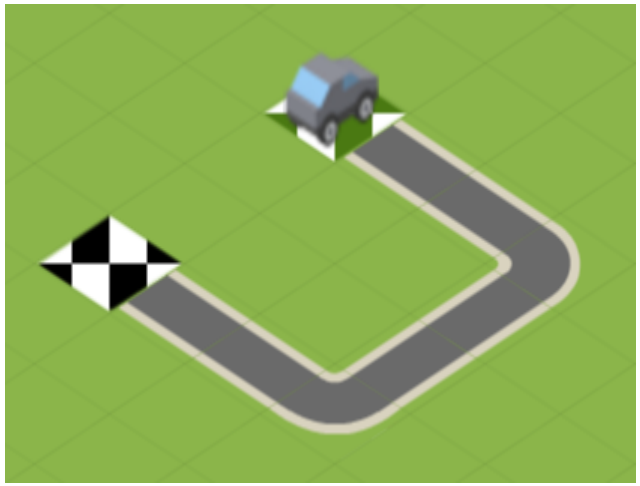


Moduulin käyttö - esimerkki

- V: korjataan vain moduulia

```
# Toimii!
def oikeaJaEteen():
    turnRight()
    moveForward()
    moveForward()
    moveForward()

oikeaJaEteen()
oikeaJaEteen()
oikeaJaEteen()
```

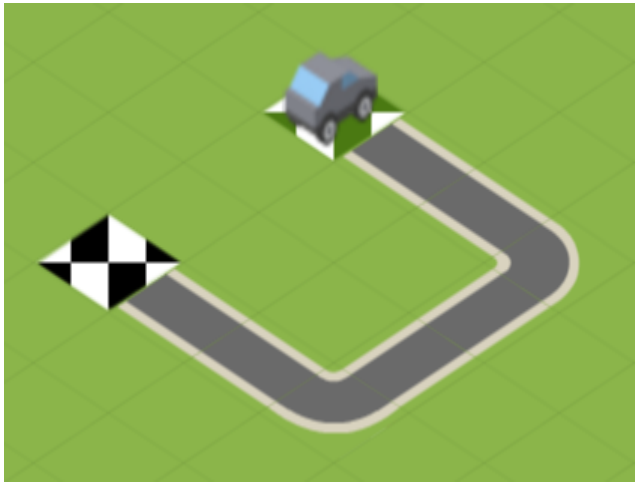


Moduulin käyttö - esimerkki

- V: Vähennetään vielä turhaa ”copy-pastea” käyttämällä silmukoita

```
# Toimii!
def oikeaJaEteen():
    turnRight()
    for i in range(3):
        moveForward()

for i in range(3):
    oikeaJaEteen()
```



Moduulien suunnittelu

- Algoritmin astettaisessa tarkentamisessa luodaan moduuleja tietyllä abstraktiotasolla
  - Tarkennettaessa algoritmin abstraktiotaso laskee
  - Ja yksityiskohtien määrä lisääntyy
- Yksityiskohtien lisääntyessä algoritmi tulee jakaa pienempiin komponentteihin
- Moduulien lukumäärän ja yksittäisen moduulin koon välillä löydettävä kompromissi

Moduulien suunnittelu

- Yksittäisen moduulin koko:
  - Voi vaihdella tehtävän mukaan
  - Voi vaihdella algoritmin suunnittelijan mieltymysten mukaan
- Tarkasteltavaa osakokonaisuutta pitäisi pystyä tarkastelemaan kerralla
  - Mahtuu näytölle kokonaisuudessaan
- Aloittelijat tekevät yleensä liian monimutkaisia moduuleja

- Hyvän moduulin ei tarvitse sisältää kuin muutama rivi
  - Esim. yksi valinta- tai toistorakennekokonaisuus
  - Toiston runkokin voi koostua alimoduuleista monimutkaisen rungon sijaan

Moduulin suunnittelu

- Jos samat rivit koodia toistuvat melkein samanlaisena:
  - Lisää ne moduuliin!

Moduulien suunnittelu

- Alimoduulit suunniteltava mahdollisimman riippumattomiksi muusta algoritmista ja alimoduuleista:
  - Voidaan suunnitella erikseen käyttöyhteydestä riippumatta
  - Riittää tuntea tehtävä, jonka moduuli ratkaisee
  - Mahdollistaa algoritmikokonaisuuden hajauttamisen:
    - Eri aikoina tehtäväksi
    - Eri ihmisten tekemäksi
  - Komponentteja voidaan myös käyttää useissa eri algoritmeissa

Moduulien suunnittelu

- Suunnittelussa huomioitavaa:
  - **Tehtäväkohtaisuus**: yksi moduuli ratkaisee yhden tehtävän
  - **Luonnollisuus**: luonnollisten osatehtävien erotus algoritmista lisää selkeyttä ja parantaa yleistä käyttökelpoisuutta
  - **Sopiva abstraktiotaso**: moduuli tulee rakentaa alimoduuleista, jotka ovat samalla abstraktiotasolla
- Aloittelijan ohjenuora: **pidä kiinni tehtäväkohtaisuudesta**
  - Vältä monimutkaisten ja useita ongelmia ratkaisevien alimoduulien tekemistä
  - Kahden tehtävän tekeminen moduulissa kostautuu myöhemmin
  - Tehtäväkohtaisuudesta voi luopua ainoastaan esim. tehokkuusvaatimusten vuoksi (ei siis tällä kurssilla, eikä käytännössäkään juuri koskaan)

Modulaarisuuden edut

- Tärkeimmät edut:
  - ***Selkeys*** -> luotettavuus ja helpompi ylläpito
  - ***Yleiskäyttöisyys*** eli ***uudelleenkäytettävyys*** (*reusability*)
- Yleiskäyttöisyys moduulin ja kokonaisuuden kannalta:
  - **Siirrettävyys**: moduulia voidaan käyttää paitsi suunnitellussa yhteydessä, myös missä tahansa missä osaongelma esiintyy
  - **Korvattavuus**: moduuli voidaan korvata toisella saman osaongelman ratkaisevalla moduulilla ilman kokonaisuuden muuttumista
    - Esim tehokkaampi ratkaisu ongelmaan
- Etujen saavuttaminen vaatii moduulien tarkkaa ja huolellista tehtävänmäärittelyä!