OHJELMOINNILLISEN AJATTELUN TYÖKALUT YLÄKOULUSSA

Turun yliopiston oppimisanalytiikan keskuksen järjestämän täydennyskoulutuskurssin sisällön kuvaus

TURUN YLIOPISTO
Oppimisanalytiikan keskus

TURUN YLIOPISTO
Oppimisanalytiikan keskus

OHJELMOINNILLISEN AJATTELUN TYÖKALUT YLÄKOULUSSA

TURUN YLIOPISTON OPPIMISANALYTIIKAN KESKUKSEN JÄRJESTÄMÄN TÄYDENNYSKOULUTUSKURSSIN SISÄLLÖN KUVAUS

Peter Larsson

Heidi Kaarto

Marika Parviainen

Oppimisanalytiikan keskus

oppimisanalytiikka.fi

TURUN YLIOPISTO
Oppimisanalytiikan keskus

SISÄLLYS

1 JOHDANTO

Tämä on kuvaus Turun yliopiston Oppimisanalytiikan keskuksen toteuttaman ja Opetushallituksen rahoittaman Ohjelmoinnillisen ajattelun työkalut perusopetuksessa -projektin toisesta täydennyskoulutuksesta. Projektissa suunniteltiin ja toteutettiin kaksi täydennyskoulutuskurssia: toinen alakoulun ja toinen yläkoulun opettajille. Tämä on kuvaus yläkoulun kurssista, joka tarjoaa tarvittavat pohjatiedot ohjelmoinnin opettamiseen osana yläkoulun matematiikkaa.

Ohjelmoinnin opetus voi tuntua haasteelta ja vaikka erilaista materiaalia on tarjolla, on vaikeaa tehdä pedagogisesti perusteltuja valintoja. Tämän kurssin tavoitteena oli antaa opettajalle perustiedot ohjelmoinnista ja valmiudet matematiikkaa soveltavien ohjelmoinnin materiaalien valitsemiseen ja luomiseen. Kurssilla huomioitiin erityisesti opetussuunnitelman määrittelemät algoritmisen ajattelun sekä ongelmanratkaisun tavoitteet. Ohjelmoinnillinen ajattelu antaa opettajalle työkalut ohjelmoinnin oppimisen ymmärtämiseen ja yläkoulun tavoitteiden saavuttamiseen.

1.1 KURSSIN KUVAUS

Kurssi koostuu neljästä kokonaisuudesta:

1. Ohjelmoinnin käsitteet
2. Algoritminen ajattelu ja ongelmanratkaisu
3. Kehittyneemmät ohjelmoinnin käsitteet
4. Ohjelmoinnillisen ajattelun soveltaminen matematiikassa

Jokainen kokonaisuus sisälsi verkkosisällön ja työpajan. Lisäksi kurssilla oli aloitusluento ja lopuksi opetuksen suunnitteluun liittyvä projekti.

Kurssi oli kahden opintopisteen laajuinen. Työmäärä jakautui seuraavasti:

	Tunnit
Verkkosisällöt	25
Työpajat	16
Kotitehtävät	8
Projekti	5
Yhteensä	**54**

Verkkosisällöt olivat teoriamateriaalia ja tehtäviä sisältäviä tutoriaaleja sähköisessä ViLLE-oppimisympäristössä. Työpajat järjestettiin etätapaamisina Zoom-palvelussa ja osallistujilta edellytettiin paikalla oloa. Lisäksi aloitusluento oli seurattavissa sekä etäluentona Zoomissa että tallenteena ViLLEssä. Kotitehtävät ja projektit tehtiin ViLLEn ulkopuolella, mutta palautettiin ViLLEen.

Kurssin suorittamiseksi hyväksytysti osallistujilta vaadittiin 75 % prosenttia ViLLEn pisteistä ja loppuprojektin palauttaminen. ViLLEn pisteet muodostuvat verkkosisältöjen tehtävistä, kotitehtävistä, loppuprojektista ja kyselyistä saaduista pisteistä.

1.2 KURSSIN TEKIJÄT

Kuva 1: Peter Larsson, Heidi Kaarto ja Marika Parviainen

Peter Larsson toimii tietotekniikan laitoksella opettajana ja tutkijana. Opetus koskee digitaalisia teknologioita ja yhteiskunnan digitalisaatiota. Tutkimus puolestaan keskittyy ohjelmoinnin ja ohjelmoinnillisen ajattelun opettamiseen peruskoulussa. Keskeisenä kysymyksenä on miten yhdistää ohjelmointi matematiikan opetukseen, niin että kumpikin aihe hyötyy. Ohjelmoinnin opetus peruskoulussa on vielä uutta, joten on tärkeää kehittää opetuksen sisältöjä ja menetelmiä. Peterin vapaa-aikaan kuuluu luonnossa liikkuminen, uinti, lauta- ja tietokonepelit sekä lukeminen.

Heidi Kaarto on suunnitellut ja toteuttanut useita peruskoulun ja lukion ohjelmointikursseja ja opettanut ohjelmointia lukiossa ja yliopistossa. Hän on aina halunnut olla opettaja ja lemmikinomistaja.

Marika Parviainen on IT-laitoksen kasvatti ja pitkän linjan villeläinen, jolla on kokemusta ohjelmointitehtävien laatimisesta, koulutusten järjestämisestä sekä käyttäjätuesta.

Kuva 2: Rahoittajana toimi Opetushallitus, oppimisympäristönä ViLLE, toteuttajana Turun yliopiston Oppimisanalytiikan keskus ja yhteistyökumppanina Vilnan yliopisto

2 KURSSILLA KÄYTETTÄVÄ OHJELMOINTIKIELI: PYTHON

Kurssilla käytämme ohjelmointikieltä nimeltä Python. Se on suosituin opetuksessa käytettävä tekstipohjainen ohjelmointikieli, sillä se on rakenteeltaan selkeä ja jo peruskomennoilla voi laatia kokonaisia ohjelmia. Toisaalta se on suosittu myös ammattikäytössä ja tarjoaa edistyneemmille käyttäjille kaikki nykyaikaisen ohjelmointikielen piirteet, kuten monipuoliset tietorakenteet ja olio-ohjelmoinnin. Python-kielellä on kirjoitettu runsaasti ohjelmakirjastoja, joilla saa käyttöön uusia ominaisuuksia, esimerkiksi data-analytiikkaan ja tekoälyyn liittyen. Google, Nasa, Facebook, kuten monet muutkin suuret it-alan toimijat käyttävät muiden ohjelmointikielien ohella myös Pythonia. Poimintoja liike-elämän ja tiedemaailman onnistumisista, joissa Python-kieli on ollut merkittävänä tekijänä, löydät osoitteesta https://www.python.org/success-stories/.

Kuva 3: Kurssilla käytetään ohjelmointikielenä Pythonia

Käytämme kurssilla Pythonin versiota 3. Se eroaa syntaksiltaan (kieliopiltaan) jonkin verran aiemmasta versiosta 2.

Hauska tietää: Nimensä Python-kieli on saanut brittiläiseltä komediasarjalta Monty Pythonilta.

Pythonin omaan IDLE-ohjelmointiympäristöön kuuluu ohjelman suorittamiseen vaadittava Python-komentotulkki ja tekstieditori, jossa värikorostukset helpottavat oikeaoppisesti kirjoitetun Python-koodin lukemista. Suosittelemme asentamaan koneelle Python-ohjelmointiympäristön IDLEn. On myös paljon ilmaisia, selainpohjaisia ohjelmointiympäristöjä, joiden etuna on se, että ei tarvitse asentaa erillistä ohjelmistoa.

Tulkattava ohjelmointikieli

Tietokoneen prosessori suorittaa käskyjä saamansa binäärikoodin mukaisesti. Tietokoneen ymmärtämää koodia sanotaan konekieleksi. Tietokoneohjelmat ovat yleensä valmiiksi konekielelle käännettyjä tiedostoja.

Käännettävällä ohjelmointikielellä laadittu ohjelma pitää kääntää konekielelle ennen ohjelman suorittamista. Osa ohjelmointikielistä on tulkattavia ohjelmointikieliä, jotka tulkki kääntää konekielelle käsky kerrallaan. Python on tulkattava ohjelmointikieli.

Asenna koneellesi Python ja IDLE (asentuu automaattisesti ohjelmointikielen kanssa) sivulta https://www.python.org/downloads.

Kuva 4: Python-ohjelmointikielen lataussivu

Jos et voi asentaa koneelle Pythonia ja IDLEä, suosittelemme tutustumaan seuraaviin maksuttomiin, selaimessa toimiviin ohjelmointiympäristöihin:

- OneCompiler: voi käyttää suoraan ilman kirjautumista, mainos ennen ohjelman suorittamista, koodin voi jakaa linkin avulla (https://onecompiler.com/python)
- repl.it: vaatii kirjautumisen, monipuoliset työryhmätoiminnot, maksullisessa versiossa opettajanäkymä (https://replit.com/languages/python3)
- trinket: vaatii opettajalta kirjautumisen, sen jälkeen voi jakaa oppilaille testattavia koodipätkiä linkin avulla, erikseen laajempi maksullinen opettajanäkymä (https://trinket.io/python3)

Voit kirjoittaa Python-käskyjä IDLE-ohjelmointiympäristössä suoraan komentotulkkiin >>> kehotteen jälkeen tai voit luoda IDLEN editorilla tiedoston, johon käskyt tallennetaan. Tiedoston sisältämät käskyt suoritetaan valitsemalla editorin valikkoriviltä Run. Ohjelman tulosteet näkyvät komentotulkin ikkunassa.

3 VERKKOSISÄLTÖ 1: OHJELMOINNIN KÄSITTEET

Kurssin ensimmäisen aihekokonaisuuden aiheena ovat ohjelmoinnin perusteet. Kokonaisuudessa käydään läpi ohjelmoinnin peruskäsitteitä ja -rakenteita, joita sovelletaan ohjelmointiin Pythonilla. Käsitteet ja rakenteet ovat samat kaikissa imperatiivisissa ohjelmointikielissä (yleisin ohjelmointikielten tyyppi), joten tässä opittua voit soveltaa myös muissa saman tyyppisissä kielissä.

3.1 OHJELMOINTI

Ohjelmoinnin tarkoitus on ohjata tietokonetta ohjelmien avulla. Ohjelmia luodaan ohjelmoimalla eli kirjoittamalla ohjelmakoodi, joka koostuu yksittäisistä ohjelmointiohjeista eli **lauseista** *(statements)*. Käytettävissä olevat lauseet on määritelty ohjelmointikielessä. Pythonissa jokainen lause kirjoitetaan omalle rivilleen.

Esimerkki:

```
# Tässä ohjelmassa on 5 lausetta.
luku = int(input("Luku:"))
if luku < 0:
    print("Luku on negatiivinen.")
else:
    print("Luku ei ole negatiivinen.")
```

Ohjelmakoodi ei sellaisenaan tee mitään, vaan se täytyy ajaa eli **suorittaa** *(run)*. Suorituksessa kone tekee ohjelmassa määritellyt toimenpiteet. Jotta ohjelman voi suorittaa, on ohjelma käännettävä tietokoneen ymmärtämään muotoon eli konekielelle. Ohjelman voi kääntää konekielelle kahdella tapaa:

1. Ohjelmakoodi käännetään konekielelle kokonaan ennen ohjelman suoritusta. Tällöin puhutaan kääntäjistä.
2. Ohjelmakoodia käännetään konekielelle lause kerrallaan juuri ennen kuin lause suoritetaan. Tällöin puhutaan tulkeista.

Python on tulkattava ohjelmointikieli eli Python-tulkki tulkkaa yhden lauseen kerrallaan juuri ennen kuin se suoritetaan.

Ohjelmat perustuvat **peräkkäisyyteen** *(sequence)*, mikä tarkoittaa sitä, että kone suorittaa ohjelman lauseet siinä järjestyksessä kuin ne on ohjelmaan kirjoitettu yksi lause kerrallaan.

Hyvä ohjelmointikäytäntö!

Pythonissa jokainen lause kirjoitetaan omalle rivilleen ja näin erotellaan lauseet toisistaan. Joissain kielissä lauseet kuitenkin erotetaan merkeillä (esim. puolipisteellä ;), jolloin lauseita voi kirjoittaa useita samalle riville. Hyvän ohjelmointikäytännön mukaista on kuitenkin kirjoittaa jokainen lause omalle rivilleen käytetystä ohjelmointikielestä riippumatta.

Ohjelmointikielillä on omat syntaksinsa eli kielioppinsa. **Syntaksi** *(syntax)* määrittelee, millaiset ohjeet ovat mahdollisia ja "miltä koodin pitäisi näyttää" samaan tapaan kuin luonnollisten kielten kieliopit. Syntaksivirheet aiheuttavat ohjelman suorituksen keskeytyksen.

Semantiikalla *(semantics)* tarkoitetaan lauseiden toimintaa ja tarkoitusta. Semanttiset virheet eivät aiheuta suorituksen keskeytymistä, vaikka ne voivat johtaa siihen, ettei ohjelma toimi, niin kuin sen pitäisi.

<u>Esimerkki:</u>

"Aurinko lämmittää Villeä." on sekä syntaktisesti että semanttisesti oikein, koska se ei sisällä kielioppivirheitä ja sen merkitys on todenmukainen. Sen sijaan lause "Ville lämmittää aurinkoa." on syntaktisesti oikein, mutta semanttisesti väärin, koska Ville ei pysty lämmittämään aurinkoa.

Huomaa, että Pythonissa syntaksivirheet ilmenevät kesken ohjelman suorituksen, koska Python on tulkattava ohjelmointikieli. Käännettävissä ohjelmointikielissä kaikki syntaksivirheet on korjattava ennen ensimmäisenkään lauseen suoritusta, mutta tulkattavissa virhettä edeltävät lauseet suoritetaan ennen virheilmoitusta. Virhe lopettaa ohjelman suorittamisen.

Ohjelmiin voi kirjoittaa myös kommentteja, jos koodia halutaan selventää. Kommentteja ei koskaan suoriteta, vaikka niissä olisi koodia.

Pythonissa kommentit aloitetaan #-merkillä. Kommentin voi joko aloittaa rivin alusta (jolloin koko rivi on kommenttia) tai kirjoittaa rivin loppuun.

```
# Tässä esimerkki koko rivin kommentista.
nimi = Python
print(nimi) # Esimerkki rivin lopussa olevasta kommentista.
```

Hyvä ohjelmointikäytäntö!

Ohjelmakoodin kommentointi on hyvän ohjelmointikäytännön mukaista. Kommenttien avulla ohjelmakoodin lukeminen on helpompaa ja huolellisten kommenttien avulla koodin tarkoitus/tehtävä selviää hetkessä (vrt. itse koodin lukemiseen). Etenkin pidemmissä ohjelmissa ja projekteissa, joissa samaa ohjelmaa saattaa ohjelmoida useampi ihminen, on kommentointi todella tärkeää.

3.2 LAUSEKKEET JA OPERAATTORIT

Tietokone tallentaa kaiken tiedon vain ykkösten ja nollien avulla. Siksi on oleellista määritellä, minkä tyyppistä tieto on, jotta tietoa käsiteltäessä ykkösten ja nollien sarjat tulkitaan ja näytetään oikein. Sama ykkösten ja nollien sarja antaa eri tulokset, jos se tulkitaan lukuna tai tekstinä.

Siksi myös ohjelmoinnissa käytetyillä arvoilla on aina **tyyppi** *(type)*. Perustyyppejä ovat kokonaisluvut, liukuluvut, merkkijonot ja totuusarvot. **Liukuluvut ovat tietokoneen (sisäinen) tapa esittää desimaalilukuja.** Pythonissa desimaalierotin on piste eikä pilkku. Merkkijonot erotetaan lainausmerkein ja niihin voi tallentaa monia erilaisia merkkejä (isoja ja pieniä kirjaimia, numeroita, erikoismerkkejä).

<u>Esimerkkejä tyypeistä:</u>

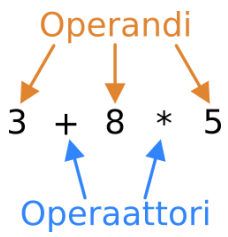
kokonaisluku	`-6`
	`0`
	`92`
	`9834`

liukuluku	`-6.3`
	`5.3498`
	`984.52`

merkkijono	`"jono"`
	`"Terve!"`
	`"Ol1pa kerran m3rkki_jonoja."`

totuusarvot	`True`
	`False`
	
	(ei muita!)

Lauseke *(expression)* voi olla yksittäinen arvo (mitä tahansa tyyppiä) tai **operaattoreiden** ja **operandien** yhdistelmä.



Kuva 5: Lausekkeet voivat olla yksittäisiä arvoja tai operandien ja operaattorien yhdistelmiä.

Matemaattisten operaattoreiden laskujärjestys on Pythonissa sama kuin matematiikassa yleensä. Jakojäännösoperaattori rinnastetaan kerto- ja jakolaskun kanssa. *Sulkeita () voidaan käyttää laskujärjestyksen muuttamiseen kuten matematiikassakin.*

<u>Matemaattiset operaattorit:</u>

Operaatio	**Operaattori**	**Esimerkki**	**Esimerkin tulos**
yhteenlasku	`+`	`5 + 18`	`23`
vähennyslasku	`-`	`5 - 6`	`-1`
kertolasku	`*`	`5 * 4`	`20`
jakolasku	`/`	`12 / 3`	`4.0`
potenssilasku	`**`	`2 ** 3`	`8`
jakojäännös (modulo)	`%`	`15 % 4`	`3`

Koska arvoilla on tyypit, on niillä merkitystä myös lausekkeiden laskemisessa. Esimerkiksi luvut, jotka on kirjoitettu merkkijonona (eli lainausmerkkien sisään), ei yhteenlaskussa lasketakaan lukuina yhteen vaan muodostetaan uusi merkkijono, jossa luvut on kirjoitettu peräkkäin. Siis `"53"+"23"="5323"`.

Pythonissa liukuluvun sisältävän lausekkeen lopputulos on aina liukuluku eli esim. `5+5.0=10.0`. **Lisäksi jakolaskun lopputulos on aina liukuluku, vaikka kahden kokonaisluvun jakolasku menisikin tasan** eli `6/4=1.5` ja `6/3=2.0`. (Näin ei siis ole kaikissa kielissä!)

Vertailuoperaattorit:

Operaatio	Operaattori	Esimerkki	Tulos
yhtä suuri kuin	`==`	`6 == 4`	`False`
ei yhtä suuri kuin	`!=`	`6 != 4`	`True`
pienempi kuin	`<`	`6 < 4`	`False`
suurempi kuin	`>`	`6 > 4`	`True`
pienempi tai yhtä suuri kuin	`<=`	`6 <= 4`	`False`
suurempi tai yhtä suuri kuin	`>=`	`6 >= 4`	`True`

Loogiset operaattorit:

Operaatio	Operaattori	Esimerkki	Tulos
ja	`and`	`True and False`	`False`
tai	`or`	`True or False`	`True`
ei (negaatio)	`not`	`not True`	`False`

Lausekkeiden arvoa laskettaessa lasketaan ensin matemaattiset operaatiot, sitten vertailuoperaattoreiden arvot ja vasta viimeiseksi loogisten operaattoreiden arvot. Vertailu- tai loogisia operaattoreita sisältävän lausekkeen tulos on aina totuusarvo eli `True` tai `False`. Niitä kutsutaan **ehtolausekkeiksi**.

3.3 TULOSTAMINEN RUUDULLE

Ohjelmoinnissa käytetään usein aliohjelmia toteuttamaan monenlaisia toimintoja. Ohjelmointikielestä riippuen niitä kutsutaan pääasiassa funktioiksi *(function)* tai metodeiksi *(method)*. Pythonissa puhutaan funktioista. **Funktioita** on ohjelmointikielessä valmiina, mutta niitä voi myös määritellä itse (tähän palataan myöhemmin). Funktioiden toimintaa voidaan ohjata **parametreilla** *(parameters)* ja ne voivat palauttaa jonkin arvon siihen kohtaan, jossa funktiota on käytetty.

Tällä kurssilla käytetyin funktio on `print`, joka *tulostaa parametrina saamansa arvot ruudulle*. Parametrien määrällä ei ole syntaktisesti rajaa, mutta on hyvä muistaa pitää koodi luettavana eli parametrien määrä maltillisena.

`print` tulostaa parametrit aina yhdelle riville välilyönnein eroteltuna. Jos halutaan tulostaa useammalle riville, on käytettävä funktiota uudelleen seuraavalla rivillä.

Syntaksi:

```
print(<parametri1>, <parametri2>, ...)
```

Esimerkki:

```
# Esimerkki ruudulle tulostamisesta.

print("Tulostetaan erityyppisiä arvoja!")
print("Kokonaislukuja:", 264, -42)
print("Liukulukuja:", 32.26, -0.001)
print("Totuusarvot:", True, False)
```

Tulostus:

```
Tulostetaan erityyppisiä arvoja!
Kokonaislukuja: 264 -42
Liukulukuja: 32.26 -0.001
Totuusarvot: True False
```

Kun `print`-funktion parametriksi annetaan jokin operaattoreita sisältävä lauseke, lasketaan lausekkeen arvo ennen tulostamista.

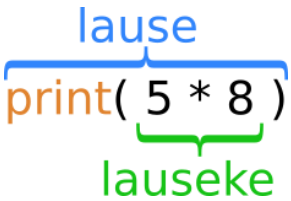
Esimerkki:

```
# Esimerkki lausekkeiden laskemisesta ennen tulostusta.

print("Lausekkeet lasketaan ennen tulostusta!")
print("4*6 =", 4*6)
print("73%3*4 =", 73%3*4)
print("3.2+(4*2)=", 3.2+(4*2))
```

Tulostus:

```
Lausekkeet lasketaan ennen tulostusta!
4*6 = 24
73%3*4 = 4
3.2+(4*2)= 11.2
```

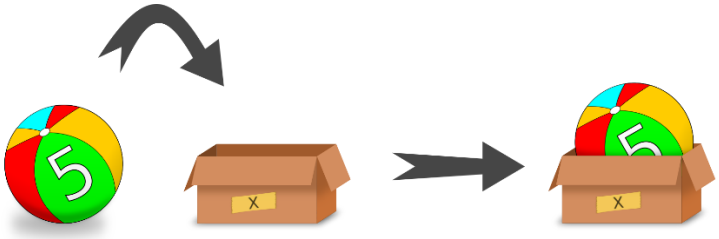
Huomaa, että lauseke ja lause ovat eri asioita.



Kuva 6: Lausekkeiden tulokset ovat yksittäisiä arvoja ja lauseet ovat jotakin, mitä tietokoneet tekevät.

3.4 MUUTTUJAT

Ohjelmoinnissa tieto tallennetaan **muuttujiin** *(variable)*, jotta tietoa voidaan käyttää myöhemmin. Muuttujilla on tunniste (nimi) ja **arvo** *(value)*. Muuttujat on määriteltävä ennen kuin niitä voidaan käyttää. Muuttujien määrittelyä kutsutaan **asettamiseksi**.



Kuva 7: Muuttujia voi ajatella ikään kuin laatikoina, joihin säilötään arvoja. Laatikoilla on nimet, joiden avulla arvot löydetään.

Pythonissa muuttujien arvot asetetaan **asetusoperaattorilla** =. Asetuksen jälkeen muuttujan arvoon pääs-tään käsiksi muuttujan tunnisteella.

<u>Syntaksi:</u>

```
<tunniste> = <arvo>
```

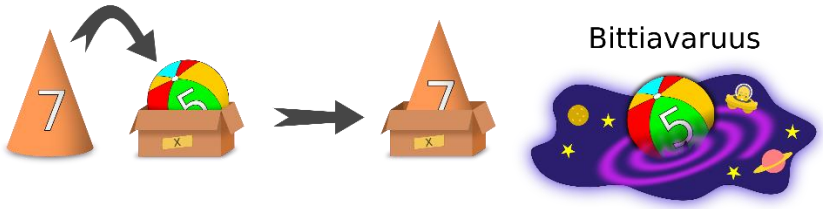
<u>Esimerkki:</u>

```
# Esimerkki muuttujien asettamisesta.
kerroin = 3
luku = 98
totuus = False
asetus = "Asetusoperaattori on =."

# Ja muuttujien käyttämisestä!
print(asetus)
print(not totuus)
print("Luku kerrottuna:", kerroin*luku)
```

Tulostus:

```
Asetusoperaattori on =.
True
Luku kerrottuna: 294
```

Muuttujien arvoja voidaan myös muuttaa. Tällöin uusi arvo kirjoitetaan edellisen päälle ja jatkossa vain uusi arvo löytyy muuttujan nimellä.



Kuva 8: Kun muuttujan arvo muutetaan, vain uusi arvo löytyy jatkossa muuttujasta ja vanha arvo katoaa "bittiavaruuteen".

Pythonissa muuttujien arvojen muuttaminen tapahtuu samalla operaattorilla kuin asettaminen. Arvojen asetuksessa ja muuttamisessa voidaan käyttää myös operaattoreita sisältäviä lausekkeita, joiden lasketut arvot tallennetaan muuttujiin.

Syntaksi:

```
<tunniste> = <arvo>
<tunniste> = <uusi arvo>
```

Esimerkki:

```
# Esimerkki muuttujien arvojen muuttamisesta.

luku = 3
print("Luku:", luku)
luku = 6
print("Luku nyt:", luku)
```

Tulostus:

```
Luku: 3
Luku nyt: 6
```

Muuttujan arvoa voidaan muuttaa myös viittaamalla muuttujan arvoon asetuslausekkeessa.

Esimerkki:

```
# Luodaan muuttuja, jonka arvoksi asetetaan 0.
luku = 0

# Kasvatetaan muuttujan arvoa yhdellä.
luku = luku + 1

# Kasvatetaan muuttujan arvoa kahdellatoista.
luku = luku + 12

# Lopuksi luku on 13.
```

Muuttujien arvoja laskettaessa **asetuslauseke** (eli merkin = jälkeinen lauseke) lasketaan aina ensin ja sen tulos asetetaan muuttujan arvoksi. Näin ollen asetuslausekkeessa voidaan viitata muuttujan arvoon, joka seuraavaksi korvataan uudella.

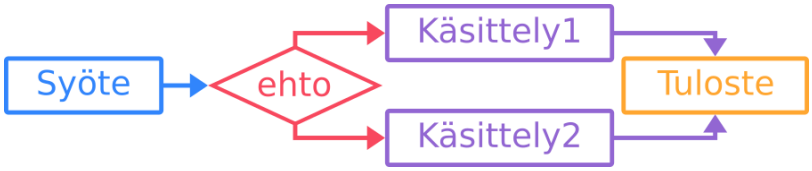
Pythonissa **muuttujan nimi saa sisältää vain kirjaimia, numeroita ja alaviivoja** (_). **Nimen pitää alkaa kirjaimella.** Huomaa, että Pythonissa pienet ja isot kirjaimet ovat eri merkkejä, joten muuttujien nimissä kirjainkoolla on merkitystä. Esimerkiksi muuttujat `eka` ja `Eka` ovat siis eri muuttujia.

Hyvä ohjelmointikäytäntö!

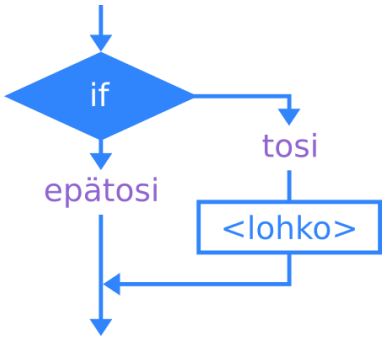
Muuttujat on hyvä nimetä mahdollisimman kuvaavasti niiden käyttötarkoituksen mukaan, jotta koodia on helpompi ymmärtää. Muuttujat kannattaa myös nimetä lyhyesti, mutta etenkin jos koodia lukee joku muukin (kuten usein tapahtuu), lyhenteitä kannattaa välttää.

3.5 EHTOLAUSEET

Joskus ohjelmoinnissa on tarpeellista tehdä jotain eri tavalla riippuen syötteestä, muuttujien arvoista tai laskutoimituksista. Voi olla hyödyllistä tarkistaa esimerkiksi, onhan luku jotain muuta kuin nolla, jos aiotaan jakaa sillä. *Ohjelman suoritus haarautuu silloin asetetun ehdon perusteella.*



Ohjelmoinnissa suorituksen haarautuminen toteutetaan ehtolauseilla. **Ehtolauseet** *(conditional statements)* ovat kokonaisuuksia, joissa lauseita suoritetaan, jos määritelty ehto pitää paikkansa. Pythonissa ehtolauseiden avainsana on `if` (jos), jonka perään kirjoitetaan suluissa tarkistettava ehto. Ehdon perusteella joko suoritetaan tai ei suoriteta lauseita. Suoritettavien lauseiden kokonaisuutta kutsutaan lohkoksi ja se voi sisältää yhden tai useamman lauseen.



Kuva 9: if-lauseessa ohjelman suoritus haarautuu joko lohkon suoritukseen tai lohkon jälkeiseen lauseeseen.

Lohkot merkitään Pythonissa sisennyksellä: rivit, joilla on sama sisennys, kuuluvat samaan lohkoon. Sisentäminen tehdään välilyönneillä tai sarkaimella eli tabulaattorilla (usein Q-kirjaimen vieressä oleva näppäin: -->|). Samaan lohkoon kuuluvilla lauseilla täytyy olla sama sisennys!

Syntaksi:

```
if <ehtolauseke>:
    <lohkon lause 1>
    <lohkon lause 2>
```

Esimerkki:

```
# Esimerkki ehtolauseesta if.

luku = 9
if (luku<100):
    print("Luku", luku, "on pienempi kuin 100!")

if (luku%3 == 0):
    print("Luku", luku, "on jaollinen kolmella!")

if (luku>0 and luku<10):
    print("Luvussa", luku, "on vain yksi numero.")
```

Tulostus:

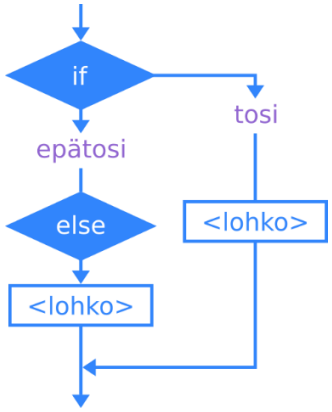
```
Luku 9 on pienempi kuin 100!
```

```
Luku 9 on jaollinen kolmella!
Luvussa 9 on vain yksi numero.
```

Hyvä ohjelmointikäytäntö!

Pythonissa lohkot erotellaan sisentämällä, mutta joissain kielissä lohkot erotetaan (aalto)sulkeilla. Syntaksista riippumatta on hyvän ohjelmointikäytännön mukaista aloittaa lohko seuraavalta riviltä ja sisentää se. Näin koodin lukeminen helpottuu.

`if`-lohkon jälkeen voidaan määrittää **vaihtoehtoinen lohko**, joka suoritetaan vain silloin, kun `if`-lauseen ehto on epätosi. Tällainen rakenne on hyvä silloin, kun yhden ehdon tarkistamisella voidaan päätellä molemmat lopputulokset. Rakenteessa siis suoritetaan aina joko `if`-lauseen lohko tai vaihtoehtoinen lohko. Pythonissa (ja monissa muissakin kielissä) vaihtoehtoinen lohko määritellään `else`-lauseella.



Kuva 10: if-else-rakenteessa suoritetaan joko if-lauseen lohko tai else-lauseen lohko ehtolausekkeen perusteella.

Syntaksi:

```
if <ehtolauseke>:
    <lohko>
else:
    <vaihtoehtoinen lohko>
```

Esimerkki:

```
# Esimerkki ehtolauseesta if-else.

luku = 93425
if (luku>0):
    print("Luku on positiivinen.")
```

```
else:
    print("Luku ei ole positiivinen.")

if (luku%5 == 0):
    print("Luku on jaollinen viidellä.")
else:
    print("Luku ei ole jaollinen viidellä.")
```

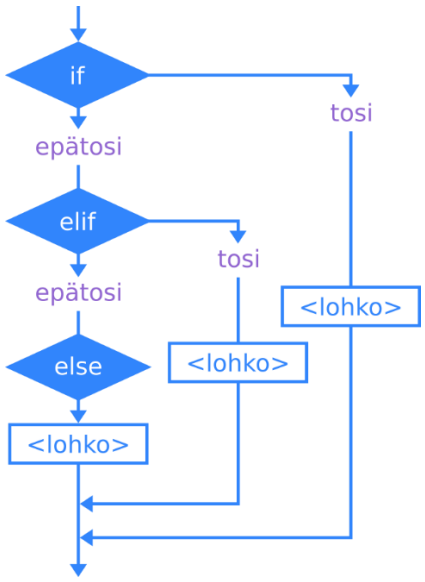
Tulostus:

```
Luku on positiivinen.
Luku on jaollinen viidellä.
```

Miksi if-else, eikä if-if?

Kaksi peräkkäistä if-lausetta voi johtaa siihen, että molempien lohkot suoritetaan, vaikka se ei olisi tarkoitus. if-else-rakenteessa näin ei käy koskaan eli if-else-rakenteella saadaan siis varmistettua, että vain toinen vaihtoehto suoritetaan.

`if`-lohkon jälkeen on mahdollista laittaa *lisää ehtolauseita, joiden ehdot testataan vain silloin, kun alkuperäinen ehto on epätosi*. Pythonissa tällainen lohko toteutetaan `elif`-lauseella. `elif`-lauseita voi olla niin monta kuin niitä tarvitaan. Lopuksi voi kirjoittaa `else`-lohkon, mutta se ei ole pakollista. `else`-lohko suoritetaan vain, jos kaikki edeltävät ehtolausekkeet ovat epätosia.



Kuva 11: Suoritus voi haarautua elif-lauseen avulla niin moneen osaan kuin tarvitaan. Ainoastaan if-lause on pakollinen.

Syntaksi:

```
if <ehtolauseke1>:
    <lohko>
elif <ehtolauseke2>:
    <lohko>
elif <ehtolauseke3>:
    <lohko>
# elif-lauseita voi olla niin monta kuin tarvitaan
else:
    <lohko>
```

Esimerkki:

```
# Esimerkki ehtolauseesta if-elif-else.

luku = 0
if (luku<0):
    print("Luku on negatiivinen.")
elif (luku==0):
    print("Luku on 0.")
else:
    print("Luku on positiivinen.")
```

Tulostus:

```
Luku on 0.
```

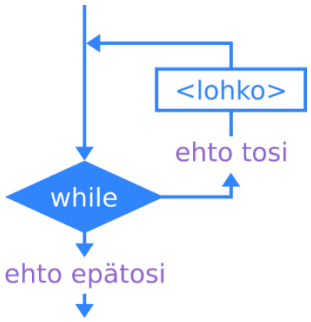
3.6 TOISTOLAUSEET

Toistolauseiden *(repetitive statements)* eli **silmukoiden** *(loops)* avulla ohjelmakoodin osaa voidaan suorittaa monta kertaa peräkkäin. Toistolauseiden käytöllä vältytään saman koodin uudelleen kirjoittamiselta.

Pythonissa toistolauseen sisällä suoritettavat lauseet sisennetään omaksi lohkoksi (samalla tavalla kuin ehtolauseissa).

Pythonin toistolause voidaan toteuttaa `while`-lauseella, jolle määritellään ehto, jonka ollessa voimassa lohkon suoritusta toistetaan.

`while`-lause eroaa `if`-lauseesta: `while`-lauseen lohkoa suoritetaan niin kauan, kun ehto on tosi, mutta `if`-lauseen lohko suoritetaan ainoastaan kerran, jos ehto on tosi.



Kuva 12: while-silmukan avulla lohkoa voidaan suorittaa uudelleen ja uudelleen niin kauan, kun ehtolauseke on tosi.

`while`-lauseen ehtona voi olla mikä tahansa ehtolauseke (eli `True`, `False` tai mikä tahansa vertailu- ja loogisilla operaattoreilla muodostettu ehtolauseke). **Usein lohkon sisällä muutetaan jotakin arvoa niin, että ehto voi jossain kohtaa muuttua epätodeksi.** Jotta silmukan suoritukseen ei jäädä ikuiseksi ajaksi, on ehdon jossain kohtaa muututtava epätodeksi.

Syntaksi:

```
while <ehtolauseke>:
    <lohko>
```

Esimerkki:

```
# Esimerkki toistolauseesta while.

# Luku, jonka kertotaulu tulostetaan
luku = 7
# Apumuuttuja
apu = 0
# Toistolause kertotaulun tulostamiseksi
while (apu <= 10):
    # Tulostetaan kertolasku ja sen tulos
    print(apu, "*", luku, "=", apu*luku)
    # Kasvatetaan apumuuttujan arvoa
    apu = apu + 1
```

Tulostus:

```
0 * 7 = 0
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
```

```
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
```

Vinkki!

Jos päädyt luomaan ja suorittamaan ikuisen silmukan Python IDLEssä, saat sen lopetettua näppäinyhdistelmällä Ctrl + C (Windows-tietokoneissa) tai sulkemalla IDLEn Shell-ikkunan.

`while`-lauseen lisäksi lohkoja voidaan toistaa `for`-lauseella. `for`-lauseen avulla käydään läpi olioita, joita voidaan käsitellä listana. (Listoihin palataan kurssilla myöhemmin.) Esimerkiksi merkkijonot ovat tällaisia: `for`-lauseella voidaan käydä läpi yksitellen jokainen merkkijonon merkki.

Syntaksi:

```
for <merkkimuuttuja> in <merkkijono>:
    <lohko>
```

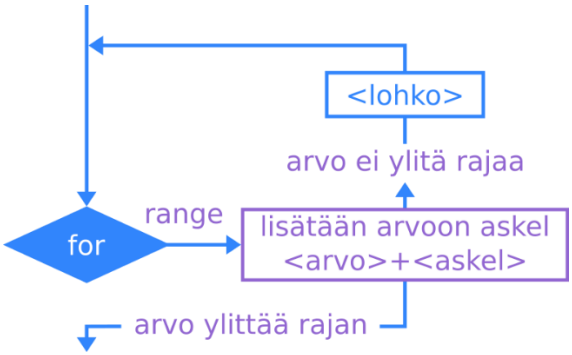
Esimerkki:

```
# Esimerkki merkkijonon merkkien läpi käymisestä.

nimi = "Python"
for kirjain in nimi:
    print(kirjain)
```

Tulostus:

```
P
y
t
h
o
n
```

`for`-lauseen kanssa voidaan käyttää apuna myös funktiota `range`, joka luo listan parametreina annettujen arvojen mukaan. Parametrit ovat alkuarvo, loppuarvo ja askel. `for`-silmukan suoritus päättyy, kun loppuarvo saavutetaan eli silmukkaa ei koskaan suoriteta loppuarvolla.



Kuva 13: for-silmukan kanssa voidaan myös käyttää range-funktiota, joka sitten määrittelee, millä luvuilla silmukka suoritetaan.

Syntaksi:

```
for <luku> in range(<alkuarvo>, <loppuarvo>, <askel>)
    <lohko>
```

Esimerkki:

```
# Esimerkki toistolauseesta for.

print("for luku in range(1,10,1):")
for luku in range(1,5,1):
    print(luku)

print("for luku in range(0,50, 5):")
for luku in range(0,50,5):
    print(luku)
```

Tulostus:

```
for luku in range(1,5,1):
1
2
3
4
for luku in range(0,50, 5):
0
5
10
15
20
25
30
35
40
45
```

3.7 EHTOLAUSEIDEN JA TOISTOLAUSEIDEN YHDISTÄMINEN

Usein ohjelmissa tarvitaan toistolauseiden ja ehtolauseiden yhdistämistä niin, että lauseet ovat sisäkkäin. Ehtolause voi olla toistolauseen sisällä tai päinvastoin. **Tällaisissa tapauksissa on syytä olla erittäin huolellinen lohkojen sisennyksissä!**

Syntaksi (sulkuihin on merkitty se lause, jonka lohkon osa on kyseessä)**:**

```
# Ehtolause toistolauseen sisällä:
while <ehtolauseke>:
    <lohko (while)>
    if <ehtolauseke>:
        <lohko (if)>
    else:
        <lohko (else)>
    <lohko (while)>

# Toistolause ehtolauseen sisällä:
if <ehtolauseke>:
    <lohko (if)>
    for <merkki> in <merkkijono>:
        <lohko (for)>
    <lohko (if)>
else:
    <lohko (else)>
```

Esimerkki:

```
# Esimerkki ehtolauseiden ja toistolauseiden yhdistämisestä.

# Ehtolause toistolauseen sisällä
luku = 19
while luku > 0:
    if luku / 2 == 5:
        print("Luku:", luku)
        print("Puolet luvusta:", luku / 2)
    luku = luku - 1
print ("Luku toistolauseen jälkeen:", luku)

# Toistolause ehtolauseen sisällä
luku = 5
if luku == 5:
    print("Luvun 5 kertotaulu:")
    for indeksi in range(0, 11, 1):
        print(indeksi * luku)
else:
    print ("Luku ei ollut 5!")
```

Tulostus:

```
Luku: 10
Puolet luvusta: 5.0
Luku toistolauseen jälkeen: 0
Luvun 5 kertotaulu:
0
5
10
15
20
25
30
35
40
45
50
```

4 TYÖPAJA 1: KÄYTTÄJÄN SYÖTE, EHTOLAUSEET JA TOISTOLAUSEET

Työpajassa käsiteltiin ohjelmoinnin perusteita, joihin tutustuminen aloitettiin verkkosisällössä 1. Työpajassa kerrattiin ohjelmoinnin perusteet, mutta erityisesti kiinnitettiin huomiota ehtolauseisiin ja toistolauseisiin, koska ne ovat tärkeitä ohjelmoinnin rakenteita. Uutena asiana käsiteltiin tiedon pyytäminen käyttäjältä ohjelman suorituksen aikana.

Työpajassa käytiin lyhyesti lävitse edeltävä verkkosisältö, jonka jälkeen harjoiteltiin käytännössä käyttäjän syötteen käsittelyä, ehtolauseiden hyödyntämistä ja lopuksi toistolauseiden käyttöä.

<u>Käyttäjän syöte</u>

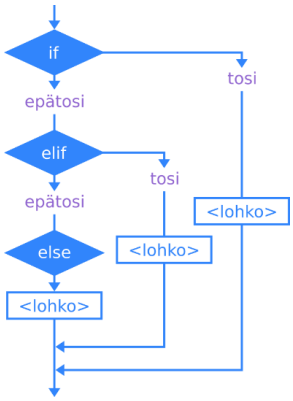
- Monissa ohjelmissa hyödynnetään käyttäjältä saatua tietoa.
- Käyttäjältä voidaan esimerkiksi kysyä hänen tietojaan, ohjelmassa tarvittuja lukuarvoja tai muita asioita, joita tarvitaan ohjelman suorittamiseksi.
- input-lause
- Merkkijonoista luvuiksi

Esimerkki

```
# Kysytään käyttäjältä kaksi kokonaislukua
syote1 = input("Anna ensimmäinen kokonaisluku: ")
syote2 = input("Anna toinen kokonaisluku: ")

# Muutetaan syötteet kokonaislukutyyppisiksi
luku1 = int(syote1)
luku2 = int(syote2)

# Tulostetaan luvuilla tehtäviä laskutoimituksia
print(luku1, "+", luku2, "=", luku1+luku2)
print(luku1, "*", luku2, "=", luku1*luku2)
print(luku1, "^", luku2, "=", luku1**luku2)
```

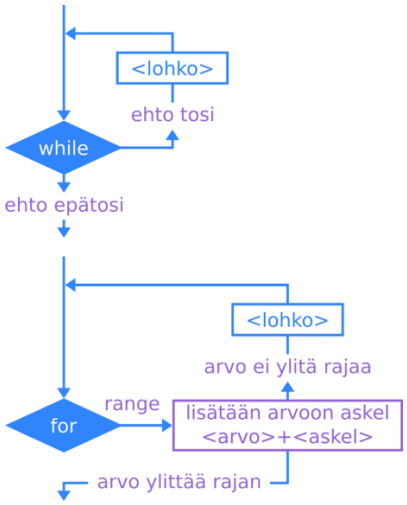
<u>Ehtolauseet</u>

- suorituksen haarautuminen
- if -lause
- ehtolauseke: loogiset ja vertailuoperaattorit
- if - else -rakenne
- if - elif - else rakenne



Toistolauseet

- suorituksen toistaminen
- while -toistolause
- for -toistolause
- jonon läpikäynti
- range -funktio
- Ehto- ja toistolauseiden yhdistäminen

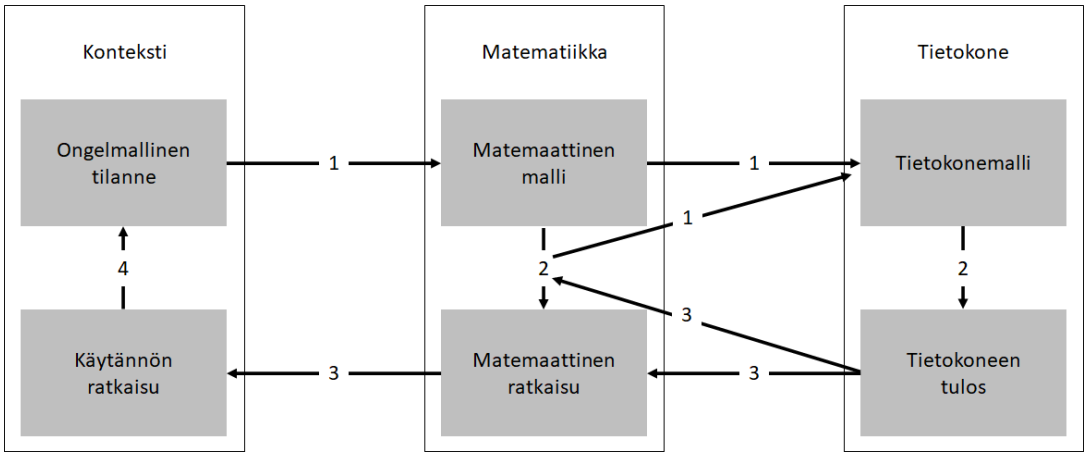


5 VERKKOSISÄLTÖ 2: ONGELMANRATKAISU JA ALGORITMINEN AJATTELU

Matematiikan ja ohjelmoinnin yhteys juontaa ensimmäisestä tietokoneesta, joka luotiin laskimeksi. Poiketen sen ajan mekaanisista laskimista tietokone pystyi tallentamaan välitulokset ja hyödyntämään niitä uusissa laskutoimituksissa. Ensimmäisen tietokoneen ohjaaminen oli kuitenkin konekeskeistä eli koneen toiminta määriteltiin kytkimillä ja johdotuksilla. Vuonna 1945 julkaistussa nykyaikaisen tietokoneen mallissa (Neumann 1945) toimintaperiaate oli muuttunut matemaattiseksi ja ohjaaminen tapahtui symbolien avulla. Symbolisella ohjaamisella tarkoitetaan peräkkäis-, ehto- ja toistorakenteita, joita käytetään koneen ominaisuuksien hyödyntämisen järjestämiseen. Ohjausrakenteita voidaan pitää matemaattisina, koska niillä voidaan myös kuvata laskutoimitusten algoritmien/proseduurien etenemistä.

Tietokoneessa algoritmi ohjaa koneen toimintaa, joten sitä voidaan tulkita sekä matemaattisesta että teknisestä näkökulmasta. Ensimmäisestä tulkinnasta vastaa peruskoulun opetuksessa matematiikka ja toisesta käsityö. Keskitymme tässä matematiikkaan, mutta on hyvä huomioida myös käsityön näkökulma, koska osana matematiikka käytämme tietokoneen teknisiä ominaisuuksia ja osana käsityötä hyödynnämme myös matematiikan mahdollisuuksia.

Yläkoulun matematiikan opetussuunnitelma määrittelee ohjelmoinnin keskeiseksi tavoitteeksi (T20) "ohjata oppilasta kehittämään algoritmista ajatteluaan sekä taitojaan soveltaa matematiikkaa ja ohjelmointia ongelmien ratkaisemiseen" (OPH, s.375, 2014). Ratkaisemisella tarkoitetaan tässä tosielämän ongelman kuvaamista, kuvauksen muuttamisesta matemaattiseksi malliksi (kaava) ja ratkaisun laskemista (Kallia ym. 2021; ks. Kuva 14). Tapauksessa, jossa tarvitaan tietokoneen ominaisuuksia, matematisoitu ongelma voidaan mallintaa ohjelmointikielellä ja laskea tulos tietokoneella.



Kuva 14: Matemaattinen ja ohjelmoinnillinen ongelmanratkaisu. Matemaattista ja tietokoneen ohjelmointiin perustuvaa ongelmanratkaisua yhdistää algoritmin laskeminen (2). Lähtökohtana on ongelman kuvaus, jonka sisältämistä käsitteistä ja niiden välisistä suhteista luodaan ongelmaa kuvaava kaava (1)

eli ongelman matemaattinen malli. Kaavan voi ratkaista laskemalla käsin (2), mutta jos laskeminen on hankalaa tai dataa on paljon, voidaan käyttää tietokonetta apuvälineenä. Silloin luodaan tietokonemalli kaavasta ohjelmoimalla (1). Tietokoneen tulosta voi hyödyntää (3) sekä kokonaisena että osittaisena matemaattisena ratkaisuna. Mahdollisen matemaattisen käsittelyn jälkeen pitää ratkaisu vielä tulkita ongelman kontekstissa (3), ennen kuin ongelma on ratkaistu käytännössä (4).

Ongelman kuvaamisella tarkoitetaan sen keskeisten piirteiden esille tuontia. Kuvauksen matemaattisella analyysilla pyritään löytämään ongelman keskeiset muuttujat ja niiden väliset suhteet. Lopputuloksena on ongelman matemaattinen malli eli sitä kuvaava kaava, joka voidaan ratkaista laskemalla. Laskutoimitus on algoritmin noudattamisen tulos, joka on tuotava takaisin ongelman kontekstiin sen ratkaisemiseksi. Ongelman kontekstin huomioiminen vastauksessa on tärkeää, koska siinä voi olla reunaehtoja, jotka eivät ole matemaattisia (Rich & Yadav 2020). Esimerkiksi tosielämässä ei tilata osittaisia busseja vaan kokonaisia (ei 1,4 bussia vaan 2 bussia).

Ohjelmoimalla voidaan luoda matemaattisista malleista tietokonemalleja, kun laskeminen olisi muuten liian työlästä, tarvitaan nopeita kokeiluja tai ratkaisutapa on seurausta usean laskutoimituksen vuorovaikutuksesta nk. simulaatiot. Tietokonemallin suorituksen tuloksista palataan matemaattiseen kontekstiin, koska tulos voi olla laskutoimituksen osa tai tulos voi vielä vaatia matemaattista tarkastelua.

Tässä osiossa keskitymme ongelmanratkaisuun luomalla matemaattisia ja tietokonepohjaisia malleja. Mallintaminen perustuu molemmissa algoritmiseen ajatteluun, mutta samalla huomataan, että matematiikassa ja ohjelmoinnissa algoritmia käsitellään eri tavalla.

Lähteet:

OPH. (2014) Perusopetuksen opetussuunnitelman perusteet. Opetushallitus.

Kallia, M., van Borkulo, S. P., Drijvers, P., Barendsen, E., & Tolboom, J. (2021). Characterising computational thinking in mathematics education: a literature-informed Delphi study. Research in Mathematics Education, 1–29.

Rich, K. M., & Yadav, A. (2020). Applying levels of abstraction to mathematics word problems. TechTrends, 1–9.

5.1 MATEMAATTINEN ONGELMANRATKAISU

Yksinkertaiset laskutoimitukset perustuvat pitkälti ulkoa opittuun ja aina ei välttämättä edes ajatella laskemista vaan vastaus tiedetään. Kun laskeminen menee monimutkaisemmaksi, niin se muuttuu symbolien manipuloimiseksi ja usein käytetään lisäksi ulkoisia apuvälineitä. Symbolien manipulointi on algoritmista alkaen allekkainlaskuista ja jatkuen algebran kaavojen käsittelyyn. Myös harpin ja viivaimen käyttö

geometriassa on esimerkki algoritmista. Ihmisen suorittamana algoritmi muodostuu niistä proseduureista, joita tarvitaan laskutoimituksen tuloksen saamiseksi. Proseduurien valinta tapahtuu niiden oppimisen jälkeen monesti tiedostamatta.

Ongelman ratkaisu on yksinkertaisimmillaan ongelmaan sopivan algoritmin havaitsemista ja suorittamista. Aina ongelmat eivät kuitenkaan ole näin suoraviivaisia. Algoritmin taustalla olevien matematiikan käsitteiden ymmärtäminen mahdollistaa ratkaisun perustamisen ongelman luonteeseen. Silloin kiinnitetään huomiota ongelmaa kuvaaviin käsitteisiin ja niiden välisiin suhteisiin, joiden perusteella valitaan tai luodaan ratkaisun kaava. Usein ongelma voidaan ratkaista usealla tavalla, jolloin on pystyttävä arvioimaan mikä on tilanteessa paras. Välillä ongelmaan ei löydy valmista algoritmia vaan sellainen on luotava. Nämä kaikki ovat esimerkkejä algoritmisesta ajattelusta, joka voidaan jakaa kolmeen tasoon: tieto ja taito, käsitteellinen ymmärtäminen sekä arviointi ja luominen (Fan & Bokhove 2014).

Algoritmisen ajattelun tasot:

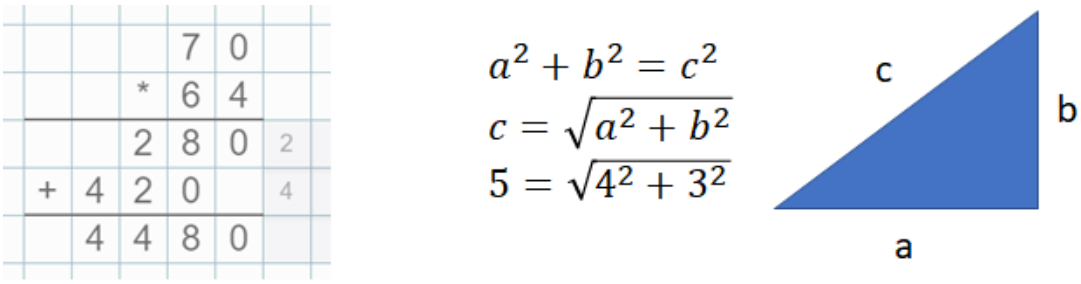
Taso	**Kuvaus**
1. Tieto ja taito	Tuntee kaavan sisältämät proseduurit, jonka perusteella osaa suorittaa laskutoimituksen. Soveltaminen on tällä tasolla suoraviivaista ja algoritmin muokkaaminen on vähäistä.
2. Käsitteellinen ymmärtäminen	Tietää miksi algoritmi toimii ja miten sitä voi soveltaa monimutkaisempiin ongelmiin.
3. Arviointi ja luominen	Pystyy arvioimaan erilaisten algoritmien paremmuutta (tehokkuus tai yksinkertaisuus). Osaa yleistää algoritmin sopimaan samantyyppisiin ongelmiin ja luoda uusia algoritmeja.

Lähde:

Fan, L., & Bokhove, C. (2014). Rethinking the role of algorithms in school mathematics: A conceptual model with focus on cognitive development. ZDM, 46(3), 481-492.

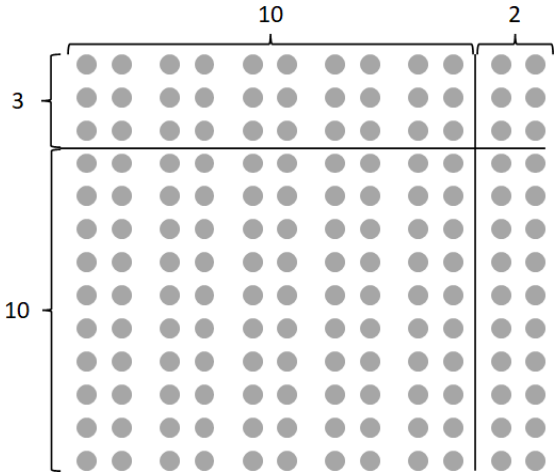
5.2 ALGORITMINEN AJATTELU MATEMATIIKASSA

Edellä algoritminen ajattelu jaettiin kolmeen tasoon. Tiedon ja taidon tasolla osaaminen liittyy laskukaavan ja sen toteuttavan algoritmin muistamiseen. Oppilas taitaa tarvittavat proseduurit ja pystyy yhdistämään ne kaavan määrittelemän laskutoimituksen suorittamiseksi. Tällä tasolla osaaminen on mekaanista ja se soveltuu hyvin suoraviivaisiin laskutehtäviin. Esimerkkeinä algoritmeista ovat alakoulusta tuttu kertolasku allekkain aritmetiikassa ja yläkoulussa Pythagoraan lause geometriassa (ks. Kuva 15).



Kuva 15: Kertolasku allekkain ja Pythagoraan lause sekä toteutus

Kun ongelmat ovat monimutkaisempia tai vaativat menetelmien soveltamista, pelkän ratkaisutavan osaaminen ei riitä, vaan tarvitaan sen lähtökohtien ymmärtämistä. Käsitteellisen ymmärtämisen tasolla oppilas tuntee matematiikan osa-alueen käsitteet ja niiden väliset suhteet. Oppilas ymmärtää miten algoritmi on seurausta osa-alueen rakenteesta. Ymmärtämistä voidaan tukea käyttämällä kuviokieltä (Joutsenlahti & Tossavainen 2018) esim. moninumeroisilla luvuilla kertominen voidaan esittää visuaalisesti alkioista koostuvana taulukkona, josta toinen luku määrittelee rivin alkioiden ja toinen rivien määrän (ks. Kuva 16).



Kuva 16: Kertolasku 12 kertaa 13 on esitetty visuaalisesti: siinä näkyvät lukujen edustamat määrät ja kertolaskun vaikutus tulokseen

Arvioinnin ja luomisen tasolla osaaminen ei koske enää yksittäisten algoritmien tuntemista vaan silloin pystytään vertaamaan erilaisten algoritmien paremmuutta, havaitsemaan miten tietystä ratkaisusta voi luoda yleiskäyttöisemmän algoritmin ja luomaan uusia algoritmeja aikaisemmin opittuja yhdistelemällä. Tavallisistakin laskutoimituksista löytyy erilaisia algoritmeja (Morgan 2019), jolloin ero johtuu operaatioiden järjestyksestä tai löytyy erilaisia operaatioiden yhdistelmiä, joilla saadaan sama tulos. Algebrassa kaava voidaan yleistää korvaamalla numeroita muuttujilla, jolloin se sopii saman tyyppisiin ongelmiin, joissa vain muuttujien arvot vaihtelevat. Uuden algoritmin luominen johonkin matematiikan osa-alueeseen on epätodennäköistä, mutta myös uudelleen keksiminen on opettavaa. Paremmat mahdollisuudet

uniikkien algoritmien luomiselle on yhdistää algoritmeja jonkin matematiikan ulkopuolisen ongelman ratkaisemiseksi.

Lähteet:

Joutsenlahti, J., & Tossavainen, T. (2018). Matemaattisen ajattelun kielentäminen ja siihen ohjaaminen koulussa. In Matematiikan opetus ja oppiminen (pp. 410-431).

Morgan, J. (2019). A Compendium of Mathematical Methods. John Catt Educational.

5.3 MATEMAATTISESTA ONGELMANRATKAISUSTA TIETOKONEMALLIIN

Kun ongelma on ensin kuvattu täsmällisesti, voidaan se mallintaa matemaattisesti (ks. edellä luku 5 alku Kuva 14 nuoli 1 ongelmasta matemaattiseen malliin). Mallinnettaessa luodaan ongelmaa kuvaava matemaattinen kaava, jonka muuttujat ja arvot edustavat ongelman tekijöitä. Operaattorit edustavat ongelman tekijöiden välisiä suhteita. Ongelman ratkaisu on kaavan edustaman algoritmin laskeminen (ks. edellä luku 5 alku Kuva 14 nuoli 2 matemaattisesta mallista ratkaisuun). Laskemista voi tehdä eri tavalla, jolloin kaavan operaatiot tehdään eri järjestyksessä. Saman ongelman ratkaisuun voi löytyä myös erilaisia kaavoja. Nämä perustuvat siihen, että matematiikan osa-alueessa voi olla erilaisia käsitteiden yhdistelmiä, joilla voidaan esittää sama asia.

Edellä on esitetty kolme matemaattisen algoritmin ymmärtämisen tasoa: tiedon ja taidon, käsitteellisen ymmärtämisen sekä arvioinnin ja luomisen. Algoritmien ymmärtäminen vaikuttaa myös kykyyn ratkoa ongelmia. Tiedon ja taidon tasolla on ongelman muistutettava oppilaan aikaisemmin kohtaamaa (tieto), jotta hän voi ratkaista sen tuntemansa kaavan (tieto) ja sen ratkaisevan algoritmin avulla (taito). Käsitteellinen ymmärtäminen mahdollistaa monimutkaisemmasta ongelmasta tutun rakenteen havaitsemisen tai ongelman muokkaamisen sen esille tuomiseksi. Kyky arvioida algoritmien paremmuutta ja luoda omia algoritmeja on ongelmanratkaisun korkein taso.

Vaikka matemaattinen ratkaisuperiaate tiedetään, voi laskun monimutkaisuus tai datan määrä olla sellainen, että laskutoimitusta (algoritmi) ei pystytä suorittamaan järkevässä ajassa. Tietokoneet kehitettiin laskemaan ihmistä tehokkaammin ja ohjelmoimalla voidaan algoritmi kuvata tietokoneen ymmärtämässä muodossa. Tekoälyn kehittymisen myötä voidaan ongelmia myös ratkaista koneoppimisella esimerkiksi silloin, kun data on liian epätasaista perinteisin menetelmin määriteltäväksi tai on pystyttävä käsittelemään myös sellaisia tapauksia, joita ei tiedetä etukäteen. Koneoppimisessa tietokone opetetaan mallidatalla tunnistamaan haluttuja piirteitä, jolloin varsinaisesta datasta voidaan löytää samankaltaisia asioita.

Matemaattisen mallin (kaavan) voi muuntaa tietokonemalliksi määrittelemällä sitä laskevan ohjelman (ks. edellä luku 5 alku Kuva 14 nuoli 1 matemaattisesta mallista tietokonemalliin). Vaihtoehtoisesti tietokoneelle voidaan delegoida vain laskutoimituksen osa (ks. edellä luku 5 alku Kuva 14 matematiikan nuolesta

2 tietokoneeseen lähtevä nuoli 1). Algoritmia ohjelmoitaessa on otettava huomioon tietokoneen ominaisuudet (mm. laskuoperaatiot, ohjausrakenteet, muistinkäsittely sekä syöte ja tuloste) laskennan automatisoimiseksi.

5.4 ONGELMANRATKAISU OHJELMOIMALLA

Algoritmien arviointi (arvioinnin ja luomisen taso) tuo esille, että sama laskutoimitus (kaava) voidaan toteuttaa erilaisilla algoritmeilla. Algoritmi voidaan myös toteuttaa erilaisilla menetelmillä (esim. symbolinen ja visuaalinen laskenta). Ohjelmointi tarjoaa yhden menetelmän lisää ongelmanratkaisuun, joka myös automatisoi laskennan. Se poikkeaa matematiikan laskemisesta, koska laskemisen sijaan algoritmi määritellään eksplisiittisesti: näin ohjelmointi muistuttaa enemmän matemaattista päättelyä tai todistamista. Samantyyppisiä laskutoimituksia laskevassa ohjelmassa on lisäksi huomioitava erilaisten arvojen mahdollinen vaikutus algoritmin suoritukseen.

Algoritmien ohjelmoinnin ymmärtämistä voidaan myös tarkastella vastaavasti kuin matematiikan laskutoimituksia. Carsten Schulten (2008) luoma ohjelmoinnin lohkomalli (ks. Taulukko 2.2) jakaa ohjelman ymmärtämisen kolmeen luokkaan: ohjelmakoodi, ohjelman suoritus ja ohjelman tarkoitus. Kaksi ensimmäistä luokkaa liittyy ohjelman rakenteeseen ja kolmas ohjelman toimintaan. Luokat on vielä jaoteltu tarkastelutason mukaan tasoihin: atomi, lohko, lohkojen suhteet ja makro. Lohkomallin avulla voidaan analysoida ohjelmakoodin ymmärtämistä ja havaita sen puutteita.

<u>Ohjelmoinnin ymmärtämisen lohkomalli (Schulte 2008):</u>

	Rakenne		Toiminnallisuus
	Ohjelmakoodi	**Ohjelman suoritus**	**Ohjelman tarkoitus**
Makrotaso	Ohjelman rakenteen ymmärtäminen	Ohjelman/Algoritmin suorituksen ymmärtäminen	Ohjelman tarkoituksen ymmärtäminen
Lohkojen suhteet	Lohkojen väliset suhteet (esim. jaetut muuttujat tai funktiokutsut)	Lohkojen suoritusjärjestyksen ymmärtäminen	Miten toinen lohko liittyy tarkasteltavan lohkon tarkoitukseen
Lohkot	Ohjelmointikielen käsitteistä koostuvia lauseita tai lauseiden muodostamia kokonaisuuksia	Lohkon suorituksen ymmärtäminen	Lohkon tarkoitus (osatavoite)

Atomit	Ohjelmointikielen käsitteet	Lauseen suorittaminen	Lauseen suorituksen tarkoitus

Lähteet:

Schulte, C. (2008). Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In Proceedings of the Fourth international Workshop on Computing Education Research (pp. 149-160).

5.5 ALGORITMINEN AJATTELU OHJELMOINNISSA

Ohjelmoinnin ymmärtämisen lohkomallissa ohjelmakoodin luokassa ohjelman tekstin tarkoituksena on kuvata (matemaattista) algoritmia ja lisäksi tarvittavia tietokoneen toimintoja, kuten syötteen lukeminen ja tuloksen julkaiseminen. Oppilaan on tunnistettava ohjelmointikielen käsitteet (atomit), jotta hän voi niistä koostaa järkeviä lauseita (lohko). Samaan tehtävään liittyvät lauseet muodostavat isomman lohkon ja usein ohjelmointikielessä on myös valmiita käsitteitä lohkojen esille tuomiseksi (esim. toistolauseen sisältämät lauseet). Kahden lohkon välillä voi olla suhde: esim. yhdestä lohkosta kutsutaan toisaalla määriteltyä funktiota. Makrotasolla teksti kuvaa ohjelman rakennetta, joka koostuu lohkoista ja yksittäisistä lauseista. Kun oppilas tuntee ohjelmointikielen käsitteet ja osaa niiden yhdistämisen säännöt, pystyy hän kuvaamaan haluamansa algoritmin.

Ohjelmointi on tietokoneen ohjaamista ja vaikka ohjelmointikielet sopivat hyvin matemaattisten algoritmien esittämiseen, voi ohjelman suorittamisen tulos aiheuttaa yllätyksiä. Ohjelman suorituksen luokassa kiinnitetään huomiota siihen, millainen malli tietokoneesta on ohjelmointikielen taustalla. Ohjelmointikielen suorituksen malli (Sorva 2018) on tietokoneen yksinkertaistettu esitys, jolla sen ominaisuuksia voidaan hyödyntää. Imperatiivisissa kielissä (suomessa suosituin kielten luokka), kuten Python, ohjelman suorituksen seuraukset kohdistuvat keskeisesti muistiin. Jotta ohjelma toimisi oikein on muuttujien (edustavat varattua tilaa muistissa) käsittelyyn kiinnitettävä huomiota, jotta halutut muutokset tallennetaan ja vahingossa ei kirjoiteta myöhemmin tarvittavien tietojen päälle. Ohjelman toiminnan ymmärtämistä voidaan auttaa harjoittelemalla ohjelman suorituksen muistiin aiheuttamien muutosten kirjaamista yksinkertaisella visualisoinnilla (ks. Kuva 17).

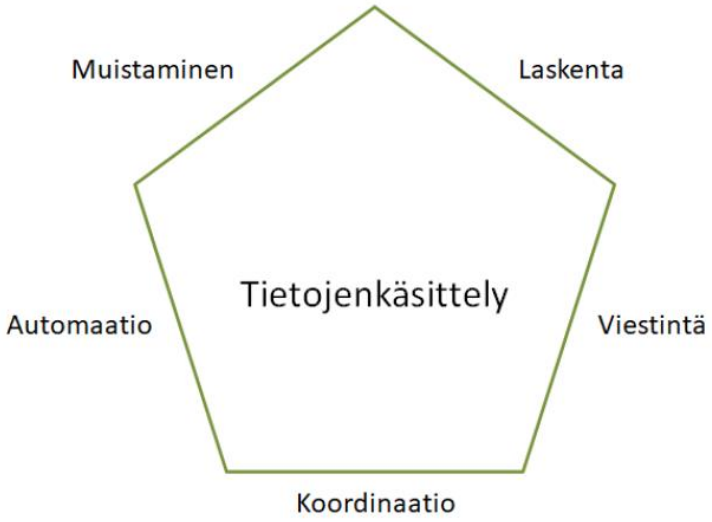
```
s = "hello"
x = 4
y = 6
x = 8
total = x + y + x
print(total)
```

muuttujan nimi	arvo
s	"hello"
x	~~4~~ 8
y	6
total	22

Tuloste
22

Kuva 17: Ohjelmointikielen suorituksen malli, joka kuvaa ohjelman muistin käsittelyä. Muisti kuvataan taulukkona, jossa vasemmalla on muuttujien nimet ja oikealla muistissa niitä vastaavat arvot. Muuttujat lisätään taulukkoon esiintymisjärjestyksessä ja jokaista muuttujaa edustaa yksi rivi. Jos aikaisemmin lisätyn muuttujan arvoa muutetaan, niin silloin yliviivataan edellinen arvo ja laitetaan sen vierelle uusi (muuttujassa voi olla vain yksi arvo kerrallaan!) Muuttujien arvojen lisäksi merkitään erikseen ohjelman tulosteet.

Ohjelmoinnin ymmärtämisen lohkomallin kolmannessa luokassa kohteena on ohjelman tavoite tai tarkoitus. Ohjelman tarkoitus muodostuu alkaen yksittäisistä lauseista (atomi), millaisia lohkoja nämä muodostavat (lohko), missä järjestyksessä eri lohkot suoritetaan (lohkojen suhteet) ja lopuksi millainen kokonaisuus syntyy näiden suorittamisesta (makro). Lauseiden ja niistä muodostettujen lohkojen merkitys tulee niiden käyttämistä tietokoneen ominaisuuksista. Tietojenkäsittelyn mekanismien (Computing mechanics; Denning 2003) avulla voidaan luokitella ominaisuudet: laskentaan, viestintään, koordinaatioon, automaatioon ja muistamiseen liittyviksi (ks. Kuva 18). Ohjelman tarkoituksen luokassa laskutoimituksen toteuttavan algoritmin määrittely on yksi osa kokonaisuutta, joka johtaa ohjelman toimimaan halutun tavoitteen mukaisesti. Oppilaan on suunniteltava kaikki ominaisuudet, jotka ohjelmalla on oltava ongelman ratkaisemiseksi.



Kuva 18: Tietojenkäsittelyn mekanismit tarjoavat viisi tietojenkäsittelytieteen näkökulmaa. Näkökulmia voi käyttää kuvaamaan tietokoneen ja siihen liittyvien teknologioiden hyödyntämisen mahdollisuuksia.

Lähteet:

Denning, P. J. (2003). Great principles of computing. Communications of the ACM, 46(11), 15-20.

5.6 ALGORITMINEN AJATTELU JA ONGELMANRATKAISU

Edellä on esitelty kolme ohjelmoinnin ymmärtämisen luokkaa tulkintana algoritmisesta ongelmanratkaisusta tietokoneella. Ohjelmakoodin luokassa keskitytään ohjelman tekstiin algoritmin kuvaajana. Ohjelman suoritus nostaa esille sen, että tietokoneella algoritmin esitys viittaa koneen toiminnan ohjaamiseen. Ohjelman tarkoitus liittää ongelman tietokoneen ominaisuuksien yhdistelmän tarjoamaan ratkaisuun. Tietokoneen tuloste on ratkaisu esitettyyn ongelmaan (ks. luku 5 alku Kuva 14). Se voi vaatia vielä matemaattista tulkintaa tai käsittelyä ennen kuin se voidaan asettaa alkuperäisen ongelman kontekstiin sen ratkaisemiseksi.

Vaikka ohjelmakoodi antaisi ymmärtää, että matemaattinen ja tietotekninen ongelmanratkaisu olisi samanlaista, tarkempi tarkastelu (matematiikan algoritmisen ajattelun tasojen ja ohjelmoinnin ymmärtämisen lohkomallin luokkien vertailu) tuo esille niiden erilaisuuden. Ohjelmakoodin tarkoituksena on ohjata tietokoneen toimintaa ja hyödyntää sen ominaisuuksia. Korkeatason ohjelmointikielten tarjoamat matemaattiset symbolit ovat valmiita teknisiä toteutuksia. Kun toimitaan näiden puitteissa ohjelmointikielen lausekkeissa, voidaan luottaa siihen, että ne käyttäytyvät matemaattisesti oikein. Matemaattisten lausekkeiden ulkopuolella arvojen käsittely perustuu tekniseen malliin, joka on huomioitava, kun yhdistetään laskutoimituksia ohjelmointikielen muilla rakenteilla.

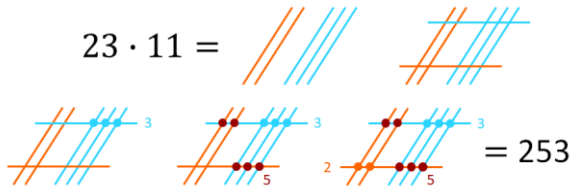
Tietokone luotiin matematiikan työvälineeksi. Matematiikasta on myös hyötyä tietokoneen ohjelmoinnissa, vaikka kohteena olisi muu kuin matemaattinen ongelma. Matematiikka tarjoaa valmiita ratkaisuja monenlaisten säännönmukaisuuksien käsittelyyn. Yhtäläisyyksistä huolimatta algoritmit matematiikassa ja ohjelmoinnissa ovat erilaisia. Ohjelmointia opetettaessa sekä yhtäläisyydet että erot on tehtävä selkeäksi, jotta vältytään väärinkäsityksiltä.

6 TYÖPAJA 2: MATEMAATTISET ALGORTIMIT, ALGORITMIEN OHJELMOINTI JA OHJELMIEN SUORITUS

Työpajassa käsiteltiin algoritmista ajattelua ja ongelmanratkaisua sekä matematiikassa että ohjelmoinnissa. Jotta ongelma voidaan ratkaista matemaattisesti, on siitä luotava malli. Matemaattinen malli on kaava, joka kuvaa ongelmaan liittyviä muuttujia, vakioita ja operaatioita. Kaava ratkaistaan laskemalla eli hyödyntämällä algoritmia, jossa symboleja manipuloimalla päästään lopputulokseen. Ratkaisu on vielä tulkittava ongelman kontekstissa, jotta todellisuuden reunaehdot tulee huomioitua. Joskus laskutoimitus on niin monimutkainen tai se käsittelee niin paljon dataa, että tarvitaan tietokonetta ongelman ratkaisemiseksi järkevässä ajassa. Matemaattinen malli voidaan ratkaista kokonaan tai osittain tietokoneella. Silloin luodaan tietokoneelle malli ohjelmoimalla, jossa ohjelman suorittaminen automatisoi ratkaisun laskemisen. Ohjelma on algoritmi, johon on lisätty tarvittavat käsitteet tietokoneen ohjaamiseksi. Ohjelman suorituksen tulos voi vielä vaatia matemaattista käsittelyä, ennen kuin sitä hyödynnetään ongelman ratkaisussa.

Työpaja alkoi lyhyellä alustuksella, jossa esitettiin yhteenveto edeltävästä verkkosisällöstä ja käytiin lävitse edellisen työpajan kotitehtävien vastaukset. Alustusta seurasi kolme erillistä osiota, joissa käsiteltiin matemaattisia algoritmeja, algoritmien toteutusta ohjelmoimalla ja ohjelman suorituksen mallintamista.

<u>Matemaattiset algoritmit</u>

• tieto ja taito • laskeminen on algoritmista • käsitteellinen ymmärtäminen kielentämällä • algoritmin luominen matematiikassa	

Algoritmien ohjelmointi

- matemaattiset operaattorit
- muuttujat
- ehtolauseet
- toistolauseet
- tiedon kysyminen käyttäjältä
- ruudulle tulostaminen
- esimerkki: suurin yhteinen tekijä

Kertoma

```
syote = input("Positiivinen kokonaisluku: ")
luku = int(syote)
if luku < 0:
      print("Kertomaa ei ole määritelty.")
elif luku == 0:
      print("Nollan kertoma on 1.")
else:
      print("Positiivinen. Lasketaan...")
      kertoma = 1
      apuluku = luku
   while apuluku > 1:
            kertoma = kertoma*apuluku
            apuluku = apuluku - 1
   print("Kertoma on", kertoma)
```

Ohjelman suorituksen mallintaminen

- ohjelman suoritus noudattaa teknistä mallia
- väärinkäsitykset
- imperatiivisen ohjelmointikielen suorituksen malli
- ohjelman suorituksen (tekninen) visualisointi

Fibonaccin lukujono

```
fa=0
fb=1
n=5
print(fa)
print(fb)
for i in range(1,n+1,1):
    apu=fa+fb
    fa=fb
    fb=apu
    print(fb)
```

muuttuja	
fa	~~0~~ ~~1~~ ~~1~~ ~~2~~ ~~3~~ 5
fb	~~1~~ ~~1~~ ~~2~~ ~~3~~ ~~5~~ 8
n	5
apu	~~1~~ ~~2~~ ~~3~~ ~~5~~ 8

tuloste
0
1
1
2
3
5
8

7 VERKKOSISÄLTÖ 3: KEHITTYNEEMMÄT OHJELMOINNIN KÄSITTEET

Verkkosisällössä jatketaan ohjelmoinnin käsitteistä. Tällä kertaa aiheena ovat aliohjelmat ja listat, joita käsitellään verkkosisällön lopuksi myös ohjelman suorituksen mallintamisen näkökulmasta.

7.1 ALIOHJELMAT

Tähän mennessä kirjoitetut ohjelmat ovat koostuneet yhdestä pääohjelmasta, johon kaikki tarvittavat lauseet on kirjoitettu peräkkäin järjestyksessä. Usein ohjelmissa kuitenkin tarvitaan samaa ohjelmakoodia useaan kertaan, jolloin se on järkevää kirjoittaa omaksi aliohjelmakseen.

Aliohjelmat ovat nimensä mukaisesti pieniä ohjelmia, jotka toteuttavat jonkin yhden tietyn toiminnallisuuden. Ne voidaan jakaa kahteen tyyppiin funktioihin ja proseduureihin, joista funktiot palauttavat jonkin arvon ja proseduurit muuttavat tietokoneen tilaa. Jaottelua ei ole yleensä tehty ohjelmointikielen kieliopissa, joten kyseessä on lähinnä hyvä ohjelmointikäytäntö.

Pythonissa aliohjelmia kutsutaan funktioiksi ja ne toimivat sekä funtioina (palauttaa arvon) että proseduureina (muuttavat tietokoneen tilaa). Esimerkiksi Pythonin omista funktioista `print` toimii proseduurin tavoin, koska se ei palauta kutsukohtaan mitään arvoa, ja `len` taas toimii funktiona, koska se palauttaa arvon.

Aliohjelmat on määriteltävä ennen kuin niitä voi käyttää eli kutsua. Määrittelyllä tarkoitetaan aliohjelman koodin kirjoittamista. Pythonissa aliohjelmat määritellään `def`-lauseella.

Syntaksi:

```
def <nimi>():
    <lohko>
```

Esimerkki:

```
# Määritellään proseduuri,
# joka tulostaa kaksi kertaa Ville.
def tulostaVilleKaksiKertaa():
          print("Ville")
          print("Ville")
# Kutsutaan määriteltyä proseduuria,
# jotta sen lauseet suoritetaan.
tulostaVilleKaksiKertaa()
```

Tulostus:

```
Ville
Ville
```

7.1.1 PARAMETRIT JA PAIKALLISET MUUTTUJAT

Aliohjelmien toimintaan voidaan vaikuttaa parametreilla. Esimerkiksi Pythonin oma funktio `print` ottaa parametrikseen ne arvot, jotka tulostetaan ja jos niitä ei anneta kutsuttaessa, tulostetaan ainoastaan tyhjä rivi.

Aliohjelmien määrittelyssä merkitään muodolliset parametrit sulkujen sisälle. Muodolliset parametrit korvataan todellisilla parametreilla, kun aliohjelmaa kutsutaan.

```
def tulostaSumma(luku1, luku2):
        print(luku1+luku2)

#tulostaa 30
tulostaSumma(10,20)
```

Parametrit ovat siis muuttujia, joita voi käyttää aliohjelmien sisällä samalla tavalla kuin muuttujia pääohjelmassa. Parametrimuuttujat ja kaikki muutkin aliohjelmassa määritellyt muuttujat ovat kuitenkin paikallisia, eikä niitä voi käyttää aliohjelman ulkopuolella. Jokaisella aliohjelmalla on omat paikalliset muuttujat.

Esimerkki:

```
# Määritellään proseduuri,
# joka ottaa parametreina kaksi lukua ja
# tulostaa niiden keskiarvon.
def keskiarvo(luku1, luku2):
          summa = luku1 + luku2
          keskiarvo = summa / 2
          print(keskiarvo)
# Kutsutaan proseduuria eri luvuilla.
keskiarvo(42, 38)
keskiarvo(692, 830)
```

Tulostus:

```
40.0
761.0
```

Esimerkki:

```
# Määritellään proseduuri,
# joka tulostaa kahden luvun summan tuplattuna.
```

```
def tulostaTuplaSumma(luku1, luku2):
    # Tuplataan ensin muuttujien arvot...
    luku1 = luku1 + luku1
    luku2 = luku2 + luku2
    # ... ja tulostetaan sitten summa.
    print(luku1 + luku2)
# Luodaan muuttujat.
luku1 = 5
luku2 = 2
# Kutsutaan proseduuria ja annetaan parametreina luvut.
tulostaTuplaSumma(luku1, luku2)
# Proseduuri käyttää omia paikallisia muuttujiaan,
# joten pääohjelman luvut pysyvät ennallaan.
print(luku1)
print(luku2)
```

Tulostus:

```
14
5
2
```

7.1.2 ARVOJEN PALAUTTAMINEN

Aliohjelmat voivat myös palauttaa arvoja. Tällöin niitä kutsutaan funktioiksi. Arvojen palauttaminen tapahtuu funktion määrittelyssä `return`-lauseella ja arvo palautetaan funktion kutsukohtaan.

Syntaksi:

```
def <nimi>(<parametrit>):
    <lohko>
    return <arvo/lauseke>
```

Esimerkki:

```
# Funktio, joka laskee ja palauttaa luvun kertoman.
def kertoma(luku):
  kertoma = 1
  while luku >= 2:
    kertoma = kertoma * luku
    luku = luku - 1
  return kertoma

# Kutsutaan kertomafunktiota ja
# tallennetaan palautettu arvo muuttujaan.
tulos = kertoma(9)
```

`return`-lauseeseen voi kirjoittaa tietyn arvon, muuttujan tai lausekkeen. Lauseke lasketaan ennen palauttamista eli funktiot palauttavat aina ainoastaan yhden arvon.

Kun arvo palautetaan, siirrytään ohjelman suorituksessa takaisin sinne, mistä funktiota kutsuttiin, joten `return`-lauseen jälkeisiä lauseita ei enää suoriteta. `return`-lauseita voi kuitenkin kirjoittaa funktioon useita. Esimerkiksi ehtolauseita käytettäessä se on usein järkevääkin.

Esimerkki:

```
# Määritellään funktio,
# joka tarkistaa luvun positiivisuuden ja
# palauttaa vastauksen totuusarvona.
def onkoPositiivinen(luku):
    # Tarkistetaan, onko luku suurempi kuin 0.
    if luku > 0:
        # Luku on suurempi kuin 0, joten palautetaan True.
        return True
    else:
        # Tässä haarassa luku ei ollut suurempi kuin 0.
        # Palautetaan siis False.
        return False
```

7.2 LISTAT

Muuttujiin on tähän mennessä tallennettu kokonaislukuja, liukulukuja, merkkijonoja ja totuusarvoja – siis yksi arvo kerrallaan. Ohjelmoidessa usein käsitellään kuitenkin suurempia tietomääriä, joten tarvitaan tietorakenteita, joihin voi tallentaa useampia arvoja.

Useimmin käytetään listoja, joihin voi tallentaa monia arvoja eli alkioita. Listat ovat mutatoituvia (toisin kuin merkkijonot), joten niihin voi lisätä alkioita, niistä voi poistaa alkioita ja alkioita voi myös muuttaa.

Hyvä ohjelmointikäytäntö!

Pythonissa yhteen listaan voi tallentaa sekaisin erityyppisiä arvoja. On kuitenkin hyvän ohjelmointikäytännön mukaista tallentaa ainoastaan yhden tyyppisiä arvoja samaan listaan. Tällöin listan käsittely on helpompaa.

Pythonissa listoja merkitään hakasulkeilla ja alkiot erotetaan toisistaan pilkuilla. Listatkin on tallennettava muuttujaan, jotta niitä voidaan käyttää ohjelmakoodissa myöhemmin.

Esimerkki:

```
luvut = [1, 45, 2, 65, 334, 1234, 16]
nimet = ["Ville", "Heidi", "Marika", "Peter"]
tyhja = []
```

7.2.1 INDEKSOINTI, ALKIOIHIN VIITTAAMINEN JA ALILISTAT

Listan alkiot on indeksoitu eli niillä on järjestysnumerot (indeksit). Indeksointi alkaa nollasta eli ensimmäinen alkio löytyy indeksistä 0.

Indeksien avulla voidaan viitata alkioihin. Viittauksessa käytetään hakasulkeita.

Esimerkki:

```
luvut = [1, 45, 2, 65, 334, 1234, 16]
print(luvut[3])
print(luvut[5])
```

Tulostus:

```
65
1234
```

Hakasulkeiden avulla listasta voidaan myös poimia alilistoja. Tällöin hakasulkeisiin merkitään alkuindeksi ja loppuindeksi erotettuna kaksoispisteillä. Alkuindeksin kohdalla oleva alkio tulee mukaan alilistaan, mutta loppuindeksin ei.

Syntaksi:

```
<listan nimi>[<alkuindeksi> : <loppuindeksi>]
```

Esimerkki:

```
luvut = [1, 45, 2, 65, 334, 1234, 16]
print(luvut[3 : 6])
```

Tulostus:

```
[65, 334, 1234]
```

7.2.2 ALKIOIDEN ARVOJEN MUUTTAMINEN

Indeksien, hakasulkeiden ja asetusoperaattorin avulla alkioiden arvoja voidaan myös muuttaa.

Syntaksi:

```
<listan nimi>[<indeksi>] = <uusi arvo>
```

Esimerkki:

```
luvut = [1, 45, 2, 65, 334]
luvut[1] = 3
luvut[3] = 9
luvut[4] = 2
print(luvut)
```

Tulostus:

```
[1, 3, 2, 9, 2]
```

7.2.3 ALKIOIDEN LISÄÄMINEN

Alkioiden lisääminen tapahtuu kuitenkin erityisellä lisäysmetodilla. Metodit ovat samantapaisia kuin funktiot, mutta ne vaativat olion (eli esimerkiksi listan tai merkkijonon), jolle operaatio suoritetaan. Listan lisäysmetodi on nimeltään append. Metodia kutsutaan laittamalla olion tunnuksen perään piste, sitten metodin nimi ja mahdolliset parametrit sulkeisiin.

Syntaksi:

```
<lista>.append(<lisättävä arvo>)
```

Esimerkki:

```
luvut = [1, 45, 2, 65, 334]
luvut.append(1234)
luvut.append(16)
print(luvut)
```

Tulostus:

```
[1, 45, 2, 65, 334, 1234, 16]
```

7.2.4 ALKIOIDEN POISTAMINEN

Listasta poistetaan alkioita poistometodien avulla. Alkioita voi poistaa sekä indeksin perusteella että arvon perusteella.

Indeksin perusteella poistettaessa käytetään metodia pop.

Syntaksi:

```
<listan nimi>.pop(<indeksi>)
```

Esimerkki:

```
luvut = [1, 45, 2, 65, 334, 1234, 16]
luvut.pop(0)
luvut.pop(3)
print(luvut)
```

Tulostus:

```
[45, 2, 65, 1234, 16]
```

Alkioita voi poistaa myös niiden arvon perusteella. Tällöin käytetään metodia `remove` ja poistettava alkio on listasta ensimmäinen, jonka arvo vastaa parametrin arvoa.

Syntaksi:

```
<listan nimi>.remove(<arvo>)
```

Esimerkki:

```
luvut = [1, 45, 2, 65, 334, 1234, 16]
luvut.remove(1234)
luvut.remove(45)
print(luvut)
```

Tulostus:

```
[1, 2, 65, 334, 16]
```

7.2.5 LISTOJEN ITEROINTI

Usein ohjelmoinnissa on tarpeellista käydä läpi kokonaisia listoja. Ohjelmoinnissa puhutaan listojen iteroimisesta. `for`-lauseella listoja voidaan käydä läpi samalla tavalla kuin merkkijonoja.

Syntaksi:

```
for <alkio> in <lista>:
    <lohko>
```

Esimerkki:

```
luvut = [1, 45, 2, 65, 334, 1234, 16]
for luku in luvut:
        print(luku)
```

Tulostus:

```
1
45
2
65
334
1234
16
```

Myös `while`-lausetta ja `for`-lausetta `range`-funktion kanssa on mahdollista käyttää. Tällöin kuitenkin täytyy tietää listan pituus eli listan alkioiden lukumäärä. Pythonissa se selviää `len`-funktiolla.

Syntaksi:

```
len(<lista/listan nimi>)
```

Esimerkki:

```
luvut = [1, 45, 2, 65, 334, 1234, 16]
pituus = len(luvut) # 7
```

Listan viimeinen alkio

Koska listojen indeksit alkavat nollasta, eikä ykkösestä, löytyy listan alkio kohdasta len(lista)-1. Eli jos esimerkiksi halutaan tulostaa listan viimeinen alkio:

print(lista[len(lista)-1])

Kun käytetään `while`-lausetta tai `for`-lausetta `range`-funktion kanssa, ei oikeastaan käydä läpi listan alkioita vaan listan indeksit.

Syntaksi:

```
for indeksi in range(0, len(<lista>), 1):
        <lohko>

indeksi = 0
while indeksi < len(<lista>):
        <lohko>
```

```
        indeksi = indeksi + 1
```

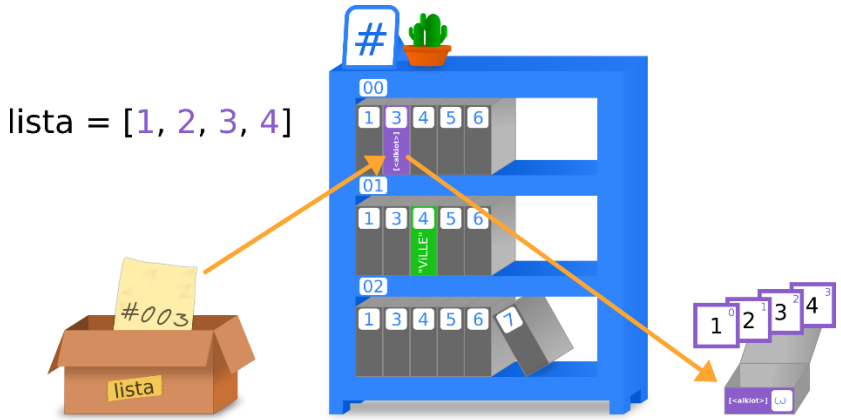
Esimerkki:

```
# Funktio kopioi listan alkiot uuteen listaan
# ja palauttaa sen.
def kopioiLista(lista):
    uusiLista = []
    indeksi = 0
    while indeksi < len(lista):
        uusiLista.append(lista[indeksi])
        indeksi = indeksi + 1
    return uusiLista
```

7.2.6 LISTOJEN VIITTAUSTYYPPISYYS

Vaikka listamuuttujia voi käyttää pääasiassa samaan tapaan kuin perustyyppisiä muuttujia, ne kuitenkin tallennetaan tietokoneen muistiin eri tavalla. Perustyyppisien muuttujien tapauksessa muuttujaan tallennetaan muuttujan arvo.

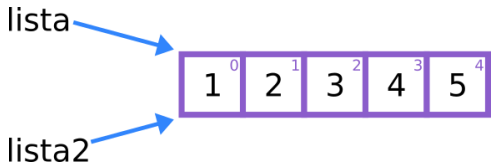
Listamuuttujien tapauksessa muuttujaan tallennetaan viittaus tietokoneen muistiin. Näin ollen itse alkiot löytyvät vasta viittauksen kautta muualta tietokoneen muistista.



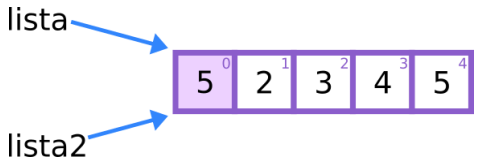
Listojen viittaustyyppisyyden takia listan kopioiminen täytyy tehdä iteroimalla ja kopioimalla yksi arvo kerrallaan uuteen listaan, koska muutoin kopioidaan vain viittaus.

Esimerkki:

```
lista = [1, 2, 3, 4, 5]
lista2 = lista
```




```
lista[0] = 5
```




```
print(lista2[0]) # 5
```

Listojen viittaustyyppisyys vaikuttaa myös listoja käsitteleviin funktioihin. Alkion muuttamiseksi on asetettava siihen uusi arvo käyttämällä listan nimeä ja alkion indeksiä.

<u>Esimerkki:</u>

```
# Kasvatetaan alkuperäisen listan alkioita.
def kasvataAlkioita(lista):
    for indeksi in range(0, len(lista), 1):
        lista[indeksi] = lista[indeksi] + 1
    # Palautusta ei tarvita,
    # koska muutetaan alkuperäistä listaa.

# Luodaan uusi lista, jonka alkioita on kasvatettu.
def uudetKasvatetutAlkiot(lista)
    uusiLista = []
    for luku in lista:
        uusiLista.append(luku+1)
    # Tarvitaan palautus, koska luotiin uusi lista.
    return uusiLista

luvut1 = [15, 16, 17, 18]
luvut2 = [15, 16, 17, 18]

# Muutetaan ensimmäistä listaa.
kasvataAlkioita(luvut1)

# Muutetaan toista listaa.
# Nyt uusi lista täytyy tallentaa uuteen muuttujaan.
uudetLuvut = uudetKasvatetutAlkiot(luvut2)

print(luvut1)
print(luvut2)
print(uudetLuvut)
```

Tulostus:

```
[16, 17, 18, 19]
[15, 16, 17, 18]
[16, 17, 18, 19]
```

7.3 KIRJASTOT

Kirjasto on kokoelma ominaisuuksia: funktioita, luokkia ja vakioarvoja. Osa kirjastoista ovat ohjelmointikielessä vakiona eli tulevat sen mukana. Ohjelmointikieliin löytyy myös kirjastoja, joita voi itse lisätä valmiiden ratkaisujen tai uusien ominaisuuksien hyödyntämiseksi. Kirjasto avataan ja sen jälkeen kirjaston ominaisuuksia pääsee käyttämään. Avaamiseen on kaksi tapaa ja valittu tapa vaikuttaa siihen, miten kirjaston ominaisuuksia käytetään myöhemmin.

Kirjaston avaaminen, tapa 1

Avataan koko kirjasto, jolloin ominaisuuksia käytettäessä täytyy viitata kirjastoon.

```
# Avataan kirjasto import-lauseella.
import math

# Käytetään ominaisuuksia viittaamalla ensin kirjastoon.
print(math.pi)
```

Kirjaston avaaminen, tapa 2

Avataan kirjastosta jokin tietty ominaisuus, jolloin kyseistä ominaisuutta voidaan käyttää viittaamatta kirjastoon.

```
# Avataan kirjastosta tietty ominaisuus.
from math import pi

# Käytetään ominaisuutta viittaamatta kirjastoon.
print(pi)
```

`math`-kirjasto sisältää paljon matemaattisia ja numeerisia ominaisuuksia (https://docs.python.org/3/library/math.html).

`random`-kirjasto sisältää satunnaisoperaatioihin liittyviä ominaisuuksia. Huomaa että kirjaston ominaisuuksiin ei kuulu todellista satunnaisuutta! Arvot lasketaan kaavalla, joka käyttää siementä. Siemenen voi asettaa metodilla `random.seed(x)`. Samaa siementä käyttämällä saa samoja näennäisen satunnaisia arvoja (https://docs.python.org/3/library/random.html).

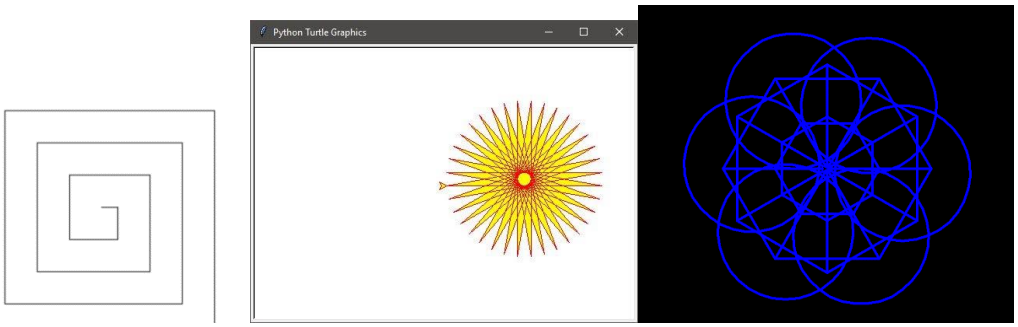
Ominaisuus	Merkitys
`math.sqrt(x)`	palauttaa x:n neliöjuuren

`math.cos(x)`
`math.sin(x)`
`math.tan(x)`
	palauttaa x:n kosinin/sinin/tangentin, kun x annetaan radiaaneina

`math.degrees(x)` `math.radians(x)`	palauttaa x:n asteina/radiaaneina, kun x annetaan radiaaneina/asteina
`math.pi`	palauttaa piin likiarvon
`random.random()`	palauttaa desimaaliluvun (float) väliltä 0.0 ja 1.0
`random.randint(a, b)`	palauttaa satunnaisen kokonaisluvun N niin, että a <= N <= b
`random.choice(lista)`	palauttaa satunnaisen alkion annetusta listasta (aiheuttaa virheen, jos lista on tyhjä)
`random.seed(a)`	asettaa generoinnille siemenen a: samalla siemenellä arvotut luvut ovat aina samat (jos siementä ei aseteta, käytetään tietokoneen aikaa)

7.4 KILPIKONNAGRAFIIKKA

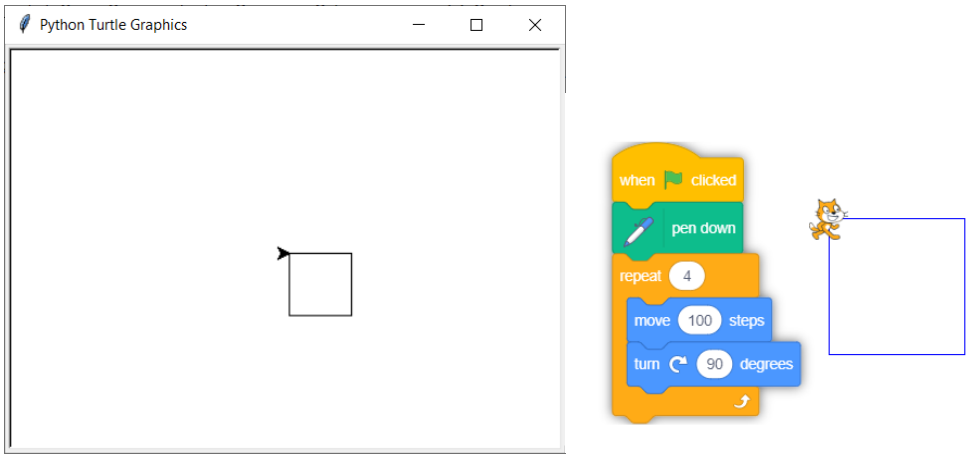
Kilpikonnagrafiikka (*Turtle graphics*) on Pythonin kirjasto, jonka avulla voidaan piirtää geometrisia kuvioita ohjelmoimalla (https://docs.python.org/3/library/turtle.html).



Kilpikonnagrafiikan käyttö alkaa turtle-kirjaston avaamisella. Sen jälkeen ohjelmassa voidaan kutsua kirjaston ominaisuuksia. Turtle-kirjasto sisältää ominaisuuksia mm. liikkumiseen ja värittämiseen. Toimintaa voi verrata alakoulussa yleisesti käytettyyn Scratch-kieleen (https://docs.python.org/3/library/turtle.html).

Esimerkki:

```
from turtle import *

#piirretään neliö
for i in range(4):
forward(50)
right(90)
```



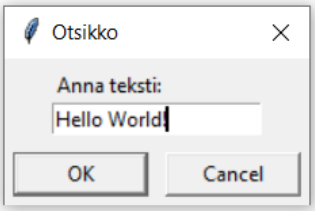
Lisätieto: Voit nimetä kaksi kilpikonnaa ja tehdä ohjelman, joka ohjaa niitä:

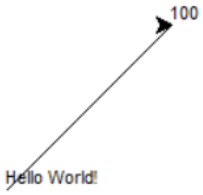
```
from turtle import *
from random import randint, choice

eka=Turtle()
toka=Turtle()
varit=["red", "green", "blue", "cyan", "pink"]
bgcolor("gray")

for i in range(40):
    eka.forward(randint(30,50))
    toka.forward(randint(30,50))
    eka.right(randint(40,60))
    toka.left(randint(40,60))
    eka.pencolor(choice(varit))
```

Kilpikonnagrafiikka näytetään omassa ikkunassaan. Ikkunassa on mahdollista pyytää käyttäjän syöte ja tulostaa tekstiä. Teksti tulostuu kilpikonnan sijainnin mukaan.

```
from turtle import *

teksti = textinput("Otsikko", "Anna teksti: ")
luku = int(numinput("Otsikko", "Anna luku: "))
write(teksti)
```

```
goto(100,100)
write(luku)
```

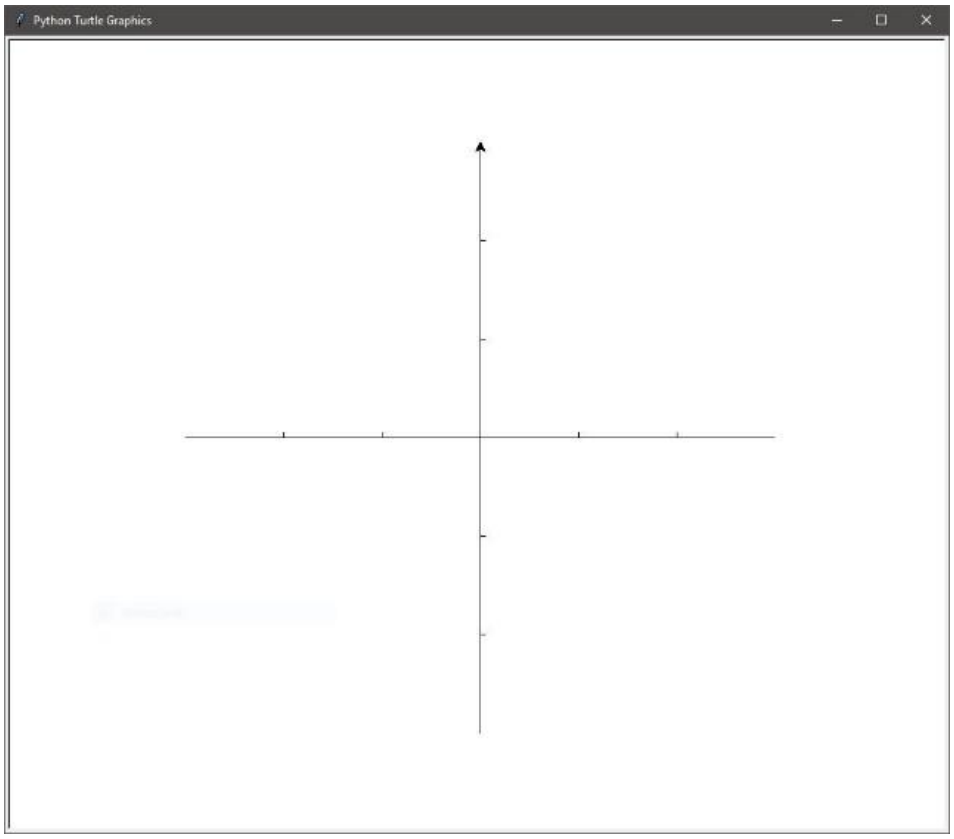




Ominaisuus	**Merkitys**
`forward(a)` `backward(b)`	liiku eteenpäin a pikseliä / taaksepäin b pikseliä
`right(a)` `left(b)`	käänny oikealle a astetta / vasemmalle b astetta
`set-heading(a)`	aseta suunta (0: itä, 90: pohjoinen, 180: länsi, 270: etelä)
`circle(r, e)`	piirtää ympyrän, jonka säde on r jos annetaan kulma e, piirretään kaari
`speed(a)`	aseta kilpikonnan nopeus (“fastest”: 0, “fast”: 10, “normal”: 6, “slow”: 3, “slo-west”: 1)
`pendown()` `penup()`	nostaa tai laskee kynän (nostettuna viivoja ei piirretä, kun liikutaan)
`pensize(x)`	aseta kynän paksuus
`pencolor(x)`	aseta kynän väriksi x (esim. “red”, “blue”, “green” jne.)
`bgcolor(x)`	aseta taustan väriksi x (esim. “red”, “blue”, “green” jne.)

Kilpikonnagrafiikka käyttää koordinaatistoa, vaikka oletuksena ei ruudukkoa näytetäkään.

Ominaisuus	**Merkitys**
`goto(x, y)`	liiku koordinaatteihin (x, y)
`setx(x)` `sety(y)`	aseta x- tai y- koordinaatti
`xcor()` `ycor()`	palauta x- tai y-koordinaatti



7.5 OHJELMAN SUORITUKSEN MALLINTAMINEN

Ohjelman suoritus noudattaa teknistä mallia. Ohjelmointikielen malli määrittelee miten se ohjaa tietokonetta. Mallit ovat tietokoneen toiminnan abstraktioita ja ne vaihtelevat matemaattisista teknisiin. Yleisin malli kuvaa tietokoneen muistin käsittelyä ja on käytössä niin kutsutuissa imperatiivisissa kielissä, kuten Python, Java, C… Malli tukee matemaattisten lauseiden laskemista. Lauseiden ulkopuolella säännöt ovat kuitenkin tekniset. Matemaattiset oletukset voivat aiheuttaa oppilaille väärinkäsityksiä. Ohjelman suorittaminen yksinkertaistetulla mallilla tukee väärinkäsitysten ehkäisemistä.

Tyypillisiä väärinkäsityksiä ovat esimerkiksi muuttujan käyttäytymiseen liittyvät virheet. Tässä esimerkissä `ala`-muuttujan arvo ei muutu, vaikka sen laskemisessa käytettyä `sade`-muuttujaa muutetaan jälkeenpäin. Jos `ala`-muuttujan arvoa halutaan muuttaa, on se tehtävä uudella asetuslauseella.

Esimerkki:

```
sade=10
ala=3.14*sade**2
sade=20
print(ala) #mitä tulostetaan?
```

Toisessa esimerkissä silmukkaa ei suoriteta kertaakaan, koska `while`-lauseeseen tultaessa muuttujan `i` arvo on suurempi kuin 10, eikä enää 0, kuten ohjelman alussa.

Esimerkki:

```
i=0
i=100
while i<10: #suoritetaanko silmukka?
          print(i)
          i=i+1
```

Esitetään imperatiivisen ohjelmointikielen suorituksen malli, joka perustuu muistin käsittelyyn. Muisti kuvataan taulukkona, jossa vasemmalla ovat muuttujien nimet ja oikealla nimiä vastaavat arvot. Uuden muuttujan arvo laitetaan edellisen alapuolelle. Jos muuttujan arvoa muutetaan, yliviivataan edellinen arvo ja laitetaan sen vierelle uusiarvo. Huomaa että muuttujassa on vain yksi arvo kerrallaan. Muistin lisäksi mallinnettaessa laitetaan tulosteet (`print`-lause) järjestyksessä Tuloste -otsikon alle.

Muuttujan arvon asettaminen

```
s = "hello"
x = 4
y = 6
x = 8
total = x + y + x
print(total)
```

muuttujan nimi	arvo
s	“hello”
x	~~4~~ 8
y	6
total	22

Tuloste
22

7.5.1 ALIOHJELMIEN ESITTÄMINEN MUISTISSA

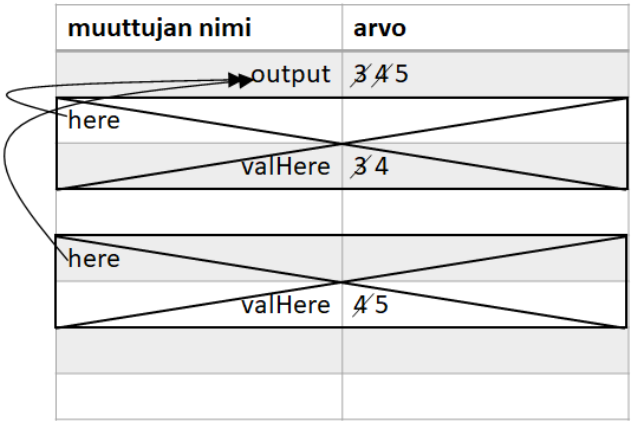
Muisti kuvataan nyt taulukkona, johon merkitään ohjelmakoodin määrittelemät muuttujat järjestyksessä ylhäältä alas.

Myös aliohjelmien (funktiot ja proseduurit) muuttujat tallennetaan samaan muistiin kuin muutkin. Jotta voisimme pitää kirjaa mallinnettaessa, mitkä muuttujat ovat aliohjelmien muuttujia, merkitsemme ne taulukkoon omaan aliohjelman nimellä nimetyn laatikon sisälle. Jos aliohjelma on funktio, eli palauttaa tuloksen, piirrämme sen laatikosta nuolen muuttujaan, johon sen tulos tallennetaan. Kun aliohjelman suoritus on päättynyt, poistetaan se muistista vetämällä rasti sitä taulukossa edustavan laatikon ylitse. Jos aliohjelmaa kutsutaan ohjelmassa uudelleen, piirretään uusi laatikko sen suoritusta varten.

HUOM! Tulostus aliohjelmasta merkitään suorituksen mallintamisessa saman Tuloste -otsikon alle, kuin muutkin `print`-lauseiden tulosteet ohjelmassa.

Aliohjelman kutsuminen

```
def here(valHere):
        valHere = valHere + 1
        return valHere

output = 3
output = here(output)
output = here(output)
```



Tuloste

7.5.2 LISTOJEN ESITTÄMINEN MUISTISSA

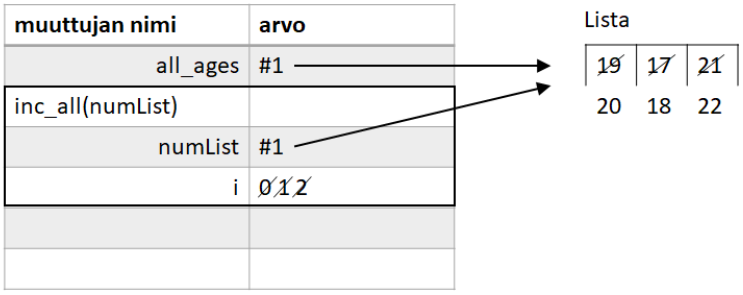
Listat poikkeavat muuttujien perusarvoista. Ne kuvaavat tietorakennetta eli ne voivat sisältää useamman arvon. Tämä näyttäisi olevan ristiriidassa muuttujien luonteen kanssa (vain yksi arvo), mutta tämä on ratkaistu niin, että muuttujan arvona on yksi viittaus, jonka osoittamassa muistipaikassa on tallennettu lista-tietorakenne. Lista on poikkeava muuttujien perusarvoista myös siinä, että listan sisältöä voi muuttaa (viittaus listaan pysyy kuitenkin samana).

Ohjelman suorituksen mallintamisessa laitetaan taulukkoon muuttujan arvoksi risuaita ja juokseva numero edustamaan viittausta tiettyyn listaan. Lista piirretään erikseen taulukon ulkopuolelle ja viittauksesta piirretään nuoli siihen. Samaan listaan voi viitata monta muuttujaa. Niillä kaikilla on sama viittaus

(risuaita ja numero) arvona. Kaikkien samaan listaan viittaavien muuttujien kautta voidaan muokata listan sisältöä. Muutokset listaan merkitään mallinnettaessa piirtämällä listan vanhan alkion päälle viiva ja uusi alkio kyseisen alkion alle. Näin listan alkioiden päivittäminen vastaa muuttujien päivittämistä, mutta suunta on ylhäältä alaspäin.

Listojen käsittely

```
def inc_all(numList):
        for i in range(len(numList)):
                numList[i] = numList[i] + 1

all_ages = [19, 17, 21]
inc_all(all_ages)
```



Tuloste

_

8 TYÖPAJA 3: KIRJASTOT, KILPIKONNAGRAFIIKKA SEKÄ ALIOHJELMAT JA LISTAT OHJELMAN SUORITUKSESSA

Työpajan aiheina olivat Pythonin kirjastot ja edistyneempien ohjelmoinnin käsitteiden suorituksen mallintaminen. Ohjelmointikielet koostuvat käsitteistä ja niiden käyttöä ohjaavista säännöistä. Monipuolisemmat ominaisuudet rakennetaan peruskäsitteistä ja ohjelmointikielen mukana tuleekin yleensä ohjelmistokirjastoja, jotka tarjoavat valmiita ratkaisuja usein esiintyviin ongelmiin. Yksi kirjasto koostuu tiettyyn aihepiiriin liittyvistä funktioista, luokista ja vakioista. Jokainen voi luoda omia kirjastoja, joten tarjolla on ohjelmointikielen mukana tulevien lisäksi monia sekä ilmaisia että kaupallisia kirjastoja. Ohjelmistokirjastoissa keskityttiin Python-kielen mukana tuleviin matemaattisiin kirjastoihin. Työpajassa jatkettiin myös ohjelman suorituksen mallintamista aliohjelmilla ja listoilla. Aliohjelma määrittelee omat muuttujansa, jotka ovat olemassa vain sen suorituksen ajan. Tavallisten muuttujien lisäksi merkitään taulukkoon myös aliohjelman syötteet. Listoilla on muistia kuvaavassa taulukossa vain viittaus omaan taulukkoon, jonka sarakkeet edustavat listan alkioita. Kun listan alkioita muutetaan, lisätään ne uudelle riville alkuperäisen alkion alle. Rivissä voi olla myös uusia sarakkeita tai niitä voi olla vähemmän riippuen, miten listaa on muokattu.

Työpaja alkoi lyhyellä alustuksella, jossa esitettiin yhteenveto edeltävästä verkkosisällöstä ja käytiin lävitse edellisen työpajan kotitehtävien vastaukset. Alustusta seurasi kolme erillistä osiota, joissa käsiteltiin Pythonin kirjastojen käyttöä, kilpikonnagrafiikka-kirjaston käyttämistä geometrian opetuksessa ja lopuksi aliohjelmat sekä listat ohjelman suorituksen mallintamisessa.

<u>Ohjelmistokirjastot</u>

- kirjasto on kokoelma ominaisuuksia
- kirjastot voivat tulla ohjelmointikielen mukana tai ovat ohjelmointikielen käyttäjien kehittämiä
- otetaan käyttöön kahdella eri tavalla
  - avataan koko kirjasto
  - tuodaan yksittäisiä piirteitä
- math-kirjasto
  - matemaattiset funktiot
- random-kirjasto
  - pseudo-satunnaislukujen luonti

Kirjaston avaaminen: tapa 1

- Avataan koko kirjasto, jolloin ominaisuuksia käytettäessä täytyy viitata kirjastoon.

```
# Avataan kirjasto import-lauseella.
import math

# Käytetään ominaisuuksia viittaamalla ensin kirjastoon.
print(math.pi)
```

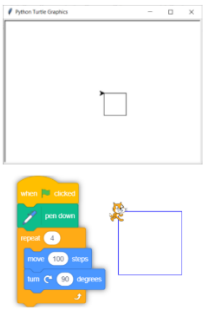
Kilpikonnagrafiikka

- Turtle graphics -kirjaston esittely
- kilpikonnagrafiikan käyttö
- kilpikonnan ominaisuuksia
- käyttäjän syöte kilpikonnan ikkunassa
- kilpikonna koordinaatistossa

Kilpikonnagrafiikan käyttö

```
from turtle import *

#piirretään neliö
for i in range(4):
  forward(100)
  right(90)
```

vertaa alakoulussa usein käytettyyn Scratchiin:



Ohjelman suorituksen mallintaminen

- aliohjelman suorituksen mallintaminen
  - merkitään taulukkoon omana laatikkona
  - omat muuttujat
  - olemassa vain aliohjelman suorituksen ajan
- listan käsittelyn suorituksen mallintaminen
  - vain viittaus lista-tietorakenteeseen
  - samaan listaan voi viitata useampi muuttuja
  - viittaus merkitään risuaidalla ja juoksevalla numeroinnilla

Tehtävä 6: ratkaisu

```
# Määritellään proseduuri,
# joka ottaa parametreina kaksi lukua ja
# tulostaa niiden keskiarvon.
def keskiarvo(luku1, luku2):
        summa = luku1 + luku2
        keskiarvo = summa / 2
        print(keskiarvo)

# Kutsutaan proseduuria eri luvuilla.
keskiarvo(10, 20)
keskiarvo(360, 440)
```

muuttujan nimi	arvo
luku1	10
luku2	20
summa	30
keskiarvo	15.0
luku1	360
luku2	440
summa	800
keskiarvo	400.0

Tuloste
```
15.0
400.0
```

9 VERKKOSISÄLTÖ 4: OHJELMOINNILLISEN AJATTELUN SOVELTAMINEN MATEMATIIKAN OPETUKSESSA

Ohjelmointi poikkeaa monesta kouluaineesta, koska se sisältää sekä matemaattista että teknistä ajattelua. Opettajalle tämä asettaa haasteen, miten saada oppilaat yhdistämään matemaattisen eksaktin kuvauksen tietokoneen ominaisuuksien hyödyntämiseen. Matematiikan opettaja joutuu vielä lisäksi miettimään, miten matematiikan ja ohjelmoinnin opetus voisi tukea toisiansa. Ensimmäinen ohjelmoinnin opetusmenetelmä vastaa näihin molempiin kysymyksiin jakamalla ohjelmoinnin opetuksen neljään erilaiseen harjoitustyyppiin ja tarjoamalla esimerkin, miten näitä voidaan soveltaa myös matematiikan sisältöihin. Toinen ohjelmoinnin opetusmenetelmä keskittyy erityisesti tekstipohjaisen ohjelmoinnin oppimisen haasteisiin. Verkkosisällössä esitellään myös kehittyneempää Pythonin matematiikkakirjastojen käyttöä sekä isomman matemaattisaiheisen ohjelman luontia aliohjelmista koostamalla.

9.1 OHJELMOINNIN OPETTAMISEN MENETELMÄT 1

9.1.1 OPETUSMENETELMIEN TEOREETTISTA TAUSTAA

"The role of the teacher is to create the conditions for invention rather than provide ready-made knowledge."

Seymour Papert

Yllä oleva lainaus antaa suuntaviivoja ohjelmoinnillisen ajattelun soveltamiseen matematiikassa. Sen sijaan että ajateltaisiin uutta oppimäärää, joka pitää mahduttaa matematiikan opetuksen lomaan, voidaan tarttua mahdollisuuteen ja tuoda matematiikan opetukseen uusi näkökulma.

Tietojenkäsittelytieteilijä ja matemaatikko Seymour Papert tunnetaan teknologian opetuskäytön kehittäjänä. Hän jatkoi Jean Piaget'n konstruktivistista ajattelua ohjelmoinnin ja teknologian piirissä ja oli tutkivan oppimisen edelläkävijä. Papert esitti, että Piaget'n ajattelun mukainen konstruktivistinen oppiminen toteutuu parhaiten, kun oppija työskentelee oman konkreettisen tuotoksensa parissa ja jakaa sekä tuloksen että prosessin muiden kanssa. Tämä konstruktionismiksi kutsuttu oppimistapa ei ole pelkästään tekemällä oppimista, vaan mukana on myös vahva sosiaalinen ulottuvuus. Papert kehitti jo 1970-luvulla käytännön työkaluksi opetusta varten Logo-ohjelmointikielen, jonka seuraaja Pythonin Turtle-kirjastokin on. Myös monille tutun visuaalisen ohjelmointikielen Scratchin taustalla ovat samat omaan luomiseen ja vertaisryhmälle jakamiseen liittyvät konstruktionistiset ajatukset.

Piaget	**Papert**	
Konstruktivismi →	Konstruktionismi →	Logo Python / Turtle Scratch
”...oppija sovittaa aktiivisesti tiedon hänellä jo oleviin tietorakennelmiin..."	"...konstruktivistinen oppiminen toteutuu, kun oppija luo ja jakaa konkreettisia itselleen merkityksellisiä artefakteja…"	

Ohjelmointi soveltuu hyvin myös matemaattisten ongelmien tutkimiseen. Papertin hengessä matemaattista ongelmaa voidaan lähteä tutkimaan oppilaalle merkityksellisten omien tavoitteiden kautta. Oppilas luo oman toteutuksensa, ohjelman, jonka toimintaa hän voi testata ja jota voidaan tutkia vertaisryhmässä tai luokassa. Iteratiivisuus ja vertaispalaute ovat tärkeässä roolissa ja sopivat hyvin matemaattiseen ongelmanratkaisuun, jossa miltei aina on useampia oikeita ratkaisuja tai useita tapoja päästä ratkaisuun.

Ohjelmoinnillinen ajattelu on taito, joka ohjelmoijalle kehittyy, jotta hän voi hallita laajempia kokonaisuuksia ja ratkaista ongelmia systemaattisesti. Ohjelmoinnillisen ajattelun oppimiseen ja hyödyntämiseen liittyy kiinteästi itse tekeminen ja kokeilu. Useat tutkijat ovat ehdottaneet malleja Papertin konstruktionististen periaatteiden mukaiseen ohjelmoinnillisen ajattelun opettamiseen. Tässä osiossa tutustumme matematiikan opettaja ja professori Donna Kotsopouloksen kehittämään ohjelmoinnillisen ajattelun pedagogiseen viitekehykseen ja seuraavassa osiossa Raspberry Pi -säätiön tutkijan Sue Sentancen johdolla kehitettyyn PRIMM-opetusmetodiin.

9.1.2 OHJELMOINNILLISEN AJATTELUN PEDAGOGINEN VIITEKEHYS

Matematiikan opetusta tutkinut professori Donna Kotsopoulos työryhmineen on suunnitellut pedagogisen viitekehyksen ohjelmoinnilliseen ajatteluun (Computational Thinking Pedagogical Framework). Siihen kuuluu neljä aluetta: 1) toiminnallinen oppiminen, 2) kokeileminen, 3) tekeminen ja 4) soveltava yhdistely. Voi olla vaikeaa käyttää tutkimustietoa suoraan omassa opetuksessa. Viitekehyksen tarkoitus on auttaa opetuksen suunnittelussa ja ohjata hyödyntämään taustalla olevaa konstruktionistista ajattelua. Suomalaisessa opetussuunnitelmassa ohjelmoinnillinen ajattelu yhdistetään matematiikan (ja käsityön) opetukseen ja tavoitteena on ongelmanratkaisutaidon kehittäminen. Esimerkeissä annetaan vinkkejä viitekehyksen soveltamiseen suomalaisessa yläkoulussa.

Computational Thinking Pedagogical Framework

1. *Unplugged*
2. *Tinkering*
3. *Making*
4. *Remixing*

9.1.2.1 UNPLUGGED, TOIMINNALLINEN OPPIMINEN

Viitekehyksen ensimmäinen alue on toiminnallinen oppiminen. Tutkijat ovat ehdottaneet, että ohjelmointia tai ohjelmoinnillista ajattelua opettaessa voidaan hyödyntää myös toiminnallista oppimista, ilman tietokonetta. Toiminnallinen osio, usein yhteistoiminnallinen tai/ja kinesteettinen, auttaa herättämään oppijan mielenkiinnon ja onnistuminen motivoi jatkamaan asiaan perehtymistä. Lisäksi toiminnallisuus asettaa opeteltavan asian oppijalle tuttuun kontekstiin. Esimerkkejä toiminnallisista ohjelmoinnillisen ajattelun harjoituksista löydät esimerkiksi vapaasti käytettävältä, useampien yliopistojen yhteistyönään kehittämältä CS Unplugged -sivustolta. Matematiikan opetuksessa helppo lähtökohta toiminnallisuuteen on aloittaa tutulla kynä-paperi-menetelmällä, vaikka tämä ei ole täysin viitekehyksen ensimmäisen osuuden suuntaviivojen mukainen tapa.

9.1.2.2 TINKERING, KOKEILEMINEN

Kokeiluosuudessa tutkitaan miten jokin olemassa oleva kokonaisuus, fyysinen tai digitaalinen, toimii. Oppijan ei tarvitse luoda uutta, vaan hän voi tarkastella miten olemassa olevaa voi vaihe vaiheelta muuttaa ja havainnoida, mitä seurauksia muutoksilla on. Tavoitteena on houkutella oppija pohtimaan erilaisia mahdollisuuksia ja huomaamaan miten voi itse vaikuttaa kokonaisuuden toimintaan. Hyvin tyypillinen ja erinomaisesti matematiikan opetukseen sopiva esimerkki on valmiin ohjelmakoodin tutkiminen. Oppija voi saada valmiin ohjelmakoodin ja hänelle voidaan osoittaa rivi, jota muokkaamalla koodi esimerkiksi piirtää erilaisia geometrisia kuvioita.

9.1.2.3 MAKING, TEKEMINEN

Tekeminen tulisi aloittaa alusta, ilman valmista mallia, jotta oppija voi suunnitella, valita työtavan, ja ylipäätään yhdistellä jo osaamiaan asioita. Tutkijat ottavat tässä yhteydessä esille “käsin kosketeltavat” *(tangible)* tuotokset, kuten Papertin kilpikonnarobotin sekä Micro:bitin tai Arduinon tapaiset korttitietokoneet. Päätavoitteena on kuitenkin tarjota oppijalle kognitiivisesti edellisiä vaativampi osuus. Jos huomioidaan opetusresurssit ja se että oppilaat todennäköisesti yläkoulun aikana vasta opettelevat ohjelmointityökalun käyttöä, voidaan ajatella, että osuuden tarkoituksena on esimerkiksi oman ohjelman laatiminen ohjatusti annetuista osista tai kommenttien avulla ohjelmoiden.

9.1.2.4 REMIXING, SOVELTAVA YHDISTELY

Soveltava yhdistely on enemmän kuin pelkkä valmiiden osien yhdistäminen. Oppijan pitää tunnistaa muiden tuotosten keskeisimmät toiminnallisuudet ja osata yhdistää näitä uuden, erilaisen tuotoksen laatimiseksi. Tämä on viitekehyksen osioista kognitiivisesti vaativin. Osien uudelleenkäyttö liittyy vahvasti ohjelmointiin. Minkä tahansa ohjelmiston koodia on paljon, eikä sitä kirjoiteta alusta alkaen, vaan hyödynnetään kirjastoja ja aiempia versioita. Viitekehyksen laatijat esittävät, että tässä yhteydessä tarjoutuu ti-

laisuus myös eettisten, kopiointiin ja digitaaliseen sisältöön liittyvien kysymysten käsittelyyn. Yksinkertaisimmillaan opetuksessa soveltava yhdistely voisi liittyä eri ongelman ratkaisemiseen aiempaa ohjelmaa täydentämällä.

Näihin neljään alueeseen kuuluu erillisiä ja lomittuvia ohjelmoinnillisen ajattelun elementtejä, jotka yhdessä antavat oppilaalle ohjelmoinnillisen ajattelun kokemuksen. Erityisesti aloittelijoiden on hyvä nähdä alueet tässä järjestyksessä suoritettavina vaiheina.

Viitekehys on kokonaisuudessaan kuvattuna Digital Experiences in Mathematics Education -julkaisussa, vuonna 2017 ilmestyneessä artikkelissa "A Pedagogical Framework for Computational Thinking". Artikkeli on saatavilla osoitteessa: utouch.cpsc.ucalgary.ca/docs/PedagogicalFramework-Springer2017.pdf.

Tutkitaan esimerkin avulla Kotsopouloksen viitekehyksen soveltamista matematiikan opetuksen yhteydessä tapahtuvaan ohjelmoinnin opetukseen ja ohjelmoinnillisen ajattelun käyttöä matematiikan tehtävien yhteydessä. Vertaa tyypillistä kirjan tehtävää ja ohjelmoinnin avulla toteutettua tutkivaa konstruktionistista lähestymistapaa.

9.1.2.5 TYYPILLINEN KIRJAN TEHTÄVÄ JA SEN TUTKIMINEN OHJELMOIMALLA

Ympyräkartion pohjan halkaisija on 5 cm ja kartion korkeus 15 cm. Mikä on kartion tilavuus?

Ratkaisu:

$d = 5\ cm, h = 15\ cm$

$$A = \pi \cdot r^2 = \pi \cdot \left(\frac{d}{2}\right)^2 = \pi \cdot d^2$$

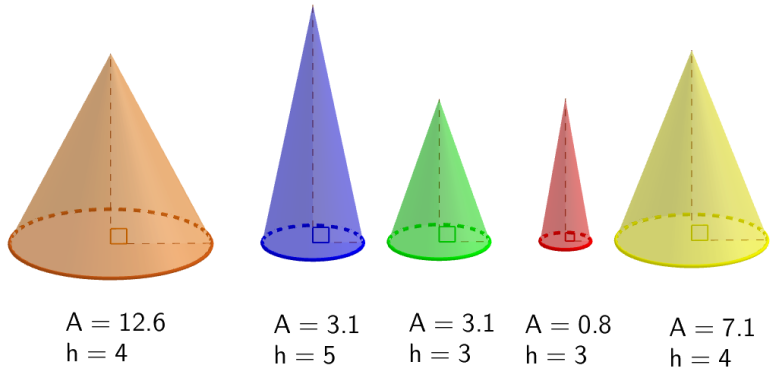
$$V = \frac{1}{3} \cdot A \cdot h$$
$$= \frac{1}{3} \cdot \frac{\pi}{4} \cdot d^2 \cdot h$$
$$= \frac{\pi \cdot d^2 \cdot h}{12}$$
$$= \frac{3{,}14 \cdot (5\ cm)^2 \cdot (15\ cm)}{12}$$
$$= 98\ cm^3$$

Vastaus: Kartion tilavuus on $98\ cm^3$

Tutkiva lähestymistapa ohjelmoimalla

Tutkitaan ympyräkartion tilavuutta.

a) Laske seuraavien kartioiden tilavuudet. Täydennä annettu ohjelma. Muokkaa ohjelmaasi siten, että saat näiden kartioiden tilavuudet laskettua.




```
#pohjan pinta-ala MUOKKAA
A=1.0
#kartion korkeus MUOKKAA
h=1.0
#kartion tilavuus LASKE
V=1.0
print("Pohjan pinta-ala:", A, "kartion korkeus:", h, "kartion tila-
vuus:", V)
```

b) Laadi ohjelma, joka laskee kartion tilavuuden, kun käyttäjä antaa kartion pohjan halkaisijan ja korkeuden. Erään kartion pohjan halkaisija on 4 ja korkeus 4. Sen tilavuus on yhden desimaalin tarkkuudella 16,8. Testaa toimiiko ohjelmasi oikein.

c) Hyödynnä b-kohdan ohjelmaa ja laadi ohjelma, joka laskee kartion korkeuden, kun pohjan halkaisija ja tilavuus tiedetään. Testaa ohjelmasi.

<u>Vastaukset:</u>

a) Oppilas muokkaa annettua koodinpätkää, siten että se laskee kartion tilavuuden:

```
#pohjan pinta-ala MUOKKAA
A=12.6
#kartion korkeus MUOKKAA
h=4.0
#kartion tilavuus LASKE
V=(1/3)*A*h
print("Pohjan pinta-ala:", A, "kartion korkeus:", h, "kartion tila-
vuus:", V)
```

Muokkaamalla muuttujien A ja h arvoja saadaan kartioiden tilavuudet (kuvan järjestyksessä): 16,8 5,2 3,1 0,8 9,5

b) Oppilas lisää ohjelmaan käyttäjän syötteen ja pohjan pinta-alan laskemisen halkaisijan avulla. Tehtävässä on annettu testidata, jolla ohjelmaa voi testata.

```
import math
#pohjan pinta-ala
d=float(input("Anna kartion pohjan halkaisija:"))
A=math.pi*(d/2)**2
#kartion korkeus
h=float(input("Anna kartion korkeus:"))
#tilavuus
V=A*h/3
print("Kartion tilavuus:", V)
```

c) Oppilas ratkaisee eri ongelman. b-kohdan ohjelmakoodista on varmasti apua:

```
import math
#pohjan pinta-ala
d=float(input("Anna kartion pohjan halkaisija:"))
A=math.pi*(d/2)**2
#kartion korkeus
V=float(input("Anna kartion tilavuus:"))
#tilavuus
h=3*V/A
print("Kartion korkeus:", h)
```

Tilan säästämiseksi vastauksista on jätetty tulostus pois. Oppilaan vastaukseen toivotaan myös ohjelman testitulostuksia sopivalla testidatalla.

Tämän tehtävän ratkaisemisessa **toiminnallinen**/unplugged vaihe on ratkaisun hahmottelu kynän ja paperin avulla: oppilas tunnistaa, että tarvitaan kaava kartion tilavuuden laskemiseen. **Kokeileminen** toteutuu, kun oppilas a) -kohdassa rakentaa ohjelman rungon ja saa ohjelman laskemaan eri kartioiden tilavuuksia. b) -kohdassa oppilas **tekee** toimivan ohjelman ja c)-kohdassa **yhdistää soveltaen** valmiin ohjelman toiseen matemaattiseen ongelmaan. Vielä paremmin nämä eri vaiheet ilmenisivät tehtävässä, jossa tutkitaan uutta asiaa. Samoin voidaan lisätä, että konstruktionismille keskeinen sosiaalinen ulottuvuus jää vain luokkahuonekeskustelun varaan.

9.2 OHJELMOINNIN OPETTAMISEN MENETELMÄT 2

Ohjelmointia voi olla vaikea oppia. Ohjelmointikielen vieraat käsitteet ja niiden merkityksen ymmärtäminen on haastavaa. Lisäksi on pystyttävä luomaan ohjelmointikielellä loogisia kokonaisuuksia, jotka ratkaisevat matemaattisen ongelman tai saavat tietokoneen toimimaan halutulla tavalla. Oppilaiden motivaatio

voi kadota jatkuvien virheilmoitusten tai ohjelman hiljaisen toimimattomuuden edessä. Internetistä valmiiden ohjelmien kopiointi voi olla nopea ratkaisu, mutta oppilaiden käsitys kopioidusta ohjelmasta voi olla puutteellinen tai sisältää väärinkäsityksiä. PRIMM (Predict, Run, Investigate, Modify, Make; Sentance 2020; Sentance & Waite 2017) -menetelmä keskittyy juuri em. ohjelmoinnin aloittamiseen liittyvien ongelmien ehkäisemiseen.

PRIMM-menetelmää voidaan pitää "Ohjelmoinnillisen ajattelun pedagogisen viitekehyksen" (Kotsopoulos ym 2017; ks. edellinen kappale Ohjelmoinnillisen ajattelun pedagoginen viitekehys) sovelluksena. Se keskittyy viitekehyksen ohjelmoinnin oppimiseen kokeilemalla ja ohjelman tekemiseen liittyviin osiin. Muut kaksi osaa jäävät ulkopuolelle, mutta toiminnalliset harjoitukset voisivat pohjustaa PRIMM-menetelmää ja soveltava yhdistely jatkaa menetelmää edistyneempään ohjelmointiin.

PRIMM antaa rakenteen ohjelmoinnin opetukselle jakamalla oppimisen viiteen vaiheeseen (ks. seuraava taulukko): ennusta, suorita, tutki, muokkaa ja tee. Vaiheet voivat muodostaa yhden oppitunnin tai vaihtoehtoisesti sarjan niitä. Menetelmässä aloitetaan ohjelmointi lukemalla esimerkkiohjelmia ennen kuin kirjoitetaan omia. Siinä opitaan keskustelemaan ohjelmista pienryhmissä ja samalla kun ilmaisee ajatuksensa muille, tulee myös omaa tietoa jäsenneltyä. Ohjelmoinnin oppimisen haastavuutta vähennetään pilkkomalla opittavat asiat osiin ja etenemällä ymmärrys edellä. Oppilaan vastuu ohjelmista rakentuu menetelmässä vaiheittain ohjelmien rakenteisiin ja niiden seurauksiin tutustumalla, kunnes lopuksi kirjoitetaan omia ohjelmia.

<u>PRIMM-menetelmän vaiheet ja niiden kuvaukset:</u>

Menetelmä (PRIMM)	**Kuvaus**
Predict – Arvioi	• keskitytään ohjelman seurauksiin/tuloksiin • työskentelyä kahden tai kolmen hengen ryhmissä • keskustelua; mitä ohjelman suorituksesta seuraa
Run – Suorita	• oppilaille suoritettava ohjelma • verrataan arvioita ja ohjelman suorituksen tuloksia
Investigate – Tutki	• hyödynnetään erilaisia keinoja ohjelman tutkimiseen esim. suorituksen seuraamista (muuttujien arvojen muutokset), ohjelmakoodin annotointia, ohjelmakoodiin liittyviä kysymyksiä, virheiden korjaamista, jne. • työskentelyä kahden tai kolmen hengen ryhmissä
Modify – Muokkaa	• kokeillaan pienten muutosten vaikutusta ohjelmakoodiin • hyödynnetään mitä on opittu ohjelman rakenteesta • lisätään vaikeusastetta askelittain
Make – Tee	• ohjelman luomista

	• opittujen taitojen hyödyntämistä • ohjattuja tai avoimia tehtäviä

PRIMMin vaiheet muodostavat selkeän opetusprosessin sisältäen menetelmät ja tavoitteet. Arviointivaiheessa oppilaat keskustelevat ohjelmakoodista ja sen eri osista. Samalla he pyrkivät arvioimaan, mitä ohjelman eri osien suorittamisen yhdistämisestä seuraisi. He voivat kirjoittaa, mutta jos ohjelma on graafinen, niin myös piirtää tuloksen. Ohjelman suorittamisvaiheessa oppilaat kokeilevat vastaavatko tulokset arvioita. Jos he huomaavat, että tulos poikkesi arvioista, on tärkeää kannustaa heitä arvaamaan, missä kohtaa ohjelmaa ero syntyi. Tutkimusvaiheessa ohjelmaa tarkastellaan tarkemmin. Opettajalla on tässä vaiheessa keskeinen rooli erilaisten koodin luonnetta esille tuovien tehtävien keksimisessä. Muokkausvaiheessa muokataan ohjelmakoodia askelittain helposta haastavaan. Oppilaiden ohjelman omistajuus kehittyy sitä mukaan, kun heidän kykynsä muokata ohjelmaa kasvaa. Tekemisvaiheessa oppilaat luovat uuden ohjelman hyödyntäen oppimaansa. He voivat hyödyntää osasia aikaisemmista harjoituksista ja yhdistellä niitä uudella tavalla.

Tutkimuslähtöisenä menetelmänä PRIMMin rakenne perustuu aikaisempiin ohjelmoinnin opetuksesta kehitettyihin teorioihin ja strategioihin (Sentance ym. 2019a; Sentance ym. 2019b). Sen lähtökohtana on, että ohjelmoinnin oppimisessa tarvitaan strukturoitua opetusta ja ohjattua oivaltamista. Kun ohjelmoinnin periaatteet ovat tuttuja voidaan siirtyä vapaamuotoisempaan oppimiseen. Opimme lukemaan ennen kuin opimme kirjoittamaan, mutta perinteisesti ohjelmointi aloitetaan heti ensimmäisistä käsitteistä. Ohjelman lukemisen taitoja tarvitaan myös omien ohjelmien kirjoittamisessa, koska näin voidaan tarkastella jo kirjoitettua ja etsiä täydennettäviä kohtia sekä vaihtoehtoisia etenemistapoja. Valmiilla malliohjelmilla voidaan välittää hyviä ohjelmointikäytänteitä ja tuoda esille selkeän rakenteen merkityksen ohjelmakoodin luettavuudelle. PRIMM on yksi tulkinta käytä-kokeile-luo -mallia hyödyntävästä opetuksesta, johon on yhdistetty em. ohjelmoinnin opetukseen ja oppimiseen liittyviä tutkimustuloksia.

Lähteet:

Sentance, S., & Waite, J. (2017). PRIMM: Exploring pedagogical approaches for teaching text-based programming in school. In Proceedings of the 12th Workshop on Primary and Secondary Computing Education (pp. 113-114).

Sentance, S., Waite, J., & Kallia, M. (2019a). Teachers' Experiences of using PRIMM to Teach Programming in School. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (pp. 476-482).

Sentance, S., Waite, J., & Kallia, M. (2019b). Teaching computer programming with PRIMM: a sociocultural perspective. Computer Science Education, 29(2-3), (pp. 136-176).

Sentance, S. (2020). The I in PRIMM. Hello World, 14, pp. (50–53).

Löydät kolme ensimmäistä artikkelia PRIMM-menetelmää esittelevältä sivustolta (primmportal.com/research/) ja neljännen artikkelin Hello World lehdestä (helloworld.cc), joka on ilmainen opettajille tarkoitettu online-lehti ohjelmoinnin ja digitaalisen teknologian hyödyntämisestä opetuksessa.

9.2.1 PRIMM OPPITUNNIN MALLI

Tämä oppitunnin esimerkin suunnitelma ja sisältö perustuu vuonna 2018 PRIMM-menetelmästä tehtyyn tutkimukseen (https://primmportal.com/2018/08/23/primm-materials-2018/). Oppimateriaali on tarkoitettu 8–9 luokkalaisille (Brittiläinen koulujärjestelmä) ja se sisälsi kymmenen oppituntia.

Ensimmäisessä oppitunnissa kerrataan 7. luokan ohjelmoinnin sisältöjä. Ohjelmointikielenä Python. Kaikki materiaali löytyy https://primmportal.com/2018/08/23/primm-materials-2018/ kohdasta Lesson 1 - Recap.

9.2.1.1 RAKENNE

- Harjoitus 1 (10 min): Oppilaiden tulisi lukea saatu ohjelmakoodi, arvioida ryhmissä mitä se tekee ja kirjoittaa arvio paperille.
- Harjoitus 2 (5 min): Kokeillaan ohjelman suorittamista ja verrataan tulosta arvioon.
- Harjoitus 3 (10 min): Oppilaat vastaavat ohjelmakoodia käsitteleviin kysymyksiin.
- Harjoitus 4 (25 min): Oppilaat muokkaavat saamaansa koodia.
- Harjoitus 5 (10 min): Tehdään omia ohjelmia ja harjoitellaan virheiden etsimistä ohjelmakoodista.

9.2.1.2 TAVOITTEET

- Yksinkertaisten ohjelmien kirjoittaminen
- Käyttäjän syötteen lukeminen
- Tulostus-lauseen kirjoittaminen, joka sisältää yhden muuttujan

Oppitunnin lopuksi pitäisi ymmärtää mitä print() ja input() komennot tekevät sekä miten käyttää niitä Python ohjelmissa.

9.2.1.3 SISÄLTÖ

Harjoitus 1 - arvioi

Oppilaiden tulisi lukea saatu ohjelmakoodi (alla), arvioida ryhmissä mitä se tekee ja kirjoittaa arvio paperille.

```
# starter program lesson 1

name = "Billy"
print("We want to know if you like progamming!")
print()
```

```
print("Do you like programming " + name + "?")
answer = input()
print("Great! You said " + answer + "!")
print("Let's learn some Python today")
```

Kommentti:

- oppilaiden tulisi ymmärtää muuttuja ja tulostus lause
- miten "`name`" muuttujan arvo saadaan yhdistettyä tulostukseen
- miten asetetaan käyttäjän syöte muuttujan arvoksi

Harjoitus 2 - suorita

Oppilaat suorittavat yllä esitetyn koodin esim. Pythonin IDLE:llä.

Kommentti:

- Kyseessä on kertaus, mutta kuitenkin voi syntyä keskustelua, jos ei muistettu täsmällisesti käskyjen ja operaattoreiden merkitystä.

Harjoitus 3 - tutki

Opettajalla on valmiina kysymykset, jotka syventävät oppilaiden ymmärrystä ohjelmasta

- How many variables can you find in this program and what are they being used for?
- print() is a function that you can use to display to the screen. What other function is in this program and what does it do?
- What is the "+" being used for in this program?
- What happens when you use print() with nothing inside the brackets?

Lopuksi pyydetään oppilaita hyödyntämään oppimaansa ja annotoimaan ohjelmakoodi lisäämällä siihen kommentit.

Now add comments (beginning with #) to the program to make sure everything you understand is included in your program for future reference. When you think you understand how this program works then go on to the exercises.

Harjoitus 4 - muokkaa

Tässä oppilaat muokkaavat jo nyt tutuksi tullutta ohjelmaa.

1. Change the program so that it uses your name instead of Billy.
2. Change the program so that it lets you type in your answer on the same line as the question.
3. Change the program so that it asks the user (the person using your program) what their name is at the beginning of the program

Heille annetaan ohje tallentaa muokattu ohjelma uudella nimellä (Save as...).

Harjoitus 5 - tee

Hyödyntäen edellä esitetyn ohjelmakoodin ja siihen tehtyjen muokkausten tietoja (edellä tehty tallenne), oppilailla on seuraavana tehtävänä luoda uusi ohjelma.

- Ask the user three questions, and each time give a suitable response:
- What their name is
- What they had for breakfast
- What their favourite colour is
- Then ask another person to test your program.

Tähän osuuteen kuuluu myös ohjelman virheiden löytäminen ja korjausten ehdottaminen.

Löydä virheet:

```
##name = "Billy"
##print("We want to know if you like progammming!)
##print()
##print("Do you like programming " + Name + "?")
##answer = input(
##Print("Great! You said " + anwer + "!")
##print("Let's learn some Python today")
```

9.3 PYTHONIN KIRJASTOJA OHJELMOINNIN YHDISTÄMISEKSI MATEMATIIKKAAN

Random-, math- ja turtle matemaattisten kirjastojen lisäksi Pythonista löytyy valmiina myös mm. statistics- ja fractions-kirjastot, joissa on valmiina tilastoihin ja murtolukuihin liittyviä ominaisuuksia. Seuraavaksi tutustutaankin näihin kirjastoihin.

Pythoniin on olemassa muitakin (matematiikka)kirjastoja, kuten NymPy (https://numpy.org/). Niitä ei kuitenkaan käsitellä tällä kurssilla, eivätkä ne välttämättä sovellu peruskoulun opetuskontekstiin, koska ne sisältävät paljon hyvin edistyneitä ominaisuuksia ja ne vaativat asentamisen tietokoneelle (mikä ei monilla koulun laitteilla ole mahdollista). Niihin voi kuitenkin törmätä, jos hakee tietoa Pythonin matemaattisista ominaisuuksista.

9.3.1 OHJELMOINTIA JA TILASTOJA

Statistics-kirjasto (https://docs.python.org/3/library/statistics.html) sisältää tilastomatematiikkaan soveltuvia ominaisuuksia. Monet niistä ylittävät perusopetuksen tason, mutta esimerkiksi listojen keskiarvoja ja mediaaneja voi laskea helposti tämän kirjaston avulla.

statistics-kirjaston ominaisuuksia:

Ominaisuus	Merkitys
`statistics.mean(lista)`	Palauttaa listan aritmeettisen keskiarvon
`statistics.geometric_mean(lista)`	Palauttaa listan geometrisen keskiarvon
`statistics.median(lista)`	Palauttaa listan mediaanin (jos parillinen lukumäärä, palauttaa kahden keskimmäisen arvon keskiarvon)
`statistics.median_low(lista)`	Palauttaa listan mediaanin (jos pariton, palauttaa keskimmäisen arvon, jos parillinen, palauttaa kahdesta keskimmäisestä pienemmän)
`statistics.median_high(lista)`	Palauttaa listan mediaanin (jos pariton, palauttaa keskimmäisen arvon, jos parillinen, palauttaa kahdesta keskimmäisestä suuremman)

9.3.2 OHJELMOINTIA JA MURTOLUKUJA

Pythonin fractions-kirjasto (https://docs.python.org/3/library/fractions.html) antaa mahdollisuuden käsitellä murtolukuja. Murtolukuja muodostetaan Fraction-ominaisuudella. Ominaisuutta käytetään funktion tapaan, ja murtoluvun voikin muodostaa erilaisista arvoista, jotka annetaan parametrina. Huomaa, että Fraction on kirjoitettava isolla alkukirjaimella!

fractions-kirjaston ominaisuuksia:

Ominaisuus	Merkitys
`fractions.Fraction(osoittaja, nimittäjä)`	Murtoluku muodostetaan annetusta kokonaislukuosoittajasta ja -nimittäjästä (murtoluku sievennetään)
`fractions.Fraction(toinen_murtoluku)`	Murtoluku kopioidaan toisesta murtoluvusta (esim. toisesta murtolukumuuttujasta)
`fractions.Fraction(desimaaliluku)`	Murtoluku muodostetaan annetusta desimaaliluvusta (float)
`fractions.Fraction(merkkijono)`	Murtoluku muodostetaan annetusta (murtoluvun, kokonaisluvun tai desimaaliluvun sisältävästä) merkkijonosta

Esimerkkejä:

```
# Avataan kirjasto.
from fractions import Fraction

# Luodaan murtoluku antamalla osoittaja ja nimittäjä.
murtoluku1 = Fraction(84, 24)
print("Murtoluku:", murtoluku1)

# Luodaan murtoluku antamalla toinen murtoluku.
murtoluku2 = Fraction(murtoluku1 * murtoluku1)
print("Murtoluku toisesta murtoluvusta:", murtoluku2)

# Luodaan murtoluku desimaaliluvusta.
murtoluku3 = Fraction(-2.5)
print("Murtoluku desimaaliluvusta:", murtoluku3)

# Luodaan murtoluvut merkkijonosta.
murtoluku4 = Fraction("1/7")
murtoluku5 = Fraction("19.25")
print("Murtoluvut merkkijonoista:", murtoluku4, "ja", murtoluku5)
```

Tulostus:

```
Murtoluku: 7/2
Murtoluku toisesta murtoluvusta: 49/4
Murtoluku desimaaliluvusta: -5/2
Murtoluvut merkkijonoista: 1/7 ja 77/4
```

***Lisätietoa fractions-kirjastosta**: Fractions ei ole funktio (funktioiden nimet kirjoitettaisiin pienellä), vaan luokka (luokkien nimet kirjoitetaan aina isolla alkukirjaimella). Luokat ovat osa olio-ohjelmointia, joka on ohjelmoinnin perusteita edistyneempää. Luokat ovat ikään kuin käsikirjoituksia, joiden perusteella voidaan muodostaa olioita (instansseja), joita käsitellään ohjelmassa. Myös turtle-kirjastosta muodostetut kilpikonnat ovat oliota. Fractions-kirjastoa (ja turtle-kirjastoa) voi kuitenkin hyvin käyttää ilman, että ymmärtää tarkemmin luokkien ja olioiden luonnetta.*

Koska Fraction on aika pitkä sana kirjoittaa etenkin silloin, kun murtolukuja luodaan useita, voidaan ominaisuutta avattaessa määritellä sille lyhenne `as`-avainsanalla.

Esimerkkejä:

```
# Avataan kirjasto.
```

```
from fractions import Fraction as F

# Luodaan murtoluku antamalla osoittaja ja nimittäjä.
murtoluku1 = F(84, 24)
print("Murtoluku:", murtoluku1)

# Luodaan murtoluku antamalla toinen murtoluku.
murtoluku2 = F(murtoluku1 * murtoluku1)
print("Murtoluku toisesta murtoluvusta:", murtoluku2)

# Luodaan murtoluku desimaaliluvusta.
murtoluku3 = F(-2.5)
print("Murtoluku desimaaliluvusta:", murtoluku3)

# Luodaan murtoluvut merkkijonosta.
murtoluku4 = F("1/7")
murtoluku5 = F("19.25")
print("Murtoluvut merkkijonoista:", murtoluku4, "ja", murtoluku5)
```

Kun murtoluku on muodostettu, voidaan hyödyntää kirjaston muita ominaisuuksia.

fractions-kirjaston muita ominaisuuksia:

Ominaisuus	Merkitys
`Fraction.numerator`	Palauttaa murtoluvun osoittajan
`Fraction.denominator`	Palauttaa murtoluvun nimittäjän
`Fraction.limit_denominator(max_nimittäjä)`	Palauttaa murtolukua lähimmän murtoluvun, jonka nimittäjä on korkeintaan parametrina annettu maksimiarvo

Esimerkkejä:

```
# Avataan kirjasto.
from fractions import Fraction as F

# Luodaan murtoluku antamalla osoittaja ja nimittäjä.
murtoluku1 = F(84, 24)
print("Murtoluku:", murtoluku1)
print("Osoittaja:", murtoluku1.numerator)
print("Nimittäjä:", murtoluku1.denominator)

# Luodaan murtoluku desimaaliluvusta.
murtoluku2 = F(-4.43915678873458)
# Tulostetaan murtoluku sellaisenaan...
print(murtoluku2)
```

```
# ...ja niin, että rajoitetaan nimittäjäksi korkeintaan 100.
print(murtoluku2.limit_denominator(100))
```

Tulostus:

```
Murtoluku: 7/2
Osoittaja: 7
Nimittäjä: 2
-2499023107448033/562949953421312
-182/41
```

9.4 ISOMMAT OHJELMAT

Tähän mennessä kirjoittamamme ohjelmat ovat olleet hyvin lyhyitä. Todellisuudessa ohjelmat ovat paljon laajempia kokonaisuuksia sisältäen useita tiedostoja ja jopa kansiollisia eri tiedostoja. Pidemmissä ohjelmissa nousee erityisen tärkeäksi jakaa ohjelmaa pienemmiksi palasiksi (aliohjelmiksi), jotta koodin toistamiselta voidaan välttyä ja kokonaisuus saadaan hallittavammaksi.

Seuraavassa esitetään esimerkki isommasta ohjelmasta, joka toimii pinta-ala- ja tilavuuslaskimena. Ohjelma laskee erilaisten kolmiulotteisten kappaleiden pinta-aloja ja tilavuuksia. Ohjelma koostuu monesta aliohjelmasta, joista jokainen laskee tietyn kappaleen pinta-alan tai tilavuuden, ja pääohjelmasta, joka arpoo erilaisia kappaleita ja tulostaa niiden pinta-alan tai tilavuuden ruudulle.

Pääohjelman lisäksi ohjelma koostuu funktiosta (aliohjelmat), jotka laskevat suorakulmaisen särmiön, suoran ympyrälieriön, suoran ympyräkartion ja pallon pinta-alan ja tilavuuden:

1. `tilavuus_sarmio(a, b, c)`: a, b ja c ovat särmiön särmiä ja tilavuus $V = a \cdot b \cdot c$
2. `ala_sarmio(a, b, c)`: a, b ja c ovat särmiön särmiä ja pinta-ala $A = 2 \cdot a \cdot b + 2 \cdot a \cdot c + 2 \cdot b \cdot c$
3. `tilavuus_lierio(r, h)`: r on pohjaympyröiden säde, h lieriön korkeus ja tilavuus $V = \pi \cdot r^2 \cdot h$
4. `ala_lierio(r, h)`: r on pohjaympyröiden säde, h lieriön korkeus ja pinta-ala $A = 2 \cdot \pi \cdot r^2 + 2 \cdot \pi \cdot r \cdot h$
5. `tilavuus_kartio(r, h)`: r on pohjaympyrän säde, h kartion korkeus ja tilavuus $V = \frac{\pi \cdot r^2 \cdot h}{3}$
6. `ala_kartio(r, h, s)`: r on pohjaympyrän säde, h kartion korkeus, s reunan korkeus ja pinta-ala $A = \pi \cdot r^2 + \pi \cdot r \cdot s$
7. `tilavuus_pallo(r)`: r on pallon säde ja tilavuus $V = \frac{4 \cdot \pi \cdot r^3}{3}$
8. `ala_pallo(r)`: r on pallon säde ja pinta-ala $A = 4 \cdot \pi \cdot r^2$

```
import random
from math import pi
from math import sqrt

random.seed(rnd_seed)
```

```
# ALIOHJELMAT: kolmiulotteisten kappaleiden
# pinta-alan ja tilavuuden laskevat funktiot
#
# Suorakulmainen särmiö
# Tilavuus: särmiöiden pituudet a, b ja c
def tilavuus_sarmio(a, b, c):
    return a*b*c

# Pinta-ala: särmiöiden pituudet a, b ja c
def ala_sarmio(a,b,c):
    return 2*a*b+2*a*c+2*b*c

# Suora ympyrälieriö
# Tilavuus: ympyrän säde r, lieriön korkeus h
def tilavuus_lierio(r, h):
    return pi*r**2*h

# Pinta-ala: ympyrän säde r, lieriön korkeus h
def ala_lierio(r, h):
    return 2*pi*r**2+2*pi*r*h

# Suora ympyräkartio
# Tilavuus: ympyrän säde r, kartion korkeus h
def tilavuus_kartio(r, h):
    return (pi*r**2*h)/3

# Pinta-ala: ympyrän säde r, kartion korkeus h, reunan korkeus s
def ala_kartio(r, h, s):
    return pi*r**2+pi*r*s

# Pallo
# Tilavuus: ympyrän säde r
def tilavuus_pallo(r):
    return (4*pi*r**3)/3

# Pinta-ala: ympyrän säde r
def ala_pallo(r):
    return 4*pi*r**2

# PÄÄOHJELMA: luodaan satunnaisia kolmiulotteisia kappaleita
# kuvaavia lukujen joukkoja ja lasketaan niiden perusteella
# kappaleiden pinta-alat ja tilavuudet kutsumalla aliohjelmia.
#
# Toistetaan laskuja 10 kappaletta
for i in range(10):
    # Arvotaan uudet arvot särmille, säteelle
    # ja korkeudelle (pyöristetään yhteen desimaaliin).
    a = round(random.random()*20+1, 1)
```

JATKUU ->

```
b = round(random.random()*20+1, 1)
c = round(random.random()*20+1, 1)
r = round(random.random()*10+1, 1)
h = round(random.random()*20+1, 1)
# Arvotaan, mikä kappale kyseessä.
kappale = random.randint(1,4)

# Eri kappaleilla eri käsittely.
if kappale == 1: # Suorakulmainen särmiö
    print("Suorakulmainen särmiö")
    print("Särmien pituudet ovat " + str(a) + " cm, " + str(b)
      + " cm ja " + str(c) + " cm.")
    print("Tilavuus on (noin) "
      + str(round(tilavuus_sarmio(a, b, c),1)) + " cm^3.")
    print("Kokonaispinta-ala on (noin) "
      + str(round(ala_sarmio(a,b,c),1)) + " cm^2.")

elif kappale == 2: # Suora ympyrälieriö
    print("Suora ympyrälieriö")
    print("Pohjan säde on " + str(r) + " cm ja lieriön korkeus on"
      + str(h) + " cm.")
    print("Tilavuus on (noin) "
      + str(round(tilavuus_lierio(r,h),1)) + " cm^3.")
    print("Kokonaispinta-ala on (noin) "
      + str(round(ala_lierio(r,h),1)) + " cm^2.")

elif kappale == 3: # Suora ympyräkartio
    print("Suora ympyräkartio")
    # Lasketaan reunan korkeus s.
    s = sqrt(r**2+h**2)
    print("Pohjan säde on " + str(r) + " cm, kartion korkeus on "
      + str(h) + " cm ja kartion reunan korkeus on " + str(s) +
      " cm.")
    print("Tilavuus on (noin) "
      + str(round(tilavuus_kartio(r,h),1)) + " cm^3.")
    print("Kokonaispinta-ala on (noin) "
      + str(round(ala_kartio(r,h, s),1)) + " cm^2.")

elif kappale == 4: # Pallo
    print("Pallo")
    print("Säde on " + str(r) + " cm.")
    print("Tilavuus on (noin) "
      + str(round(tilavuus_pallo(r),1)) + " cm^3.")
    print("Kokonaispinta-ala on (noin) "
      + str(round(ala_pallo(r),1)) + " cm^2.")

else: # Ei mikään kappaleista.
    print("Pieleen meni.")
```

10 TYÖPAJA 4: OHJELMOINTI YHTEISTYÖNÄ, PRIMM-MENETELMÄ JA OHJELMOINNIN OPETUS

Työpajassa käsiteltiin ohjelmointia yhteistyönä, ohjelmoinnin opetusta PRIMM-menetelmällä ja ohjelmoinnin opetuksen suunnittelua. Nykyajan ohjelmistot ovat suuria, joten niiden kehittämiseen tarvitaan usean ihmisen yhteistyötä. Ohjelmointi voi olla yksi tapa harjoitella yhteistyötä koulussa, mutta yhteistyö on myös ohjelmoinnin oppimisen menetelmä. Työpajassa harjoiteltiin saman ohjelman kehittämistä yhdessä niin, että aliohjelmien tekeminen jaettiin osallistujien kesken. Sopimalla yhteiset periaatteet aliohjelmien nimeämiselle ja kutsumisella, mahdollistettiin kehittäminen yhdessä, vaikka jokainen vastasi oman aliohjelman toteutuksesta. PRIMM (Predict, Run, Investigate, Modify ja Make) -menetelmän hyödyntämisessä harjoiteltiin oppimista yhdessä. Tarkoituksena oli, että oppilaat toimisivat sekä pienryhmissä että keskustelisivat kokemuksistaan ja havainnoistaan koko luokan kesken. Opettaja toimisi tässä enemmän ohjaajana ja auttaisi oppilaita kysymyksillä sekä kiinnittämällä huomiota keskeisiin asioihin. Menetelmässä edetään koodin havainnoinnin, kokeilun, tutkimisen ja muokkaamisen kautta omien ohjelmien luomiseen. Lopuksi työpajassa keskityttiin ohjelmoinnin opetuksen suunnitteluun ja ohjelmointitehtävien luomiseen. Viimeinen osuus toimi myös johdantona loppuprojektiin, joka sisälsi oman oppitunnin suunnittelun.

Työpaja alkoi lyhyellä alustuksella, jossa esitettiin yhteenveto edeltävästä verkkosisällöstä ja käytiin lävitse edellisen työpajan kotitehtävien vastaukset. Alustusta seurasi kolme erillistä osiota, joissa käsiteltiin ohjelmointia yhteistyönä, ohjelmoinnin opetusta PRIMM-menetelmällä ja ohjelmoinnin opetuksen suunnittelua.

<u>Ohjelmointia yhteistyönä</u>

- tehdään yhdessä samaa koodia
- yhdistetään eri henkilöiden tekemiä ohjelmiston osia
- tehostaa ohjelmoinnin oppimista
- esimerkki: codingrooms.com
- esimerkki: laskuja kysyvä peli
- vinkkejä opettajalle

Laskuja kysyvä peli aliohjelman pohja:

```
import math
import random

#nimeä aliohjelma
def OMA_NIMI():
    #kirjoita aliohjelman koodi
    print("OMA_NIMI aliohjelma")
    return True

print("testataan aliohjelmaa")
tulos = OMA_NIMI()     #vaihda aliohjelman nimi tähän
print("aliohjelma palautti", tulos)
```

Ohjelmoinnin opetus PRIMM-menetelmällä

- tavoite
  - vähentää oppimisen alkuun pääsemisen esteitä
  - edetään ymmärtäminen edellä
  - opitaan keskustelemaan ohjelmakoodista
- esimerkki: for ja while -silmukat
  - Etsi virheet (5 min)
  - Kertaus: for ja while silmukka (5 min)
  - Arvioi, suorita, tutki (10 min)
  - Ohjelmointiharjoituksia (30 min)
  - Opitun läpikäynti (10 min)
- yhteenveto

PRIMM

Tehtävä (pareittain)

```
def message():
    message=input("Enter your message: ")
    for i in range(10):
        print(message)

def timesTable():
    num=int(input("Enter a number: "))
    for i in range(1,11):
        print(i*num)

def main():
    choice = ""
    while choice !="3":
        print("1. Message Repeater")
        print("2. Time Table")
        print("3. Exit")
        print("Enter your choice:  ")
        choice=input()
        if choice=="1":
            message()
        elif choice=="2":
            timesTable()
    print("Goodbye")

main()
```

Mitä vieressä oleva tehtävä tekee?

Lataa tehtävä zoomin chatista ja kokeile.

Ohjelmoinnin opetuksen suunnittelu

- millaisia erilaisia haasteita tai huomioon otettavia asioita opetuksen eri vaiheissa on?
  - opetuksen suunnittelu
  - ohjeistuksen antaminen oppilaille
  - tehtävän tekeminen
  - tehtävän jälkeen
- keskustelua suunnittelusta
- kehitellään ohjelmointitehtävä ryhmässä
  - mitä ohjelmointitaitoa tehtävä kehittää?
  - miten tehtävä on tarkoitus tehdä?
  - miten tehtävä esitellään oppilaille?
- keskustelua tehtävien luomisesta
- loppuprojektin esittely

Loppuprojekti 1/2

- Suunnittele yläkouluun ohjelmoinnin oppitunti.
- Aiheita: ohjelmoinnin käsitteet, ohjelmoinnin tuki matematiikalle tai matematiikan tuki ohjelmoinnille.
- Myös algoritmisesta ajattelusta sekä matemaattisesta ja ohjelmoinnillisesta ongelmanratkaisusta voi löytyä ideoita aiheeksi.
- Voit hyödyntää Ohjelmoinnillisen ajattelun pedagogista viitekehystä, PRIMM-menetelmää tai keksiä oman tavan oppitunnin jäsentämiseen.

Loppuprojekti 2/2: oppitunnin suunnitelma

- Mille luokalle
- Aihe ja tavoite
- Oppitunnin rakenne/aikataulu
- Sisältö
- Tehtävät
- Mihin asioihin kannattaa kiinnittää huomiota
- Mahdollisuuksia erilaisiin toteutuksiin
- Palauta pdf:nä

11 YHTEENVETO

Edellä oleva sisältö on kooste syksyllä 2021 yläkoulun matematiikan opettajille järjestetyn kahden opintopisteen täydennyskoulutuskurssin, ohjelmoinnin opettamisesta matematiikan osana, tekstimateriaaleista. Kurssi oli jatkoa keväällä pidetylle alakoulun luokanopettajien vastaavalle kurssille. Kohderyhmänä olivat aineenopettajat, jotka kaipasivat opetuksen kehittämiseksi taustatietoa ohjelmoinnista. Lisäksi vastavalmistuneelle opettajalle kurssi tarjosi lähtökohdan ohjelmoinnin opetuksen aloittamiselle. Kurssi tarjosi viitekehyksen, jolla opettaja voi jäsentää oppikirjojen ohjelmointisisältöjä, täydentää niitä ulkopuolisilla materiaaleilla, luoda yhteyksiä ohjelmoinnin ja matematiikan välille sekä hyödyntää ohjelmoinnin opetusmenetelmiä. Sisältö noudatti yläkoulun matematiikan opetussuunnitelmaa ja antoi selityksen sen aika lyhyille ohjelmoinnin sisältöjen määritelmille.

Yläkoulussa ohjelmoinnin ja matematiikan yhteys on tiiviimpi. Tavoitteena on algoritmisen ajattelun sekä matemaattisen ja ohjelmoinnillisen ongelmanratkaisun oppiminen. Lisäksi yläkoulussa siirrytään tekstipohjaisiin ohjelmointikieliin. Kurssilla tarkasteltiin algoritmista ajattelua sekä matematiikan että ohjelmoinnin näkökulmasta. Myös ongelmanratkaisun yhtäläisyyksiä ja eroavaisuuksia tuotiin esille. Ohjelmointikielenä käytettiin Pythonia, joka on suosittu koulukäytössä. PRIMM-menetelmä esitettiin yhtenä ratkaisuna ohjelmoinnin oppimisen haasteisiin. Tekstisisältöjen lisäksi kurssi koostui harjoituksista, työpajoista ja ohjelmoinnin oppitunnin suunnitteluprojektista. Työpajoissa käytännön tekemisen lisäksi keskeistä oli keskustelu, jolloin opettajat saivat kuulla kollegoittensa näkemyksiä ja kokemuksia ohjelmoinnista. Oppituntien suunnittelu päätti kurssin ja se mahdollisti kurssilla opitun soveltamisen käytäntöön.