Python-oppimateriaali (CHEM-A2600)

Lyhyt opas Python-ohjelmointiin

Site: MyCourses
Course: CHEM-A2600 - Kemiantekniikan ohjelmointikurssi, 29.10.2018-07.12.2018
Book: Python-oppimateriaali (CHEM-A2600)
Printed by: Antti Karttunen
Date: Friday, 21 December 2018, 12:07 AM

Table of contents

4.4. Laskuoperaatiot NumPy-taulukoilla
4.5. NumPyn matemaattiset funktiot
4.6. Matplotlib-kirjasto
4.7. matplotlib.pyplot-moduuli
4.8. Matplotlib-määritelmiä
4.9. NumPy-polynomit

5. Kierros 5
5.1. Tiedostojen avaaminen ja käsittely
5.2. Datan lukeminen ja kirjoittaminen
5.3. JSON-tiedostot
5.4. Tiedostojen helppo käsittely NumPy:llä
5.5. Virhetilanteiden käsittely (try-except-finally)

6. Kierros 6
6.1. SciPy
6.2. scipy.constants
6.3. scipy.stats
6.4. scipy.linalg
6.5. scipy.integrate.odeint
6.6. Olio-ohjelmointia 1
6.7. Olio-ohjelmointia 2
6.8. Olio-ohjelmointia 3

7. Lisämateriaalia
7.1. Anacondan asennusohje
7.2. Virheiden etsiminen ja korjaaminen

Oppimateriaalin lisenssi

 Creative Commons Attribution-ShareAlike 4.0 International License.

Oppimateriaalin tekijät: Antti Karttunen (2016-2018), Tarmo Nieminen (2018)

Kierros 1

Kurssin ensimmäisellä kierroksella tutustutaan ohjelmoinnin peruskäsitteisiin ja Python-ohjelmointikielen perusteisiin.

Oppaan lukuohje

Kun oppaassa esitetään Python-koodia, se näyttää tältä:

```
print("Nyt lasketaan!")
print("11*11 on", 11*11)
```

Kun oppaassa näytetään, mitä Python-koodi tulostaa, se näyttää tältä:

```
Nyt lasketaan!
11 * 11 on 121
```

Oppaan ohjelmien kokeileminen itse

Voit myös itse kokeilla ajaa minkä tahansa oppaassa listatuista Python-ohjelmista. Kopioi vain koodi Spyder-editoriin ja aja se (paina Spyderissä vihreää "Run"-painiketta tai F5-nappia). Koodien kokeileminen itse on erittäin suositeltavaa, koska se voi helpottaa merkittävästi esimerkkien ymmärtämistä.

Isot ja pienet kirjaimet

Pythonissa isot ja pienet kirjaimet ovat merkitseviä. Käsky **print** on siis eri asia kuin **Print** tai **PRINT**.

Ohjelmakoodin kommentointi

Ohjelmien huolellinen kommentointi on ensiarvoisen tärkeää, jotta:

- Muut ymmärtävät, mitä kirjoittamasi koodi tekee
- Muistat itse, mitä kirjoittamasi koodi tekee!

Ohjelmakoodiin voi lisätä kommentteja #-merkin jälkeen:

```
# Aloitetaan!
print("Eka ohjelmani")
# Jatketaan!
print("Moi!") # Rivin loppuun voi myös lisätä kommentteja
```

Ylläoleva ohjelma tulostaisi:

```
Eka ohjelmani
Moi!
```

Huomaa, että kommentit eivät tulostuneet.

Monirivisiä kommentteja voi kirjoittaa """ *kommentti* """ –merkinnällä:

```
print("Eka ohjelmani")
"""
Olipa hieno kokemus!
Tämä on kolmerivinen välikommentti.
Sitten jatketaan!
"""
print("Moi!")
```

Oppaan sisältämät tehtävät

Opas sisältää myös erilaisia tehtäviä, joilla on kaksi eri tarkoitusta:

- Oppaassa esitettyjen asioiden havainnollistaminen
- Voit tarkistaa, kuinka olet sisäistänyt oppaassa esitetyt asiat

Oppaassa olevien tehtävien tarkoitus on tukea oppimista, **ne eivät vaikuta kurssin arvosteluun**!

Alla on kaksi esimerkkiä erilaisista tehtävätyypeistä.

Tehtävä 1.0.1

Mitä ohjelmointikieltä tällä kurssilla opetellaan?

- Cobra
- INTERCAL
- Perl
- Python

Tehtävä 1.0.2

Täydennä koodi niin, että ohjelma tulostaa
Yksi
Kaksi
Kolme

```
print("Yksi")
[ ]("Kaksi")
print("Kolme")
```

Check

Tulostaminen (print) ja syötteen lukeminen (input)

Tulostaminen *print*-funktiolla

Pythonissa voi tulostaa tietoa ruudulle *print*-funktiolla (opimme lisää funktioista myöhemmin).

```
# Tulostetaan merkkijono "Terve!"
print("Terve!")
# Tulostetaan kolme lukuarvoa ja kolme tyhjää riviä ("\n")
print("Tulostetaan lukuja:", 2, 1001, -40.55, "\n\n\n")
# Voimme tulostaa myös laskutoimitusten tuloksia
print("11*11 on", 11*11)
```

Ylläolevat kolme lauseketta tulostavat näin (huomaa kolme tyhjää riviä lukuarvojen jälkeen):

```
Terve!
Tulostetaan lukuja: 2, 1001, -40.55




11*11 on 121
```

Rivinvaihdot: print-funktio lisää tekstin loppuun oletuksena rivinvaihdon "\n". Tähän voi vaikuttaa print-funktion *end*-parametrillä:

```
print("Rivi 1.")
print("Rivi 2. Rivien väliin tuli rivinvaihto.")
print("Teksti 1.", end = " ")
print("Teksti 2. Tekstien väliin tuli välilyönti.")
```

tulostaa

```
Rivi 1.
Rivi 2. Rivien väliin tuli rivinvaihto.
Teksti 1. Teksti 2. Tekstien väliin tuli välilyönti.
```

Lopuksi: Halutessasi voit tehdä laskutoimituksia myös suoraan Python-konsolissa. Kokeile kirjoittaa konsoliin esim. 5*5 ja paina Enter

Käyttäjän syötteen lukeminen *input*-funktiolla

Käyttäjältä voi kysyä tietoja *input*-funktiolla:

```
# Kysytään käyttäjän nimeä
nimi = input("Mikä nimesi on?")
print("Hieno nimi sinulla", nimi)
```

Lopputulos:

```
Mikä nimesi on?Antti
Hieno nimi sinulla Antti
```

Kysymys ja vastaus tulostuvat selkeämmin, jos lisätään välilyönti merkkijonon loppuun:

```
nimi = input("Mikä nimesi on? ")
print("Hieno nimi sinulla", nimi)
```

Lopputulos:

```
Mikä nimesi on? Antti
Hieno nimi sinulla Antti
```

Kaikkein selkeintä on yleensä käyttää rivinvaihtoa "\n" kysymyksen lopussa

```
nimi = input("Mikä nimesi on?\n")
print("Hieno nimi sinulla", nimi)
```

Lopputulos (**Huom!** Tästä lähtien käyttäjän input-funktiolle antama syöte merkitään ">"-merkillä):

```
Mikä nimesi on?
> Antti
Hieno nimi sinulla Antti
```

Huom! *input*-funktio lukee aina ns. *merkkijonon* (engl. string). Tämä koodi:

```
luku = input("Anna luku niin kerron sen kahdella:\n")
print("Antamasi luku", luku, "kerrottuna kahdella on", 2 * luku)
```

ei siis annakaan odotettua lopputulosta:

```
Anna luku niin kerron sen kahdella:
> 5
Antamasi luku 5 kerrottuna kahdella on 55
```

Tämä ongelma ratkeaa seuraavassa luvussa, jossa opimme käsitteet *muuttuja* ja muuttujan *tyyppi*.

Tehtävä 1.1.1

Täydennä allaoleva koodi niin, että se tulostaa:

Hiilimonoksidin (CO) moolimassa on 28.01 g/mol
0.5 mol hiilimonoksidia painaa siis 14.005 grammaa

[ ]("Hiilimonoksidin (CO) moolimassa on 28.01 g/mol")
print("0.5 mol hiilimonoksidia painaa siis", [ ] / 2, "grammaa")

Check

Muuttujat

Ohjelmoidessa tallennamme tietoa *muuttujiin* (engl. *variable*). Esimerkiksi *input*-funktio tallentaa tässä esimerkissä käyttäjän syötteen merkkijonona *nimi*-muuttujaan:

```
nimi = input("Anna nimesi\n")
```

Tavallisia muuttujatyyppejä Pythonissa ovat:

- Merkkijonot, **str**, merkitään lainausmerkeillä ("Hei!" tai 'Hei!')
- Kokonaisluvut, **int** (2, -2, 1000000)
- Liukuluvut, **float** (1.0, -3.00003, 1258.941662) – eli "desimaaliluvut"
- Kompleksiluvut, **complex** (2.0 + 3.0j)
- Totuusarvot (engl. boolean), **bool** (True, False)

Muutama esimerkki muuttujien käytöstä:

```
iso_luku = 50000005 * 50000005
print("Iso lukumme on", iso_luku)
pieni_luku = 1/iso_luku
print("Pieni lukumme on", pieni_luku)
```

Lopputulos:

```
Iso lukumme on 2500000500000025
Pieni lukumme on 3.99999920000012e-16
```

Muuttuja **iso_luku** on ylläolevassa kokonaisluku, kun taas muuttuja **pieni_luku** on liukuluku. Toisin kuin monissa muissa ohjelmointikielissä, Pythonissa muuttujan tyyppiä ei tarvitse määritellä ennen muuttujan käyttämistä. Python päättelee muuttujan tyypin, kun muuttujan arvo asetetaan.

Huom! Älä käytä muuttujien nimissä koskaan ääkkösiä (ä, ö, å)! Se johtaa ongelmiin.

Tehtävä 1.2.1

Mikä on muuttujan *vastaus* tyyppi?

vastaus = 42

- Kokonaisluku (int)
- Merkkijono (str)
- Kompleksiluku (complex)
- Liukuluku (float)

Tyyppimuunnokset

Monesti on tarpeen muuntaa muuttujia yhdestä tyypistä toiseen.

Muunetaan merkkijono liukuluvuksi *float*-funktiolla:

```
merkkijono = "2.0"
luku = float(merkkijono)
print("Luku", merkkijono, "jaettuna kahdella on", luku / 2, "\n")
```

Lopputulos:

```
Luku 2.0 jaettuna kahdella on 1.0
```

Liukuluvun tai kokonaisluvun taas voi muuntaa merkkijonoksi *str*-funktiolla:

```
luku1 = 5
luku2 = 5.0
jono1 = str(luku1)
jono2 = str(luku2)
print("Yhdistämällä merkkijonot", jono1, "ja", jono2, "saadaan merkkijono", jono1 + jono2)
print("Yhdistämällä kokonaisluku", luku1, "ja liukuluku", luku2, "saadaan liukuluku", luku1 + luku2)
```

Lopputulos:

```
Yhdistämällä merkkijonot 5 ja 5.0 saadaan merkkijono 55.0
Yhdistämällä kokonaisluku 5 ja liukuluku 5.0 saadaan liukuluku 10.0
```

Muunnetaan *input*-funktiolla luettu merkkijono suoraan kokonaisluvuksi *int*-funktiolla:

```
luku = int(input("Anna luku niin kerron sen kahdella\n"))
print("Antamasi luku", luku, "kerrottuna kahdella on", 2 * luku, "\n")
```

Lopputulos (Muista, että ">"-merkki tarkoittaa käyttäjän *input*-funktiolle antamaa syötettä):

```
Anna luku niin kerron sen kahdella
> 3
Antamasi luku 3 kerrottuna kahdella on 6
```

Muunnetaan *input*-funktiolla luettu merkkijono suoraan liukuluvuksi *float*-funktiolla:

```
luku = float(input("Anna luku niin kerron sen numerolla 2.6\n"))
print("Antamasi luku", luku, "kerrottuna numerolla 2.6 on", 2.6 * luku)
```

Lopputulos:

```
Anna luku niin kerron sen numerolla 2.6
> 5
Antamasi luku 5.0 kerrottuna numerolla 2.6 on 13.0
```

Tärkeää muistaa: luku = float(input("Teksti")) on käytännössä helpoin tapa lukuarvojen lukemiseen *input*-funktiolla.

Tehtävä 1.3.1

Täytä aukkopaikat niin, että tyyppimuunnokset ovat oikein

```
# Merkkijono kokonaisluvuksi
kokonaisluku = [          ]("2017")
# Merkkijono liukuluvuksi
liukuluku = [          ]("3.14")
# Kokonaisluku merkkijonoksi
merkkijono = [          ](82347827)
# Luetaan liukuluku input-funktiolla
luku = [          ](input("Anna liukuluku:\n"))
```

Check

Kokonaisluvut, liukuluvut ja pyöristäminen

Edellisessä luvussa opeteltiin lukemaan lukuarvoja input-funktiolla:

```
luku = float(input("Anna luku niin kerron sen numerolla 2.6\n"))
print("Antamasi luku", luku, "kerrottuna numerolla 2.6 on", 2.6 * luku)
```

Tarkastellaan, mitä tämä koodi tulostaa, kun annamme syötteeksi liukuluvun 3.0:

```
Anna luku niin kerron sen numerolla 2.6
3.0
Antamasi luku 3.0 kerrottuna numerolla 2.6 on 7.800000000000001
```

Oho? Miksi koodi tulostaa 7.800000000000001 eikä 7.8? Tämä johtuu tavasta, jolla tietokoneet käsittelevät liukulukuja (lisätietoa aiheesta kiinnostuneille Python-tutoriaalissa). Luonnollisesti meille riittäisi tässä tapauksessa yhden desimaalin tarkkuus. Liukulukujen kanssa tarvitsemme siis usein pyöristysfunktiota *round*.

round-funktio

Kokonaisluvun (int) muuntaminen liukuluvuksi (float) on yksinkertaista. Muunnetaan kokonaisluku 5 liukuluvuksi ja tulostetaan se:

```
print(float(5))
```

tulostaa

```
5.0
```

Mutta liukulukujen muuntamisessa kokonaisluvuiksi tulee olla tarkkana:

```
print(int(5.1))
print(int(5.9))
```

tulostaa

```
5
5
```

Liukuluvun suora muunnos *int*-funktiolla siis katkaisee liukuluvun desimaalipisteen kohdalta. Liukuluvun voi pyöristää lähimpään kokonaislukuun *round*-funktiolla:

```
print(round(5.1))
print(round(5.9))
```

tulostaa

```
5
6
```

Liukulukuja voi myös pyöristää haluttuun tarkkuuteen. *round*-funktion toinen parametri kertoo käytettävien desimaalien määrän:

```
print(round(5.666, 1))
print(round(5.666, 2))
```

tulostaa

```
5.7
5.67
```

round-funktiota voi siis hyödyntää, kun ilmoitamme liukulukulaskujen tuloksia käyttäjälle. Kierroksen 2 materiaalissa kerrotaan lisäksi str.format-funktiosta, jonka avulla liukulukujen pyöristäminen tulostamista varten on hyvin helppoa.

Huom! Älä koskaan pyöristä liukulukuja varsinaisten laskutoimitusten aikana! Liukuluvuilla työskennellään aina mahdollisimman suurella tarkkuudella ja ainoastaan käyttäjälle ilmoitettava luku pyöristetään johonkin ihmissilmälle sopivampaan tarkkuuteen. Ilmoitustarkkuuteen pätevät tässä samat säännöt kuin normaalistikin, eli tuloksen ilmoitustarkkuus riippuu esim. lähtöarvojen tarkkuudesta.

Kokonaislukujen pyöristäminen round-funktiolla

round-funktiolla on myös vähemmän tunnettu ominaisuus, jonka avulla voi helposti pyöristää lukuja haluttuun ilmoitustarkkuuteen myös desimaalipisteen vasemmalta puolen. Tätä ominaisuutta tarvitaan usein luonnontieteissä, kun mittaustarkkuus rajoittaa vastauksen tarkkuutta. Tällöin funktion toinen parametri annetaan negatiivisena:

```
print(round(5624, -3)) # tarkkuus: 10^3
print(round(5624, -2)) # tarkkuus: 10^2
print(round(5624, -1)) # tarkkuus: 10^1
```

tulostaa

```
6000
5600
5620
```

Tässä esimerkissä pyöristettiin siis kokonaislukuja haluttuun tarkkuuteen. Eli *round*-funktion toinen parametri *ndigits* tarkoittaa sekä positiivisten että negatiivisten lukujen kohdalla "pyöristä tarkkuuteen $10^{-ndigits}$".

Tehtävä 1.4.1

```
# Mitä tämä ohjelma tulostaa?
print(round(3.14159, 2))
```

- 3.14
- 3.1
- pii
- 3.14159

Matemaattiset perusoperaattorit

Erilaisia laskutoimituksia varten Pythonissa on käytettävissä normaalit matemaattiset operaattorit:

Operaattori	Selitys	Kokeile konsolissa
+	Yhteenlasku	5 + 5
-	Vähennyslasku	1000 - 4
*	Kertolasku	11 * 11
/	Jakolasku	11 / 5 (tulos = 2.2, eli **float**)
//	Katkaiseva jakolasku	11 // 5 (tulos = 2, eli **int**)
%	Jakojäännös	11 % 5 (tulos = 1, eli **int**)
**	Potenssiin korotus	2 ** 4
abs(x)	Itseisarvo	abs(4-16)

Lisähuomioita

1) Laskujärjestystä voi säätää suluilla:

```
print(2 ** (2 + 2))
print(2 ** 2 + 2)
```

tulostaa:

```
16
6
```

2) Jakojäännösoperaattorilla on kätevä testata kokonaislukujen jaollisuutta:

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

3) Merkkijonoja voi yhdistää:

```
print("Lappeen" + "ranta")
```

tulostaa

```
Lappeenranta
```

4) Myös merkkijonoja (string) ja kokonaislukuja (int) yhdistävät operaatiot on sallittu:

```
print("tip tap" * 5)
```

tulostaa

```
tip tap tip tap tip tap tip tap tip tap
```

Lyhennetyt laskuoperaatiot

Pythonissa voi käyttää myös lyhennettyjä laskuoperaatioita +=, -=, *= ja /=

```
# Annetaan muuttujalle n alkuarvo
n = 10

# Sama kuin: n = n + 1 (eli n on nyt 11)
n += 1

# Sama kuin: n = n - 1 (eli n on nyt 10)
n -= 1

# Sama kuin: n = n * 2 (eli n on nyt 20)
n *= 2

# Sama kuin: n = n / 2 (eli n on nyt 10.0)
n /= 2
```

On puhdas makuasia, kumpaa muotoa haluaa käyttää, pitkää vai lyhyttä. Pitkä on aloittelijalle selkeämpi valinta.

Tehtävä 1.5.1

Paljonko on 55 // 11?

- 1
- 5
- 5.0
- 0

if-elif-else -ehtolauseet ja vertailuoperaattorit

if-ehtolauseen avulla ohjataan ohjelman suoritusta haluttuun suuntaan. Siitä on kaksi eri muotoa: if-else ja if-elif-else.

```
if ehto:
    jos ehto on tosi (True) suoritetaan tämä koodi
else:
    jos ehto on epätosi (False), suoritetaan tämä koodi
```

Huomaa sisennykset: Pythonissa sisennykset ovat tärkeässä roolissa! Ylläoleva koodi ei toimi, jos if-else-rakennetta ei ole sisennetty.

Vertailuoperaattorit

Ehtolauseissa käytetään hyvin usein vertailuoperaattoreita:

Operaattori	Vertailuoperaattorin merkitys	Esimerkkejä ehtolauseessa
==	Yhtäsuuri kuin	if numero == 1000: if nimi == "tytti":
!=	Erisuuri kuin	if hinta != 10: if vierailija != "loiri":
>	Suurempi kuin	if massa > 55.5:
<	Pienempi kuin	if lampotla < 0.0:
>=	Suurempi tai yhtä suuri kuin	if paine >= 32:
<=	Pienempi tai yhtä suuri kuin	if tilavuus <= 24:

if-else

```
luku = int(input("Anna kokonaisluku:\n"))
if luku >= 0:
    print("Antamasi luku on suurempi tai yhtäsuuri kuin nolla")
else:
    print("Antamasi luku on pienempi kuin nolla")
```

tulostaa

```
Anna kokonaisluku:
> 5
Antamasi luku on suurempi tai yhtäsuuri kuin nolla
```

if-ehtolauseita voi olla useita sisäkkäin (huomaa sisennysten käyttö!):

```
luku = int(input("Anna kokonaisluku:\n"))
if luku >= 0:
    print("Antamasi luku on suurempi tai yhtäsuuri kuin nolla")
    if luku > 1000:
        print("Se on jopa suurempi kuin 1000")
    else:
        print("Se on kuitenkin enintään 1000")
else:
    print("Antamasi luku on pienempi kuin nolla")
```

tulostaa

```
Anna kokonaisluku:
> 999
Antamasi luku on suurempi tai yhtäsuuri kuin nolla
Se on kuitenkin enintään 1000
```

if-elif-else

Ehtolauseeseen voi myös lisätä mielivaltaisen määrän lisäehtoja **elif**-käskyllä:

```
luku = int(input("Anna kokonaisluku: "))
if luku > 1000:
    print("Antamasi luku on suurempi kuin tuhat")
elif luku > 100:
    print("Antamasi luku on suurempi kuin sata")
elif luku > 10:
    print("Antamasi luku on suurempi kuin kymmenen")
elif luku >= 0:
    print("Antamasi luku on välillä 0..10")
else:
    print("Antamasi luku on pienempi kuin nolla")
```

else-osio ei ole pakollinen:

```
kuukausi = input("Mikä kuukausi nyt on?\n")
if kuukausi == "joulukuu":
    print("Joulu tulla jolkottaa")
elif kuukausi == "elokuu":
    print("Vielä on kesää jäljellä")
```

Lisätietoja: Liukulukujen yhtäsuuruuden vertailu

Huom! Liukulukujen yhtäsuuruuden vertailun kanssa pitää olla tarkkana! Yhtäsuuruuden vertailu on parasta tehdä *math.isclose*-funktiolla (lisätietoja 2. kierroksen materiaalissa):

Tehtävä 1.6.1

Täydennä if-elif-else -lause vetämällä sanat oikeille paikoilleen

```
paine = float(input("Anna renkaan paine (bar):\n"))
                paine <= 0.0:
        print("Virheellinen paine")
                paine <= 5.0:
        print("Turvallinen paine")
                paine <= 7.0:
        print("Rajoilla ollaan")
                :
        print("Rengas räjähti")
```

elif | else | if | elif

Check

Tehtävä 1.6.2

```
# Mitä koodi tulostaa, kun käyttäjä antaa arvon 0.25?
konsentraatio = float(input("Anna konsentraatio (mol):\n"))
if konsentraatio <= 0:
    print("Virheellinen konsentraatio")
elif konsentraatio < 0.1:
    print("Laimea")
elif konsentraatio < 0.5:
    print("Keskivahva")
else:
    print("Väkevä")
```

- Keskivahva
- Väkevä
- Laimea
- Ei mitään

Totuusmuuttujat

Ehtolauseissa hyödynnetään usein totuusmuuttujia (**bool**). Totuusmuuttujan arvo on joko **True** tai **False**, joten totuusmuuttujaan on kätevä tallentaa tieto siitä, onko joku ehto täyttynyt ja testata tätä ehtoa myöhemmin:

```
paine = float(input("Anna paine reaktorissa (bar):\n"))
# Jos paine on yli 1 bar, tallennetaan tieto totuusmuuttujaan ylipaine
if paine > 1.0:
    ylipaine = True
else:
    ylipaine = False

T = float(input("Anna lämpötila (K):\n"))
if T > 385.0:
    if ylipaine:
        print("Varoitus! Reaktorissa ylipaine ja korkea lämpötila")
else:
    print("Olosuhteet OK")
```

Huomaa, miten totuusmuuttujaa ylipaine voi käyttää if-ehtolauseessa yksinkertaisesti muodossa

```
if ylipaine:
```

eikä tarvitse siis kirjoittaa

```
if ylipaine == True:
```

Tämä johtuu siitä, että if-ehtolauseen testin arvo on aina True tai False, joten totuusmuuttujan voi laittaa suoraan ehtolauseen testiksi.

Tehtävä 1.7.1

Vedä sanat koodin aukkopaikkoihin niin, että if-ehtolause on kemiallisesti mielekäs

```
yhdiste = input("Anna yhdiste:\n")
if yhdiste == "HCl":
    happo =
elif yhdiste == "H2SO4":
    happo =
else:
    happo =

if happo:
    print("Varoitus, happoa!")
```

True | True | False

Check

Loogiset operaattorit

Loogiset operaattorit toimivat yhdessä totuusmuuttujien kanssa.

not-operaattorilla voi kääntää totuusmuuttujan arvon päinvastaiseksi:

```
if not ylipaine:
    print("Ei vaaraa ylipaineesta")
```

Toinen esimerkki:

```
# Tämän ehdon voisi ilmaista myös näin: if p * V != n * R * T:
if not (p * V == n * R * T):
    print("Ei ideaalikaasu")
```

and-operaattorilla voi yhdistää kaksi totuusmuuttujaa (tai ehtolauseen ehtoa). and-lauseen arvo on True, jos molempien ehtojen arvo on True:

```
if alkuaine1 == "Cu" and alkuaine2 == "O":
    print("Kuparioksidi")

if ylipaine and T > 410.0:
    print("Kriittiset olosuhteet!")
```

or-operaattorilla voi myös yhdistää kaksi totuusmuuttujaa (tai ehtolauseen ehtoa). or-lauseen arvo on True, jos jomman kumman ehdon arvo on True:

```
if kaasu == "He" or kaasu == "Ne":
    print("Jalokaasu")

if T < 200.0 or T > 300.0:
    print("Lämpötila ei ole optimaalinen reaktion kannalta")

# Ehtoja voi myös "ketjuttaa" useammalla or-lauseella:
if kaasu == "He" or kaasu == "Ne" or kaasu == "Ar":
    print("Jalokaasu")
```

Monimutkaisemmat ehdot on parasta ryhmitellä sulkujen avulla:

```
if massa > 200.0 or (tiheys > 22.59 and tilavuus > 10.0):
    print("Kappale on liian painava")
```

Tehtävä 1.8.1

```
# Mikä on muuttujan "prosessi" arvo, kun
# muuttujan T arvo on 50 ja muuttujan p arvo on 2.5?
if T >= 200 and p < 3.0:
    prosessi = 1
elif T < 100 or p < 1.0:
    prosessi = 2
else:
    prosessi = 3
```

- 1
- 3
- 2

Laskujärjestyksestä

Alla on Pythonin operaattorien "arvojärjestys" (*operator precedence*) heikoimmasta vahvimpaan:

Operaattori	Merkitys
or	Looginen operaattori (boolean)
and	Looginen operaattori (boolean)
not	Looginen operaattori (boolean)
<, <=, >, >=, !=, ==	Vertailuoperaattorit
+, -	Yhteen- ja vähennyslasku
*, /, //, %	Kerto- ja jakolasku
**	Potenssiin nosto

Huom! Ylläolevassa taulukossa on listattu vain tällä kurssilla käytettävät operaattorit. Täydellinen lista, joka sisältää esimerkiksi bittioperaatiot, löytyy osoitteesta https://docs.python.org/3/reference/expressions.html#operator-precedence

Aivan kuten matematiikassa, järjestystä voi säätää suluilla:

```
print(4 + 2 * 5)
print((4 + 2) * 5)
```

Tulostaa

```
14
30
```

Loogiset operaattorit ovat siis heikoimpia operaattoreita. Huomaa niiden arvojärjestys: **not** on vahvempi kuin **and**, joka taas on vahvempi kuin **or**:

```
# Tulostaa False, koska 3 > 4 ei ole totta
print(3 > 4)

# Tulostaa True, koska 5 < 6 on totta
print(3 > 4 or 5 < 6)

# Tulostaa False, koska and on vahvempi kuin or ja 7 > 8 ei ole totta
print(3 > 4 or 5 < 6 and 7 > 8)
# Lausekkeen voisi siis selkeyden vuoksi kirjoittaa myös
# 3 > 4 or (5 < 6 and 7 > 8)

# Tulostaa True, koska not kääntää ehdon 7 > 8 arvosta False arvoon True
print(3 > 4 or (5 < 6 and not 7 > 8))
```

Tehtävä 1.9.1

Vedä loogiset operaattorit tyhjiin kohtiin niin, että lauseke tulostaa True

print(99 < 98 55 < 54 (36 < 37 1 > 2))

not

or

or

and

Check

while-silmukka

Silmukkarakenteilla voidaan toistaa tietty koodinpätkä useita kertoja. **while**-silmukassa toistojen määrä riippuu totuusehdosta:

```
luku = 1
while luku <= 5:
    # Huomaa sisennys: silmukka toistaa sisennettyä osaa!
    print(luku)
    luku += 1
    # luku += 1 tarkoitti samaa kuin luku = luku + 1
    # (ks. luku matemaattiset perusoperaattorit)
```

tulostaa

```
1
2
3
4
5
```

Toinen esimerkki, jossa ohjelman suoritus jatkuu silmukan jälkeen ensimmäisestä sisentämättömästä lauseesta:

```
# Alustetaan silmukassa tarvittavat muuttujat
luku = 1.0
lukuja = 0
while luku > 0.0:
    luku = float(input("Anna luku (negatiivinen luku lopettaa):\n"))
    if luku > 0.0:
        lukuja += 1
# Silmukan päätyttyä suoritus jatkuu tästä
print("Annoit yhteensä", lukuja, "positiivista lukua")
```

Esimerkkisuoritus:

```
Anna luku (negatiivinen luku lopettaa):
> 324235
Anna luku (negatiivinen luku lopettaa):
> 12
Anna luku (negatiivinen luku lopettaa):
> 1
Anna luku (negatiivinen luku lopettaa):
> -1
Annoit yhteensä 3 positiivista lukua
```

Huom! Jos totuusehto ei täyty 1. kierroksella, while-silmukkaa ei suoriteta yhtään kertaa!

Ikuinen silmukka

while-silmukkaa käytettäessä ohjelmointivirhe voi johtaa tilanteeseen, jossa totuusehto ei koskaan muutukaan epätodeksi. Tyypillisin virhe on unohtaa silmukkalaskurin päivitys:

```
luku = 1
while luku <= 5:
    print(luku)
    # Tästä on unohtunut laskurin päivitys
    # luku += 1
    # Seurauksena olisi ikuinen silmukka
```

Ikuisesta silmukasta pääsee pois painamalla Ctrl+C (ohjelman keskeytys)

break-käsky

while-silmukasta voi poistua milloin tahansa **break**-käskyllä:

```
# break-käskyä hyödynnettäessä ikuinen silmukkaehtokaan ei ole ongelma
while True:
    luku = int(input("Anna kokonaisluku ja tulostan sen. Luvulla 0 lopetan: "))
    if luku == 0:
        print("Loppu")
        break
    else:
        print("Annoit luvun", luku)
```

Esimerkkitulostus:

```
Anna kokonaisluku ja tulostan sen. Luvulla 0 lopetan: 6
Annoit luvun 6

Anna kokonaisluku ja tulostan sen. Luvulla 0 lopetan: 3
Annoit luvun 3

Anna kokonaisluku ja tulostan sen. Luvulla 0 lopetan: 0
Loppu
```

continue- ja else-käskyt

while-silmukoissa voi lisäksi hyödyntää **continue**-komentoa (hyppää silmukan alkuun) ja **else**-lausetta (suoritetaan silmukan päätyttyä). Näitä emme hyödynnä vielä tässä vaiheessa kurssia.

Tehtävä 1.10.1

Täydennä allaoleva ohjelma niin, että se tekee enintään kymmenen mittausta ja lopettaa arvojen kysymisen, jos mittauksien summa ylittää 100 g.

```
mittaukset = 0
summa = 0
[ ] mittaukset < [ ] and summa < [ ]:
    massa = float(input("Anna mitattu massa (g):\n"))
    if massa > 0:
        summa = [ ] + [ ]
        mittaukset += [ ]
    else:
        print("Virheellinen mittaus")
```

Check

for-silmukka

for-silmukassa toistojen määrä määritellään silmukan alkaessa. Toistojen määrittelyssä auttaa *range*-funktio, jota voi käyttää kolmella eri tavalla: *range*(toistot), *range*(alku, loppu), tai *range*(alku, loppu, askel). Esimerkkejä:

```
# Tulostetaan Hep! viisi kertaa
# Silmukkamuuttujaa "luku" ei hyödynnetä silmukan sisällä
for luku in range(5):
    print("Hep!")
```

tulostaa:

```
Hep!
Hep!
Hep!
Hep!
Hep!
```

Huomaa, että käytettäessä muotoa range(toistot), range-funktio silmukkalaskuri "luku" saa arvot 0 .. toistot - 1. Eli tässä esimerkissä se saa arvot 0, 1, 2, 3 ja 4:

```
for luku in range(5):
    print(luku * 10)
```

Huomaa myös, miten silmukkalaskuri "luku" kasvaa automaattisesti. Koodi tulostaa:

```
0
10
20
30
40
```

Kun range-funktion aloitusarvo määrätään käyttämällä muotoa range(alku, loppu), silmukkalaskuri "luku" saavuttaa arvon loppu - 1:

```
for luku in range(1, 6):
    print(luku)
```

tulostaa

```
1
2
3
4
5
```

Silmukkalaskurin arvoa voi kasvattaa myös isommalla askeleella muodolla range(alku, loppu, askel). Nyt laskuri "luku" saavuttaa arvon loppu - askel.

```
for luku in range(100, 110, 2):
    print(luku)
```

tulostaa:

```
100
102
104
106
108
```

Arvoja voi käydä läpi myös suuremmasta pienempään. Tällöin silmukkalaskuri saavuttaa arvon loppu + 1:

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

tulostaa

```
10
9
8
7
6
```

Myös merkkijonoja voi käydä läpi for-silmukalla:

```
for merkki in "Python":
    print(merkki * 5)
```

tulostaa:

```
PPPPP
yyyyy
ttttt
hhhhh
ooooo
nnnnn
```

for-silmukasta voi poistua **break**-käskyllä samaan tapaan kuin while-silmukasta:

```
maksimi = int(input("Anna positiivinen kokonaisluku ja tulostan kaikki sitä pienemmät kokonaisluvut\n"))
for luku in range(1, maksimi):
    print(luku)
    if luku == 5:
        print("En jaksa enää")
        break
```

tulostaa:

```
Anna positiivinen kokonaisluku ja tulostan kaikki sitä pienemmät kokonaisluvut
> 11
1
2
3
4
5
En jaksa enää
```

Sisäkkäiset silmukat

Sekä for- että while-silmukoita voi olla useampia sisäkkäin. Tässä esimerkki for-silmukalle:

```
for luku1 in range(1, 6):
    # Käytetään print-funktiossa välilyöntiä rivinvaihdon sijasta (end = " ")
    print("Luvun", luku1, "kertotaulu lukuun 10 asti:", end = " ")
    for luku2 in range(1, 11):
        print(luku1 * luku2, end = " ")
    # Tulostetaan tyhjä merkkijono, eli pelkkä rivinvaihto
    print("")
```

tulostaa:

```
Luvun 1 kertotaulu lukuun 10 asti: 1 2 3 4 5 6 7 8 9 10
Luvun 2 kertotaulu lukuun 10 asti: 2 4 6 8 10 12 14 16 18 20
Luvun 3 kertotaulu lukuun 10 asti: 3 6 9 12 15 18 21 24 27 30
Luvun 4 kertotaulu lukuun 10 asti: 4 8 12 16 20 24 28 32 36 40
Luvun 5 kertotaulu lukuun 10 asti: 5 10 15 20 25 30 35 40 45 50
```

Tulemme hyödyntämään for-silmukkaa huomattavan paljon enemmän kolmannesta kierroksesta eteenpäin, kun pääsemme käsittelemään Pythonin tietorakenteita kuten listoja ja sanakirjoja.

Tehtävä 1.11.1

```
# Minkä kokonaisluvun kertoman ohjelma laskee?
kertoma = 1
for luku in range(1,11):
    kertoma = kertoma * luku
print(kertoma)
```

- 11
- 1
- 12
- 10

Download <> Embed

Kierros 2

Toisella kierroksella opettelemme kirjoittamaan ja käyttämään funktioita. Tutustumme mm. str.format-funktioon, jolla on helppo tuottaa siististi muotoiltuja merkkijonoja erilaisista lukuarvoista.

Lisäksi tutustumme moduuleihin, joiden avulla omiin ohjelmiin voi tuoda lukuisia toimintoja erilaisista ohjelmakirjastoista. Hyvä esimerkki tärkeästä moduulista on math-moduuli, joka sisältää paljon matemaattisia funktioita.

Tehtävä 2.0.1.

Tästä alkaa kierros 2. Sitä ennen kierroksen 1 pikakertaus.

---

1 / 9

Your answer Check

Funktiot

Tähän mennessä olemme jo käyttäneet muutamia Pythonin sisäänrakennettuja funktioita kuten *print, input* ja *round*:

- *print*-funktio tulostaa sille annetut parametrit (mutta ei palauta mitään arvoa)
- *input*-funktio tulostaa sille annetun parametrin ja palauttaa merkkijonon
- *round*-funktio pyöristää liukulukuja haluttuun tarkkuuteen tai kokonaisluvuiksi

Lisäksi olemme käyttäneet funktioita tyyppimuunnoksiin:

```
tilavuus = float(input("Anna tilavuus:\n"))
```

Yllä *float*-funktio tekee siis tyyppimuunnoksen merkkijonosta liukuluvuksi.

Pythonissa on useita sisäänrakennettuja funktioita ja erilaiset ohjelmakirjastot sisältävät lukuisia funktioita eri käyttötarkoituksiin.

Tällä kierroksella opit kirjoittamaan omia funktioita. Niiden avulla toistuvien tehtävien suorittaminen helpottuu ja koodin rakenne pysyy selkeämpänä.

Funktioiden määrittely

Funktiolla on tavallisesti joku selkeä tehtävä, esimerkiksi tietty laskutoimitus

- Funktiolla voi olla *parametreja* (ei ole pakko olla)
- Funktio voi palauttaa arvoja (ei ole pakko palauttaa)
- Funktio voi suorituksen aikana tehdä lähes mitä vaan, eli se on periaatteessa *aliohjelma*

Esimerkki 1

Tarkastellaan ohjelmaa, jossa määritellään funktio **tuplaa** ja käytetään sitä:

```
# Määritellään ensin funktio tuplaa käyttäen def-avainsanaa
# Funktio ajetaan vasta, kun sitä kutsutaan pääohjelmasta
def tuplaa(luku):
    # Huomaa, miten funktion sisältö on sisennetty
    return luku * 2

# Pääohjelma alkaa tästä (ei sisennystä)
# Kutsutaan funktiota "tuplaa"
iso_luku = tuplaa(12)
print(iso_luku)
```

- Funktio määritellään avainsanalla *def*, jonka jälkeen tulee funktion nimi (**tuplaa**)
- *tuplaa*-funktiolla on yksi parametri, jonka nimi on **luku** (suluissa nimen jälkeen)
- **return**-avainsanan jälkeen tulee funktion *paluuarvo* (tässä tapauksessa parametri luku kerrottuna kahdella).

- Kun olemme määritelleet funktion *tuplaa*, voimme kutsua sitä pääohjelmassa.
- Lopuksi ohjelma tulostaa 24, eli 12 * 2

Esimerkki 2

Tarkastellaan toista esimerkkiohjelmaa, jossa määritellään funktio **tiheys** ja käytetään sitä:

```
# Määritellään ensin funktio tiheys käyttäen def-avainsanaa
def tiheys(tilavuus, massa):
    # Funktio palauttaa kappaleen tiheyden
    # Funktion parametrit:
    #   Tilavuus: Kappaleen tilavuus (m^3)
    #   Massa:    Kappaleen massa (kg)
    # Jos funktiota kutsutaan epäfysikaalisella parametrilla, se
    # tulostaa virheilmoituksen ja palauttaa arvon -1

    # Tarkistetaan ensin, että parametrit ovat fysikaalisesti mielekkäät
    if tilavuus <= 0:
        print("Virheellinen tilavuus")
        return -1
    elif massa <= 0:
        print("Virheellinen massa")
        return -1
    else:
        return massa / tilavuus

# Pääohjelma alkaa tästä (ei sisennystä)
# Kysytään arvot käyttäjältä
V = float(input("Anna kappaleen tilavuus (m^3):\n"))
m = float(input("Anna kappaleen massa (kg):\n"))
# Kutsutaan tiheys-funktiota annetuilla arvoilla
rho = tiheys(V, m)
# Tarkistetaan funktion paluuarvo. -1 tarkoittaa virhettä
if rho == -1:
    print("Tiheyden laskeminen epäonnistui")
else:
    print("Kappaleen tiheys on:", round(rho,3), "kg/m^3")
```

- Tässä esimerkissä funktion "tiheys" suorittama laskutoimitus oli hyvin yksinkertainen.
- Oikeissa ohjelmissa funktio voi suorittaa hyvinkin monimutkaisia operaatioita. Nämä monimutkaiset toiminnot kannattaa nimenomaan "paketoida" funktioihin
- Koodin testaaminen ja virheiden etsiminen on helpompaa, kun se on jaettu funktioihin
- Hyvin kirjoitetut ja dokumentoidut funktiot ovat helposti uudelleenkäytettävissä uusissa ohjelmissa

Esimerkki 3

Tässä tapauksessa meillä on funktio *kysy_suure*, joka hoitaa vuorovaikutuksen käyttäjän kanssa:

```
# Ensin määritellään funktio. Sitä kutsutaan pääohjelmasta.
def kysy_suure(suure):
    # Funktio kysyy liukulukua käyttäjältä, kunnes annettu arvo on > 0
    # Parametri suure on merkkijono, esim. "massa (g)"
    arvo = -1
    while arvo <= 0:
        arvo = float(input("Anna " + suure + ":\n"))
        if arvo > 0:
            return arvo
        else:
            print("Virheellinen arvo")

# Pääohjelma alkaa täältä
# Kysytään massa ja moolimassa funktion kysy_suure avulla
moolimassa = kysy_suure("moolimassa (g/mol)")
massa = kysy_suure("massa (g)")
n = massa / moolimassa
print("Ainemäärä on", round(n,2), "mol")
```

Etuna on se, että virheellisten arvojen käsittely while-silmukan avulla tarvitsee kirjoittaa vain kerran. Jos emme käyttäisi funktiota, ratkaisu voisi näyttää tältä:

```
# Luetaan moolimassa
arvo = -1
while arvo <= 0:
    arvo = float(input("Anna moolimassa (g/mol):\n"))
    if arvo > 0:
        moolimassa = arvo
    else:
        print("Virheellinen arvo")

# Luetaan massa
arvo = -1
while arvo <= 0:
    arvo = float(input("Anna massa (g):\n"))
    if arvo > 0:
        massa = arvo
    else:
        print("Virheellinen arvo")

n = massa / moolimassa
print("Ainemäärä on", round(n,2), "mol")
```

- Jälkimmäinen ratkaisu ei ole kovin paljon ensimmäistä pidempi, mutta kuvittele tilanne, jossa suureita pitäisi lukea huomattavasti useampia kuin kaksi. Tällöin funktio *kysy_suure* helpottaisi koodin kirjoittamista merkittävästi.
- Jos koodiin täytyisi tehdä joku muutos, esimerkiksi vaihtaa virheilmoitus "Virheellinen arvo" joksikin muuksi, ensimmäisessä tapauksessa meille riittäisi funktion *kysy_suure* päivittäminen.

Tehtävä 2.1.1

Täytä aukkopaikat niin, että funktio *kaiku* toimii määritelmän mukaisesti.

[ ] kaiku(huuto):

    # Palauttaa parametrin huuto kolme kertaa toistettuna, välilyönnillä eroitettuna

    [ ] huuto + " " + [ ] + " " + huuto

Check

Tehtävä 2.1.2

Mitä allaoleva ohjelma tulostaa?

```
def tulosta_suurempi(luku1, luku2):
    if luku1 > luku2:
        print("Luku", luku1, "on suurempi")
    elif luku2 > luku1:
        print("Luku", luku2, "on suurempi")
    else:
        print("Luvut ovat yhtäsuuret")

tulosta_suurempi(44, 653)
```

- ○ Ei mitään
- ○ Luku 44 on suurempi
- ○ Luvut ovat yhtäsuuret
- ○ Luku 653 on suurempi

Check

Erilaisia funktioita

Tässä osiossa on useita esimerkkejä erilaisista funktiosta. Esimerkkejä on parasta havainnoistaa kopioimalla koodi Spyderiin ja ajamalla se itse.

1. Funktiolla ei tarvitse välttämättä olla yhtään parametria:

```
def pii():
    # Funktio palauttaa piin arvon 15 desimaalin tarkkuudella
    return 3.141592653589793

r = 1.5
pallon_tilavuus = 4 * pii() * r**3 / 3
print(round(pallon_tilavuus, 2))
```

2. Funktiolla voi olla useita parametreja:

```
def ainemaara(massa, moolimassa):
    return massa / moolimassa

n = ainemaara(5.4, 18.02)
print(round(n, 3))
```

3. Funktiolla ei ole pakko olla paluuarvoa (*return*):

```
def tervehdys(kieli):
    if kieli == "suomi":
        teksti = "Hei!"
    elif kieli == "ruotsi":
        teksti = "Hej!"
    elif kieli == "saksa":
        teksti = "Hallo!"
    else:
        teksti = "!!??"
    print(teksti)

tervehdys("suomi")
```

4. Funktiolla voi olla useita paluuarvoja:

```
def tunnit_ja_minuutit(minuutit_yhteensa):
    tunnit = minuutit_yhteensa // 60 # katkaiseva jakolasku
    minuutit = minuutit_yhteensa % 60 # jakojäännös
    return tunnit, minuutit

luku = int(input("Anna minuuttien määrä kokonaislukuna:\n"))
h, m = tunnit_ja_minuutit(luku)
print(luku, "minuuttia on", h, "tuntia ja", m, "minuuttia")
```

tulostaa:

```
Anna minuuttien määrä kokonaislukuna:
> 124
124 minuuttia on 2 tuntia ja 4 minuuttia
```

5. Funktio voi sisältää useita return-käskyjä, mutta vain yksi niistä voi toteutua:

```
def itseisarvo(luku):
    if luku >= 0:
        return luku
    else:
        return -luku

print(itseisarvo(5.4))
print(itseisarvo(-5.4))
```

6. return-lause yksinkertaistaa parametrien arvojen tarkistamista

```
def ratkaise_p(V, n, T):
    # Ratkaistaan paine ideaalikaasun tilanyhtälön avulla
    # Parametrien yksiköt: V (m^3), n (mol), T(K)

    # Jos joku parametreista on epäfysikaalinen,
    # funktio palauttaa välittömästi arvon -1
    if V <= 0 or n <= 0 or T <= 0:
        return -1

    # Ylläolevan if-lauseen return-käsky hoitaa virheelliset parametrit
    # Jos koodi jatkaa tänne asti, tiedämme, että parametrit ovat OK
    R = 8.3144598 # J K^-1 mol^-1
    p = n * R * T / V
    return p # Pa

print(ratkaise_p(0.25, 1.25, 300))
```

7. Funktiot voivat kutsua toisiaan:

```
def tervehdys(kieli):
    if kieli == "suomi":
        teksti = "Hei!"
    elif kieli == "ruotsi":
        teksti = "Hej!"
    elif kieli == "saksa":
        teksti = "Hallo!"
    else:
        teksti = "!!??"
    print(teksti)

def keskustelu(kieli1, kieli2):
    tervehdys(kieli1)
    tervehdys(kieli2)

keskustelu("ruotsi", "saksa")
```

tulostaa:

```
Hej!
Hallo!
```

8. Valinnaiset parametrit

Funktioilla voi olla myös valinnaisia parametreja, joille on määritelty oletusarvo. Jos funktiota kutsutaan ilman valinnaista parametria, Python käyttää oletusarvoa. Tuttu esimerkki on *print*-funktio, jolla on useita valinnaisia parametrejä. Yksi niistä on *end*-parametri, jonka oletusarvo on rivinvaihto "\n". Kaksi tavallista funktiokutsua

```
print("Moi!")
print("Moi!")
```

tulostaa

```
Moi!
Moi!
```

Kun taas vaihtamalla end-parametri tyhjäksi merkkijonoksi:

```
print("Moi!", end="")
print("Moi!", end="")
```

tulostuu

```
Moi!Moi!
```

Esimerkki valinnaisten parametrien määrittelystä:

```
def ratkaise_tilavuus(n, T = 273.15, p = 101325):
    # Ratkaisee tilavuuden ideaalikaasun tilanyhtälöstä
    # Kaikki suureet SI-yksiköissä
    # Parametreillä p ja T on oletusarvot (NTP-olosuhteet)
    R = 8.3144598 # J K^-1 mol^-1
    V = n * R * T / p
    return V

# Selvennä aina funktiota kutsuessasi, minkä valinnaisen parametrin haluat antaa
V1 = ratkaise_tilavuus(0.28) # Pelkästään pakollinen parametri n
V2 = ratkaise_tilavuus(0.28, T = 400) # n ja valinnainen parametri T
V3 = ratkaise_tilavuus(0.28, T = 300, p = 200000) # n ja molemmat valinnaiset parametrit
print(round(V1, 5), round(V2, 5), round(V3, 5))
```

HUOM! Valinnaiset parametrit pitää aina määritellä vasta pakollisten parametrien jälkeen. Muuten Python antaa virheilmoituksen:

```
SyntaxError: non-default argument follows default argument
```

Tehtävä 2.2.1.

Montako parametria funktiolla *ratkaise_p* on?

```
def ratkaise_p(V, n, T):
    R = 8.3144598 # J K^-1 mol^-1
    return n * R * T / V
```

- 3
- 1
- 4
- Ei yhtään

Muotoiltu tulostaminen str.format-funktiolla

Tähän asti olemme käyttäneet *print*-funktiota tulostamiseen varsin suoraviivaisesti:

```
alkuaine = "C"
atomipaino = 12.011
print("Alkuaineen", alkuaine, "atomipaino on", atomipaino)
```

tulostaa

```
Alkuaineen C atomipaino on 12.011
```

Pythonissa on kuitenkin käytettävissä myös erittäin monipuolinen *str.format*-funktio, jolla voi muotoilla merkkijonon:

```
alkuaine = "C"
atomipaino = 12.011
print("Alkuaineen {} atomipaino on {}".format(alkuaine, atomipaino))
```

tulostaa

```
Alkuaineen C atomipaino on 12.011
```

Merkkijonon "Alkuaineen {} atomipaino on {}" kaarisulut korvautuivat siis format-funktion parametreilla *alkuaine* ja *atomipaino*.

{}-kentän muotoilu

str.format-funktion {}-kenttää voi muotoilla lukuisilla eri tavoilla. Sen tyypillisin käyttötapa on **{:<leveys>.<tarkkuus><tyyppi>}**. Muutamia esimerkkejä:

- liukuluku (f), 6 merkkiä leveä kenttä, pyöristettynä nollan desimaalin tarkkuuteen: **{:6.0f}**
- liukuluku (f) pyöristettynä kolmen desimaalin tarkkuuteen, automaattinen kentän leveys: **{:.3f}**
- kokonaisluku (d), automaattinen kentän leveys
- kokonaisluku (d), 5 merkkiä leveä kenttä: **{:5d}**

Esimerkki 1:

```
T = 300 # K
p = 1.12345 # atm
print("Olosuhteet ovat: {:d} K, {:.3f} atm".format(T, p))
```

tulostaa

```
Olosuhteet ovat: 300 K, 1.123 atm
```

Esimerkki 2:

```
n = 0.25 # mol
V = 0.00456 # m^3
T = 298.15 # K
R = 8.3145 # J/(mol K)
p = n * R * T / V # J/m^3
print("Kun n = {:3.2f} mol, V = {:7.5f} m^3, T = {} K, on paine p = {:6.0f} J/m^3".format(n, V, T, p))
```

tulostaa

```
Kun n = 0.25 mol, V = 0.00456 m^3, T = 298.15 K, on paine p = 135908 J/m^3
```

Vaikka str.format-funktion kokoaminen voi ensi alkuun vaikuttaa työläältä, on se todella paljon kätevämpää kuin pelkän print- ja round-käskyn käyttö.

Käytä lukuarvojen tulostamiseen tästä lähtien str.format-funktiota aina kun mahdollista.

str.format-funktion dokumentaatio löytyy osoitteesta https://docs.python.org/3/library/string.html#formatstrings. Dokumentaatio on hieman abstrakti, mutta sisältää myös esimerkkejä.

Muita str.format-funktion käyttötapoja

Ennen kaarisulkujen sisältämän muotoilukentän kaksoispistettä voi käyttää tunnistetta, joka yhdistää kentän *str.format*-funktion parametriin:

```
print("Olosuhteet ovat: {T_K:d} K, {p_atm:.3f} atm".format(T_K=300, p_atm=1.12345))
```

tulostaa

```
Olosuhteet ovat: 300 K, 1.123 atm
```

str.format-funktion argumentteja voi toistaa helposti käyttämällä kaarisulkujen sisällä tunnisteita:

```
alkuaine = "C"
atomipaino = 12.011
naapuri = "N"
print("Alkuaineen {aine} atomipaino on {paino:.3f}. "
      "Alkuaineen {aine} naapuri on {aine2}".format(aine=alkuaine, paino=atomipaino, aine2=naapuri))
```

tulostaa

```
Alkuaineen C atomipaino on 12.011. Alkuaineen C naapuri on N
```

Huomaa myös esimerkistä, miten pitkää merkkijonoa voi jatkaa koodissa toiselle riville yksinkertaisesti sulkemalla lainausmerkit ja aloittamalla uudet seuraavalla rivillä.

Tehtävä 2.3.1.

Mitä tulostaa print("{:.1f}".format(15.2355))

- 15.2
- 15.0
- 15
- 15.2355

Moduulit

Suuremmat ohjelmakokonaisuudet on aina parasta jakaa *moduuleiksi*. Moduulien avulla ohjelman rakenne pysyy paremmin hallinnassa ja moduuleja voi käyttää helposti uudelleen toisissa ohjelmissa.

Käytetään esimerkkinä moduulia ideaalikaasu, joka käytännössä olisi siis alla oleva koodi tallennettuna tiedostoon *ideaalikaasu.py*:

```
# Moduuli ideaalikaasu:
# Apufunktioita ideaalikaasulle
# pV = nRT

# Moduuli määrittelee myös kaasuvakion R
# Lähde NIST CODATA: https://physics.nist.gov/cgi-bin/cuu/Value?r
R = 8.3144598 # J K^-1 mol^-1

# Moduuli määrittelee neljä funktiota
def ratkaise_paine(V, n, T):
    return n * R * T / V

def ratkaise_tilavuus(p, n, T):
    return n * R * T / p

def ratkaise_ainemaara(p, V, T):
    return p * V / (R * T)

def ratkaise_lampotila(p, V, n):
    return p * V / (n * R)
```

Luodaan moduulin *ideaalikaasu.py* kanssa samaan hakemistoon tiedosto *testi.py*, jossa hyödynnämme ideaalikaasu-moduulia **import**-avainsanan avulla:

```
# Tuodaan koko ideaalikaasu-moduuli ohjelman testi.py käyttöön
import ideaalikaasu
# ideaalikaasu-moduulin funktioiden eteen pitää lisätä viittaus "ideaalikaasu."
p = ideaalikaasu.ratkaise_paine(0.002, 0.01, 300) # Parametrit V, n, T
print(round(p, 3))
```

Toinen tapa on tuoda *ideaalikaasu*-moduulista vain tietyt funktiot ja muuttujat *testi.py*-ohjelman käyttöön. Tähän käytetään käskyä **from** MODUULI **import** FUNKTIOT

```
# Tuodaan tietyt funktiot (ja/tai muuttujat) ohjelman testi.py käyttöön
from ideaalikaasu import ratkaise_paine, ratkaise_tilavuus, R
# Nyt meidän ei tarvitse käyttää "ideaalikaasu."-viittausta
p = ratkaise_paine(0.002, 0.01, 300) # Parametrit V, n, T
V = ratkaise_tilavuus(101325, 0.01, 300) # Parametrit p, n, T
print(round(p, 3))
print(round(V, 5))
print("Kaasuvakion R arvo on", R, "J/mol K")
```

Vähänkin laajemissa ohjelmakokonaisuuksissa kannattaa miettiä ohjelman pilkkomista helpommin ylläpidettäviin ja uudelleenkäytettäviin moduuleihin.

import-käskyyn voi yhdistää **as**-avainsanan, jolloin ohjelmaan tuotavan moduulin nimeä voi vaikkapa lyhentää. Käsky on tällöin **import** MODUULI **as** LYHENNE:

```
import ideaalikaasu as ik
p = ik.ratkaise_paine(0.002, 0.01, 300) # Parametrit V, n, T
```

Tehtävä 2.4.1.

Täytä puuttuvat kohdat niin, että 1) ohjelmaan tuodaan moduuli hiilivety ja käytetään sieltä funktiota laske_CO2; 2) ohjelmaan tuodaan funktio kysy_suure moduulista apufunktiot ja käytetään kyseistä funktiota.

import [ ]
from [ ] import [ ]

m_CH4 = [ ]("Anna poltettavan metaanin massa (g):")
m_CO2 = [ ].laske_CO2(m_CH4)

print("{:.3f} g CH4 tuottaa palaessaan {:.3f} g CO2".format(m_CH4, m_CO2))

Check

math-moduuli

Yksi hyödyllisimmistä Pythonin moduuleista on *math*-moduuli, joka sisältää perustavanlaatuisia matemaattisia funktioita ja vakioita.

Ensin math-moduuli täytyy tuoda ohjelmaan **import**-käskyllä:

```
import math
```

Tämän jälkeen moduulin funktioita ja vakioita voi käyttää näin:

```
# exp(x) -> Eksponenttifunktio e^x
print(math.exp(4))

# log(x) -> Luvun x luonnollinen logaritmi, ln(x)
print(math.log(54.598150033144236))

# log(x, y) -> Luvun x logaritmi, kantaluku y
print(math.log(8, 2))

# log10(x) -> Luvun x 10-kantainen logaritmi
print(math.log10(10000))

# pow(x, y) -> luku x potenssiin y. Sama kuin x**y, mutta muuntaa aina luvut (ja tuloksen) liukuluvuksi
print(math.pow(3, 2))

# sqrt(x) -> Luvun x neliöjuuri (kuten x**(1/2))
print(math.sqrt(9))

# pi -> pii (ei ole funktio vaan vakio)
print(math.pi)

# e -> Neperin luku (ei ole funktio vaan vakio)
print(math.e)

# sin(x), cos, tan, ... -> trigonometriset funktiot
print(math.sin(math.pi / 2))

# degrees(x) -> muuntaa radiaanit asteiksi
print(math.degrees(math.pi))

# radians(x) -> muuntaa asteet radiaaneiksi
print(math.radians(180))

# ceil(x) -> pyöristä kokonaislukuun ylöspäin
print(math.ceil(5.4))

# floor(x) -> pyöristä kokonaislukuun alaspäin
print(math.floor(5.6))
```

Math-moduulin dokumentaatio ja listaus funktioista löytyy osoitteesta https://docs.python.org/3/library/math.html

Liukulukujen yhtäsuuruuden vertailu math.isclose-funktiolla

Liukulukujen yhtäsuuruuden vertailuun ei pidä käyttää == -operaattoria vaan math.isclose-funktiota. Tällöin voit itse määritellä tarkkuuden, jolla liukulukuja verrataan:

```
import math
luku1 = 2.0
luku2 = 1.0 * 2.00000001
print("Luku 1 on:", luku1)
print("Luku 2 on:", luku2)
print("Vertailuoperaatio luku1 == luku2:", luku1 == luku2)
# Jos luku1 ja luku2 eroavat vahemman kuin 10^-6 (parametri rel_tol),
# math.isclose palauttaa True
print("math.isclose(luku1, luku2, rel_tol = 1e-06):", math.isclose(luku1, luku2, rel_tol = 1e-06))
```

lopputulos

```
Luku 1 on: 2.0
Luku 2 on: 2.00000001
Vertailuoperaatio luku1 == luku2: False
math.isclose(luku1, luku2, rel_tol = 1e-06): True
```

Tehtävä 2.5.1.

Vedä sanat oikeisiin lokeroihin

```
            math
# Täydennä niin, että tulostuu 0
print(math.          (11.1) - math.          (12.9))

# Täydennä niin, että tulostuu 5
print(round(math.          (          .log(5))))

# Täydennä niin, että tulostuu 1
x = 0.15
print(round(math.          (math.          (x), 2) + math.          (math.          (x), 2)))
```

import
pow
floor
exp
cos
pow
sin
math
ceil

Check

Muuttujien näkyvyys

Tärkeää: Funktion sisällä määritellyt muuttujat, eli **lokaalit muuttujat** näkyvät vain kyseisessä funktiossa:

```
def ratkaise_p(V, n, T):
    R = 8.3144598 # Lokaali muuttuja (vakio), ei näy funktion ulkopuolelle
    if V > 0 and n > 0 and T > 0:
        p = n * R * T / V
    else:
        p = 0
    return p

paine = ratkaise_p(0.025, 0.30, 300)
print("Paine (Pa) on:", round(paine))

# Tämä komento EI toimisi, koska kaasuvakio R on määritelty
# vain funktion ratkaise_p sisällä:
# print("Kaasuvakio (J K^-1 mol^-1) on:", round(R))
```

Tärkeää: Funkion lokaalien muuttujien arvot "unohtuvat" samalla hetkellä kun funktiosta poistutaan! Et siis voi tallentaa lokaaleihin muuttujiin mitään pysyvää tietoa.

Globaalit muuttujat

Yleensä muuttujat kannattaa välittää funktiolle parametreina. Joskus voi silti olla tarpeen käyttää ns. *globaaleja muuttujia*.

Allaolevassa esimerkissä hyödynnetään globaalia muuttujaa *paine.* Myös ATM_TO_PA on kaikkien funktioiden käytössä, mutta se on vakio, ei muuttuja (isot kirjaimet viittaavat vakioon, jota ei tule muuttaa, ks. seuraava luku).

```
ATM_TO_PA = 101325 # Muuntokerroin atm -> Pa

def muuta_painetta(muutos, yksikko):
    # Muutetaan globaalia muuttujaa paine funktion sisällä.
    # Tällöin globaali muuttuja pitää määritellä avainsanalla global
    # "yksikko" on joko 'Pa' tai 'atm'
    global paine
    if yksikko == 'Pa':
        paine = paine + muutos
    elif yksikko == 'atm':
        paine = paine + muutos * ATM_TO_PA

def raportoi_paine():
    # Tulostetaan paine käyttäen globaalia muuttujaa "paine"
    # Huomaa, että jos globaalin muuttujan arvo halutaan vain *lukea*,
    # muuttujaa ei tarvitse määritellä global-avainsanalla
    print("Autoklaavin paine on tällä hetkellä", round(paine, 2), "Pa")

# Pääohjelma: alustetaan globaali muuttuja "paine" yhden ilmakehän paineeseen
paine = 1 * ATM_TO_PA
raportoi_paine()

print("Reaktio käynnistyy...")
muuta_painetta(4, 'atm') # Muuttaa globaalin muuttujan "paine" arvoa
raportoi_paine()

print("Reaktio päättyi!")
muuta_painetta(-3.8, 'atm') # Muuttaa globaalin muuttujan "paine" arvoa
raportoi_paine()
```

tulostaa

```
Autoklaavin paine on tällä hetkellä 101325 Pa
Reaktio käynnistyy...
Autoklaavin paine on tällä hetkellä 506625 Pa
Reaktio päättyi!
Autoklaavin paine on tällä hetkellä 121590.0 Pa
```

Huomaa, että tässä tapauksessa sama lopputulos olisi voitu saavuttaa myös funktioiden parametreja ja paluuarvoja käyttämällä:

```
ATM_TO_PA = 101325 # Muuntokerroin atm -> Pa

def muuta_painetta(paine, muutos, yksikko):
    if yksikko == 'Pa':
        return paine + muutos
    elif yksikko == 'atm':
        return paine + muutos * ATM_TO_PA

def raportoi_paine(paine):
    print("Autoklaavin paine on tällä hetkellä", round(paine, 2), "Pa")

# Pääohjelma: alustetaan muuttuja "paine" yhden ilmakehän paineeseen
paine = 1 * ATM_TO_PA
raportoi_paine(paine)

print("Reaktio käynnistyy...")
paine = muuta_painetta(paine, 4, 'atm')
raportoi_paine(paine)

print("Reaktio päättyi!")
paine = muuta_painetta(paine, -3.8, 'atm')
raportoi_paine(paine)
```

Globaalien muuttujien käyttäminen voi olla perusteltua, jos se yksinkertaistaa koodia huomattavasti. *global*-avainsanan ajatus on, että ohjelmoijan pitää erikseen kertoa, jos hän haluaa muokata globaalia muuttujaa ja näin vältytään muokkaamasta globaalia muuttujaa vahingossa.

Vakioiden määrittely

Usein ohjelmissa on hyvä määritellä joitain kiinteitä arvoja, jotka eivät muutu ajon aikana. Pythonissa ei ole varsinaista vakion käsitettä samaan tapaan kuin monissa muissa ohjelmointikielissä. Hyvä käytäntö on

- Nimeä vakio ISOILLA KIRJAIMILLA
- Määrittele vakion arvo
- Älä koskaan muuta vakion arvoa sen määrittelemisen jälkeen. Jos sinun täytyy muuttaa arvoa, kyseessä ei ole vakio vaan muuttuja.

Tyypillisiä vakioita ovat vaikkapa luonnonvakiot ja muuntokertoimet. Esimerkki:

```
ATM_TO_PA = 101325 # Muuntokerroin atm -> Pa on vakio

p_atm = float(input("Anna paine (atm) niin muunnan sen pascaleiksi (Pa):\n"))
p_Pa = p_atm * ATM_TO_PA
print("{:.3f} atm on {:.0f} Pa".format(p_atm, p_Pa))
```

tulostaa

```
Anna paine (atm) niin muunnan sen pascaleiksi (Pa):
0.454
0.454 atm on 46002 Pa
```

Kun muuntokerroin on määritelty vakiona yhdessä paikassa, **pienenee myös inhimillisten virheiden määrä**. Näin muuntokertoimelle ei tule vahingossa käytettyä eri arvoa eri paikoissa. Jos olet kirjoittamassa laajempaa ohjelmaa, jossa käytetään useita luonnonvakioita, on yleensä hyvä ratkaisu määritellä kaikki luonnonvakiot omassa moduulissaan (esim. *luonnonvakiot.py*) ja ottaa tämä moduuli käyttöön tarpeen mukaan.

Kierros 3

Kolmannella kierroksella opettelemme käyttämään erilaisia *tietorakenteita*. Tutustumme mm. *listoihin*, *monikkoihin* ja *sanakirjoihin*. Tietorakenteiden avulla suuretkin datamäärät pysyvät hyvin järjestyksessä.

Tehtävä 3.0.1.

Tästä alkaa kierros 3. Sitä ennen kierroksen 2 pikakertaus.

---

1 / 7

Avainsana, jolla aloitetaan funktion määrittely?

Your answer Check

Käsky, jolla määritellään funktion paluuarvo?

Your answer Chec

Pythonin tietorakenteita

Tähän mennessä olemme tutustuneet yksinkertaisiin tietotyyppeihin kuten int, float, str ja bool. Nämä tietotyypit ovat yksinkertaisia, koska niihin tallennetaan käytännössä vain yksi arvo, kuten yksi kokonaisluku. Mutta entä jos haluaisimme säilöä vaikka 1000 kokonaislukua? Emme varmaankaan haluaisi määritellä tuhatta muuttujaa?

Otetaan nyt käyttöön monimutkaisempia tietorakenteita, joiden avulla voi hallita suuria tietomääriä. Pythonissa on useita erilaisia tietorakenteita eri käyttötarkoituksiin. Alla on listattu lyhyesti esimerkkejä, joita kuvataan tarkemmin seuraavissa kappaleissa.

Lista

lista (**list**) on erittäin joustava tietorakenne. Listat määritellään hakasulkeiden avulla:

```
tilavuudet = [10.2, 2.6, 3.55]
```

Listan yksittäistä arvoa kutsutaan listan **alkioksi**. Ylläolevassa listassa on siis kolme alkiota.

Monikko

monikko (**tuple**) on kuten lista, mutta sitä ei voi muokata. Monikot määritellään tavallisten sulkeiden avulla:

```
jalokaasut = (’He’, ’Ne’, ’Ar’, ’Kr’, ’Xe’, ’Rn’)
```

Kuten listojen kohdalla, myös monikon yksittäinen arvo on monikon **alkio**. Ylläolevassa listassa on siis kuusi alkiota.

Sanakirja

sanakirja (**dictionary**) on *avain*:*arvo* -parien joukko, jolla ei ole järjestystä. Avainten tulee olla uniikkeja. Sanakirjat määritellään kaarisulkeiden avulla:

```
atomipainot = {’H’:  1.008, ’C’: 12.011, ’O’: 15.999}
```

Ylläolevassa sanakirjassa on siis kolme avain:arvo -paria.

Joukko

joukko (**set**) on tietorakenne, jossa kukin arvo voi esiintyä vain kerran. Emme hyödynnä joukkoja tällä kurssilla. Joukot määritellään kaarisulkeilla, mutta toisin kuin sanakirjat, joukot koostuvat yksittäisistä arvoista ilman avaimia:

```
metallit = {’Cu’, ’Ag’, ’Cu’, ’Ag’}
```

Ylläolevan määrittelyn jälkeen *metallit*-joukon sisältö on {’Cu’, ’Ag’}, eli vain uniikit arvot on tallennettu joukkoon.

Tehtävä 3.1.1.

Mikä tietorakenne on kyseessä:

suureet = ("paine", "tilavuus", "ainemäärä", "lämpötila")

- Monikko
- Lista
- Joukko
- Sanakirja

Listat

Yhtä tietotyyppiä sisältävät listat

Alla on esimerkkejä yksinkertaisista listoista (**list**), joissa on pelkästään yhden tyyppisiä arvoja:

```
# Kokonaislukuja sisältävä lista, viisi alkiota
kokonaisluvut = [5, 6, 7, 8, 9]

# Liukulukuja sisältävä lista, kolme alkiota
liukuluvut = [0.3, 0.33333, 355.555]

# Merkkijonoja sisältävä lista, neljä alkiota
merkkijonot = ["Kupari", "Hopea", "Kulta", "Roentgenium"]

# Tyhjä lista (pelkät hakasulkeet)
vakuumi = []
```

Listan pituus

Listan pituuden voi selvittää *len*-funktiolla:

```
jalokaasut = ["He", "Ne", "Ar", "Kr", "Xe", "Rn"]
print("Jalokaasut: ", jalokaasut)
print("Jalokaasujen määrä: ", len(jalokaasut))
```

tulostaa

```
Jalokaasut: ['He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn']
Jalokaasujen määrä: 6
```

Huomaa, että kun Python tulostaa merkkijonoja sisältävän listan, se käyttää yksinkertaisia lainausmerkkejä ('He'). Tämä on aivan sama kuin "He".

Listojen indeksointi

Listan alkioilla on *indeksi*, jolla niihin voi viitata. **Huom!** Indeksointi alkaa nollasta.

```
jalokaasut  = ['He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn']
# indeksi:       0     1     2     3     4     5
print(jalokaasut[0])
print(jalokaasut[3])
```

tulostaa:

```
He
Kr
```

Alkioihin voi viitata myös negatiivisella indeksillä. Tällöin viimeisen alkion indeksi on -1. Negatiivisen indeksoinnin etuja on mm. se, ettei tarvitse käyttää *len*-funktiota viimeisen alkion osoittamiseksi:

```
jalokaasut = ['He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn']
# neg. indeksi: -6    -5    -4    -3    -2    -1
print(jalokaasut[-1]) # Palauttaa viimeisen alkion
print(jalokaasut[-2]) # Palauttaa toiseksi viimeisen alkion
print(jalokaasut[len(jalokaasut) - 1]) # Toinen tapa palauttaa viimeinen alkio
```

tulostaa

```
Rn
Xe
Rn
```

Listojen siivuttaminen

Listasta voi valita useita alkoita kerralla, jolloin tulos on uusi lista. Tätä kutsutaan listan *siivuttamiseksi* (slicing)

```
lista[alku:loppu]       # indeksistä alku indeksiin loppu-1
lista[alku:]            # indeksistä alku alkaen listan loppuun asti
lista[:loppu]           # listan alusta indeksiin loppu-1 asti
lista[alku:loppu:askel] # indeksistä alku indeksiin loppu-1, käyttäen askelväliä askel
lista[:]                # Kopio listan kaikista alkioista
```

eli käytännön esimerkit:

```
jalokaasut = ['He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn']
# indeksi:      0     1     2     3     4     5

print(jalokaasut[2:4])   # uusi lista ['Ar', 'Kr']
print(jalokaasut[:3])    # uusi lista ['He', 'Ne', 'Ar']
print(jalokaasut[3:])    # uusi lista ['Kr', 'Xe', 'Rn']
print(jalokaasut[0:6:2]) # uusi lista ['He', 'Ar', 'Xe']
# Viimeisessä esimerkissä poimitaan siis joka toinen alkio käyttämällä askelta 2
```

Listan täyttäminen range-funktion avulla

for-silmukoiden yhteydessä tutustuimme *range*-funktioon, jolla voi luoda numerosarjoja. range-funktion avulla voi myös täyttää listoja:

```
parilliset = list(range(2, 11, 2))
kymmenet = list(range(10, 101, 10))
print(parilliset)
print(kymmenet)
```

tulostaa

```
[2, 4, 6, 8, 10]
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Useita tietotyyppejä sisältävät listat

Lista on erittäin monipuolinen tietorakenne ja yksi lista voi sisältää useampia tietotyyppejä:

```
yhdiste = ['C', 2, 'H', 6, 'O', 1]  # str ja int
luvut = [0, 0.5, 1, 1.5, 2, 2.5, 3] # int ja float
```

Syventävää tietoa: listan "purkaminen" funktion parametreiksi

Joillekin funktiolle voi antaa listan "puretussa" muodossa (*unpacking*). Tällöin parametrina annettavan listan nimen eteen lisätään *-merkki:

```
jalokaasut = ['He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn']
print(jalokaasut)
print(*jalokaasut)
# Jälkimmäinen on sama asia kuin
# print('He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn')
```

tulostaa

```
['He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn']
He Ne Ar Kr Xe Rn
```

Ensimmäisessä tapauksessa jalokaasut-lista välittyi print-funktiolle listana ja sellaisena se myös tulostui. Jälkimmäisessä tapauksessa lista "purettiin" kuudeksi erilliseksi parametriksi ja print-funktio tulosti nämä parametrit välilyönnillä erotettuina.

Syventävää tietoa: listan kopioiminen

Edellä mainittiin komento lista[:], jolla voi luoda **kopion** listasta. Käytännön esimerkki, jossa luodaan kopio listasta ja kopion muokkaaminen ei vaikuta alkuperäiseen listaan:

```
jalokaasut = ['He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn']
# indeksi:      0     1     2     3     4     5

jalokaasut_kopio = jalokaasut[:]
print(jalokaasut_kopio[1]) # Tulostaa Ne
jalokaasut_kopio[1] = "Neon"
print(jalokaasut_kopio[1]) # Tulostaa Neon
print(jalokaasut[1])       # Tulostaa Ne
```

Listojen kanssa yksinkertainen sijoitus *jalokaasut2 = jalokaasut* ei enää toimikaan samalla tavalla kuin yksinkertaisten tietotyyppien (kuten int) kanssa. Komennon jälkeen lista *jalokaasut2* **viittaa** alkuperäiseen listaan *jalokaasut* ja listan *jalokaasut2* muokkaaminen muokkaa myös alkuperäistä listaa *jalokaasut:*

```
jalokaasut_viittaus = jalokaasut
print(jalokaasut_viittaus[1]) # Tulostaa Ne
jalokaasut_viittaus[1] = "Neon"
print(jalokaasut_viittaus[1]) # Tulostaa Neon
print(jalokaasut[1])          # Tulostaa Neon
```

Tähän toimintatapaan on omat järkevät syynsä, kuten muistin säästäminen. Tämän kurssin puitteissa emme käsittele ylläolevan kaltaisia viittauksia tietorakenteisiin, vaan meille riittää listojen sisällön kopioiminen. Tämä asia on kuitenkin hyvä painaa takaraivoon, koska viitteiden käyttäminen vahingossa on helppo tapa ns. ampua itseään jalkaan.

Tehtävä 3.2.1

Mitä allaoleva ohjelma tulostaa?

```
alkuaineet = ["H", "He", "Li", "Be", "B", "C", "N", "O", "F", "Ne"]
alkuaine = alkuaineet[7]
print(alkuaine)
```

- C
- F
- O
- N

Listojen käsittely

Listoja voi muokata useilla erilaisilla funktiolla.

Alkioiden lisääminen

```
# Tyhjä lista luodaan pelkillä hakasulkeilla
alkuaineet = []

# 1) Listoja voi yhdistää "+"-operaattorilla:
alkuaineet = ['C', 'H']
alkuaineet = alkuaineet + ['S', 'O']
# alkuaineet: ['C', 'H', 'S', 'O']

# 2) append-funktio lisää yhden alkion listan loppuun:
alkuaineet.append('Cu')
# alkuaineet: ['C', 'H', 'S', 'O', 'Cu']

# 3) extend-funktio lisää useita alkioita listan loppuun:
alkuaineet.extend(['Ag', 'Au'])
# alkuaineet: ['C', 'H', 'S', 'O', 'Cu', 'Ag', 'Au']

# 4) insert-funktio lisää alkion haluttuun kohtaan:
alkuaineet.insert(0, 'Na')
# alkuaineet: ['Na', 'C', 'H', 'S', 'O', 'Cu', 'Ag', 'Au']
```

Alkioiden poistaminen

```
# remove(x) poistaa alkion, jonka arvo on x
alkuaineet.remove('Au')
# alkuaineet: ['Na', 'C', 'H', 'S', 'O', 'Cu', 'Ag']

# del-komento poistaa alkion, jonka indeksi on n
del alkuaineet[0]
# alkuaineet: ['C', 'H', 'S', 'O', 'Cu', 'Ag']
```

Muita hyödyllisiä listoihin liittyviä toimintoja

```
# Listan lajittelu (aakkosjärjestykseen) sort-funktiolla
alkuaineet.sort()
# alkuaineet: ['Ag', 'C', 'Cu', 'H', 'O', 'S']

# in-avainsanalla voi testata, onko alkio listassa:
if 'C' in alkuaineet:
    print("Hiili on vahvasti mukana")

# index-funktio kertoo tietyn alkion indeksin
print("Vedyn indeksi listassa on: ", alkuaineet.index('H'))
```

tulostaa

```
Hiili on vahvasti mukana
Vedyn indeksi listassa on:  3
```

Listan pienin ja suurin alkio

Listan pienimmän alkion voi etsiä min-funktiolla ja suurimman alkion max-funktiolla:

```
aallonpituudet = [532, 632, 588, 229, 1030, 601]
print(min(aallonpituudet))
print(max(aallonpituudet))
```

tulostaa

```
229
1030
```

Tehtävä 3.3.1.

Täydennä allaoleva koodi ohjeiden mukaisesti

```
yhdisteet = []
# Täydennä niin, että listan sisältö on ['CH4']
yhdisteet.[ ]("CH4")
# Täydennä niin, että lista tyhjenee
yhdisteet.[ ]("CH4")
# Täydennä niin, että listan sisältö on ['NaCl', 'RbCl', 'CsCl']
yhdisteet.[ ](["NaCl", "RbCl", "CsCl"])
# Täydennä niin, että listan sisältö on ['NaCl', 'CsCl']
del yhdisteet[ ]
# Täydennä niin, että listan sisältö on ['NaCl', 'KCl', 'CsCl']
yhdisteet.[ ](1, "KCl")
# Täydennä niin, että tulostaa "CsCl OK"
if "CsCl" in [ ]:
    [ ]("CsCl OK")
# Täydennä niin, että tulostaa 1
[ ](yhdisteet.[ ]("KCl"))
```

Check

Listojen läpikäynti (for, zip)

Listan läpikäyminen for-silmukan avulla

Kun meillä on tietoja tallennettuna listaan, haluamme yleensä myös hyödyntää niitä. Tätä varten tarvitsemme menetelmän listojen läpikäyntiin. Seuraava tapa ei olisi kovin kätevä, jos listassa olisi 1000 alkiota:

```
# Muuntokerroin atm -> bar
ATM_TO_BAR = 1.01325

# Määritellään kolme painetta yksiköissä atm
paineet_atm = [0.56, 1.22, 2.34]
# indeksi:      0      1     2

# Muunnetaan paineet bareiksi brutaalin suoraviivaisesti ja tulostetaan
print(round(paineet_atm[0] * ATM_TO_BAR, 3))
print(round(paineet_atm[1] * ATM_TO_BAR, 3))
print(round(paineet_atm[2] * ATM_TO_BAR, 3))
```

Luonnollisin tapa listojen läpikäyntiin on *for*-silmukka (johon tutustuimme 1. kierroksella). Listojen kanssa pääsemme toden teolla hyödyntämään for-silmukoita.

Esimerkki 1

```
# Muuntokerroin atm -> bar
ATM_TO_BAR = 1.01325

# Määritellään kolme painetta yksiköissä atm:
paineet_atm = [0.56, 1.22, 2.34]
# indeksi:      0      1     2

# Tulostetaan paineet bareina yksi kerrallaan for-silmukan avulla
for paine_atm in paineet_atm:
    paine_bar = paine_atm * ATM_TO_BAR
    print(round(paine_bar, 3))
```

Näin *for*-silmukan avulla voi käydä läpi helposti listan kaikki alkiot, on niitä sitten 3 tai 3000. Listan läpikäyvän for-silmukan yleinen muoto on siis:

```
for ALKIO in LISTA:
    print(ALKIO) # silmukassa voimme tehdä alkiolla mitä haluamme
```

Esimerkki 2

Käydään läpi yhtä listaa ja lisätään samalla alkioita toiseen listaan append-funktiolla (ks. edellinen luku):

```
# Ratkaistaan paine ideaalikaasun tilanyhtälöstä usealle eri tilavuudelle
n = 0.5 # mol
T = 298.15 # K
R = 8.3144598 # J K^-1 mol^-1

# Määritellään kolme tilavuutta yksiköissä m^3
tilavuudet = [0.010, 0.045, 0.105]

# Luodaan tyhjä lista laskettavia paineita varten
paineet = []
# Lasketaan paineet yksiköissä Pa
for tilavuus in tilavuudet:
    paine = n * R * T / tilavuus
    paineet.append(paine)

# Tulostetaan tilavuudet ja paineet yksinkertaisesti ilman pyöristystä
print("tilavuudet:", tilavuudet)
print("paineet:", paineet)
```

tulostaa

```
tilavuudet: [0.01, 0.045, 0.105]
paineet: [123947.80946849998, 27543.957659666663, 11804.553282714285]
```

Esimerkki 3

Tulostetaan tietoja kahdesta yhtä pitkästä listasta.

Tehdään suoraviivainen *for*-silmukka, jossa hyödynnetään silmukkamuuttujaa **i**.

```
tilavuudet = [0.01, 0.045, 0.105]
paineet = [123947.80946849998, 27543.957659666663, 11804.553282714285]
# Hyödynnetään silmukkamuuttujaa i ja len-funktiota.
# Silmukkamuuttuja i saa siis arvot range(len(paineet)), eli [0, 1, 2]
for i in range(len(paineet)):
    print("V = {:.3f} m^3; p = {:.0f} Pa".format(tilavuudet[i], paineet[i]))
```

tulostaa

```
V = 0.010 m^3; p = 123948 Pa
V = 0.045 m^3; p = 27544 Pa
V = 0.105 m^3; p = 11805 Pa
```

Esimerkki 4

Lasketaan arvoja kolmanteen listaan kahden keskenään yhtä pitkän listan avulla:

```
ainemaarat = [0.4, 0.6, 0.8]    # mol
tilavuudet = [0.25, 0.25, 0.25] # l
konsentraatiot = []             # Lasketaan nämä (mol/l)
for i in range(len(ainemaarat)):
    c = ainemaarat[i] / tilavuudet[i]
    konsentraatiot.append(c)
print(konsentraatiot)
```

tulostaa

```
[1.6, 2.4, 3.2]
```

Ylläolevilla for-silmukoilla kurssin tehtävistä selviää täysin hyväksyttävästi. Seuraavassa kappaleessa on pari vaihtoehtoista tapaa hoitaa sama asia käyttäen Pythonin sisäänrakennettuja hienouksia.

zip-funktio

Kätevä tapa hoitaa esimerkin 4 tilanne on yhdistää kaksi listaa *zip*-funktion avulla (engl. *zip* = vetoketju):

```
ainemaarat = [0.4, 0.6, 0.8]    # mol
tilavuudet = [0.25, 0.25, 0.25] # l
konsentraatiot = []             # Lasketaan nämä (mol/l)
for n, V in zip(ainemaarat, tilavuudet):
    # silmukkamuuttuja n saa arvot listasta ainemaarat
    # silmukkamuuttuja V saa arvot istasta tilavuudet
    c = n / V
    konsentraatiot.append(c)
print(konsentraatiot)
```

Lopputulos olisi sama kuin edellä. Katsotaan tarkemmin, mitä zip-funktio palauttaa (muuntamalla funktion tulos listaksi):

```
print(list(zip(ainemaarat, tilavuudet)))
```

tulostaa

```
[(0.4, 0.25), (0.6, 0.25), (0.8, 0.25)]
```

Eli kolmen alkion lista, jossa jokainen alkio on kahden alkion monikko (eli lista, jota ei voi muokata ks. seuraava luku).

zip-funktio on erittäin kätevä tapa yhdistää listoja *for*-silmukkaa varten.

enumerate-funktio.

enumerate-funktio on myös usein avuksi listojen läpikäymisessä. Se palauttaa kullekin listan alkiolle sekä sen indeksin että alkion arvon:

```
alkuaineet = ["H", "He", "Li", "Be"]
for indeksi, alkuaine in enumerate(alkuaineet):
    print("Z: {:d}; alkuaine: {:s}".format(indeksi + 1, alkuaine))
```

tulostaa

```
Z: 1; alkuaine: H
Z: 2; alkuaine: He
Z: 3; alkuaine: Li
Z: 4; alkuaine: Be
```

Saman silmukan voisi toteuttaa myös silmukkamuuttujan avulla:

```
alkuaineet = ["H", "He", "Li", "Be"]
for i in range(len(alkuaineet)):
    print("Z: {:d}; alkuaine: {:s}".format(i + 1, alkuaineet[i]))
```

On lähinnä makuasia, kumpaa tapaa käyttää. *enumerate*-funktio voi auttaa tekemään koodista luettavampaa kuin silmukkamuuttujan käyttö.

Katsotaan vielä tarkemmin, mitä enumerate-funktio oikeastaan palauttaa (muunnetaan enumerate-funktion tulos listaksi):

```
alkuaineet = ["H", "He", "Li", "Be"]
print(list(enumerate(alkuaineet)))
```

tulostaa

```
[(0, 'H'), (1, 'He'), (2, 'Li'), (3, 'Be')]
```

Eli kukin *alkuaineet*-listan alkio on saanut parikseen indeksin. Huomaa, että listassa on neljä alkiota ja jokainen alkio on kahden alkion *monikko* (lista, jota ei voi muokata ks. seuraava luku).

Lisätietoa: List comprehension -mekanismi

(tämä kappale on syventävää tietoa, ei välttämätöntä kurssin läpäisemiseksi). Kuten ylläolevat esimerkit näyttää, *for*-silmukka on selkeä työkalu listojen läpikäymiseen ja uusien listojen luomiseen. Mainitsen tässä syventävänä tietona myös List comprehension -mekanismin, jolla Pythonissa on erityisen kätevää luoda uusia listoja olemassaolevien listojen avulla.

List comprehension-lauseke kirjoitetaan hakasulkeiden väliin:

```
uusi_lista = [ uuden_listan_alkion_lauseke for vanha_alkio in vanha_lista ]
```

Esimerkki:

```
tilavuudet_m3 = [0.010, 0.045, 0.105]
tilavuudet_litroina = [ tilavuus_m3 * 1000 for tilavuus_m3 in tilavuudet_m3 ]
print(tilavuudet_m3)
print(tilavuudet_litroina)
```

tulostaa

```
[0.01, 0.045, 0.105]
[10.0, 45.0, 105.0]
```

Toinen esimerkki:

```
# Ratkaistaan paine ideaalikaasun tilanyhtälöstä usealle eri tilavuudelle
n = 0.5 # mol
T = 298.15 # K
R = 8.3144598 # J K^-1 mol^-1

# Määritellään kolme tilavuutta yksiköissä m^3
tilavuudet = [0.010, 0.045, 0.105]

# Käytetään for-silmukan sijasta "List comprehension"-mekanismia
paineet = [ n * R * T / tilavuus for tilavuus in tilavuudet ]

# Tulostetaan tilavuudet ja paineet yksinkertaisesti ilman pyöristystä
print("tilavuudet:", tilavuudet)
print("paineet:", paineet)
```

Tehtävä 3.4.1.

Täydennä koodi vetämällä sanat oikeisiin laatikoihin

```
# Määritellään kaksi listaa
ainemaarat = [0.15, 0.25, 0.45, 0.53] # mol
tilavuudet = [0.5, 1.0, 1.5, 2.0] # litraa

# Lasketaan konsentraatiot ensin suoraviivaisesti silmukkamuuttujan avulla
konsentraatiot = []
for i in                    (                    (ainemaarat)):
                    .append(                [i] /                [i])
# konsentraatiot: [0.3, 0.25, 0.3, 0.265] mol/l

# Lasketaan konsentraatiot zip-funktion avulla
konsentraatiot = []
for ainemaara, tilavuus in                    (ainemaarat, tilavuudet):
                    .append(                /                )
# konsentraatiot: [0.3, 0.25, 0.3, 0.265] mol/l
```

tilavuudet
konsentraatiot
len
ainemaarat
tilavuus
range
konsentraatiot
zip
ainemaara

Check

Monikot

Emme käytä paljon aikaa monikkojen käsittelyyn, sillä tämän kurssin puitteissa meille riittää tieto, että monikko on muuten kuin lista, mutta sitä ei voi muokata:

```
# Monikko määritellään siis tavallisilla sulkeilla
jalokaasut = ("He", "Ne", "Ar", "Kr", "Xe", "Rn")
# indeksi      0     1     2     3     4     5
# Monikon alkioihin viitataan hakasulkeilla
print(jalokaasut[2]) # Tulostaa Ar
# Seuraavat komennot ovat virheellisiä monikkojen tapauksessa
jalokaasut[2] = "H"
    # TypeError: 'tuple' object does not support item assignment
del jalokaasut[0]
    # TypeError: 'tuple' object doesn't support item deletion
```

Törmäämme monikkoihin lähinnä tilanteissa, joissa Python käyttää sisäisesti monikkoa tyyppinä. Esimerkiksi zip-funktio (ks. edellinen luku):

```
alkuaineet = ['H', 'C', 'O']
atomipainot = [1.008, 12.011, 15.999]
alkuaine_monikot = zip(alkuaineet, atomipainot)
print(list(alkuaine_monikot))
```

tulostaa

```
[('H', 1.008), ('C', 12.011), ('O', 15.999)]
```

Eli lista, jossa on kolme alkiota, joista jokainen on kahden alkion monikko. Käytännön esimerkki zip-funktion hyödyntämisestä tässä tapauksessa:

```
alkuaineet = ['H', 'C', 'O']
atomipainot = [1.008, 12.011, 15.999]
for alkuaine, atomipaino in zip(alkuaineet, atomipainot):
    print("Alkuaineen {:s} atomipaino on {:.3f} g/mol".format(alkuaine, atomipaino))
```

tulostaa

```
Alkuaineen H atomipaino on 1.008 g/mol
Alkuaineen C atomipaino on 12.011 g/mol
Alkuaineen O atomipaino on 15.999 g/mol
```

Jälleen kerran saman asian voisi hoitaa suoraviivaisella *for*-silmukalla ja silmukkamuuttujalla, mutta *zip*-funktio on tavallaan "luonnollisempi" tapa hoitaa asia Pythonissa.

Tehtävä 3.5.1.

Mitä allaoleva koodi tulostaa?

```
hapot = ("HCl", "H2SO4", "HNO3", "CH3COOH")
print(hapot[3])
```

- ('HCl', 'H2SO4', 'HNO3')
- HNO3
- CH3COOH
- Virheellinen indeksi

Sanakirjat

Sanakirjassa alkiot määritellään **avain:arvo** -pareina:

```
atomipainot = {"H":  1.008, "C": 12.011, "O": 15.999}
```

Tämän määrittelyn jälkeen avainta vastaavan arvon voi noutaa näin:

```
print("Hiilen atomipaino on", atomipainot["C"])
```

tulostaa

```
Hiilen atomipaino on 12.011
```

Määrittelyssä käytetään siis kaarisulkeita, mutta kun arvoihin viitataan avaimella, käytetään hakasulkeita.

Tyhjän sanakirjan luominen

```
uusi_sanakirja = {}
```

Arvojen lisääminen sanakirjaan

Arvojen lisääminen sanakirjaan on helppoa: annetaan vain uusi avain ja arvo:

```
# Luodaan tyhjä sanakirja ja lisätään kolme avain:arvo -paria
atomipainot = {}
atomipainot["H"] = 1.008
atomipainot["C"] = 12.011
atomipainot["O"] = 15.999
print(atomipainot)
```

Tulostaa

```
{'H': 1.008, 'C': 12.011, 'O': 15.999}
```

Voit siis myös määritellä sanakirjan ensin tiettyjen avain:arvo parien kanssa ja lisätä siihen myöhemmin lisää pareja:

```
atomipainot = {"H":  1.008, "C": 12.011, "O": 15.999}
atomipainot["P"] = 30.973
print(atomipainot)
```

tulostaa

```
{'H': 1.008, 'C': 12.011, 'O': 15.999, 'P': 30.973}
```

Python tulostaa sanakirjojen avaimet aina yksinkertaisia lainausmerkkejä käyttäen.

in-avainsana toimii samaan tapaan kuin listojen kanssa:

```
# in-avainsanalla voi testata, onko avain sanakirjassa:
atomipainot = {"H":  1.008, "C": 12.011, "O": 15.999}
if "C" in atomipainot:
    print("Hiilen atomipaino on", atomipainot["C"])
```

tulostaa

```
Hiilen atomipaino on 12.011
```

Arvojen poistaminen sanakirjasta

Arvojen poistaminen sanakirjasta onnistuu *del*-avainsanalla:

```
atomipainot = {"H":  1.008, "C": 12.011, "O": 15.999}
del atomipainot["C"]
print(atomipainot)
```

tulostaa

```
{'H': 1.008, 'O': 15.999}
```

Sanakirjan läpikäyminen

Sanakirjan *items*()-funktio antaa arvot läpikäyntiä varten:

```
atomipainot = {"H":  1.008, "C": 12.011, "O": 15.999}
for alkuaine, atomipaino in atomipainot.items():
    print("Alkuaineen {:s} atomipaino on {:.3f} g/mol".format(alkuaine, atomipaino))
```

tulostaa

```
Alkuaineen H atomipaino on 1.008 g/mol
Alkuaineen C atomipaino on 12.011 g/mol
Alkuaineen O atomipaino on 15.999 g/mol
```

Yleinen muoto siis

```
for AVAIN, ARVO in SANAKIRJA.items():
    print(AVAIN, ARVO) # Silmukassa voimme käyttää avaimia ja arvoja kuten haluamme.
```

Sanakirjan lajitteleminen

sorted()-funktiolla voi tulostaa avaimet aakkosjärjestyksessä tai arvot järjestyksessä (*values*-funktio):

```
atomipainot = {"P": 30.973, "C": 12.011, "O": 15.999}
print(sorted(atomipainot))
print(sorted(atomipainot.values()))
```

tulostaa

```
['C', 'O', 'P']
[12.011, 15.999, 30.973]
```

Huomaa kuitenkin, että alkuperäisen sanakirjan (*atomipainot*) järjestys ei muutu, vaikka kutsuisit *sorted*-funktiota. Vain funktion paluarvoja palaava lista muuttuu.

Huom! Ennen Pythonin versiota 3.6, sanakirjan avain:arvo parit olivat satunnaisessa järjestyksessä. Versiosta 3.6 eteenpäin ne ovat siinä järjestyksessä, missä ne on lisätty sanakirjaan. Tätä ei ole kuitenkaan vielä vahvistettu standardissa. Jos tarvitset sanakirjan, joka pysyy aina järjestyksessä, katso OrderedDict.

Listat sanakirjojen sisällä

Sanakirjan arvot voivat olla vaikka listoja:

```
# Sanakirjan arvot voivat olla vaikka listoja:
yhdisteet = {"C2H6": ["C",  2,  "H", 6],
             "NaCl": ["Na", 1, "Cl", 1]
          # indeksi:   0    1    2   3
            }
print(yhdisteet["C2H6"])
print("Yhdisteessa C2H6 on", yhdisteet["C2H6"][3], "vetyatomia")
```

tulostaa

```
['C', 2, 'H', 6]
Yhdisteessa C2H6 on 6 vetyatomia
```

Tehtävä 3.6.1

Täydennä allaoleva koodi ohjeiden mukaisesti

```
# Määrittele sanakirja moolimassat, jossa on avaimet XeF2, XeF4 ja XeF6
moolimassat = {"          ": 169.29, "          ": 207.28, "          ": 245.3}

# Tulosta sanakirjan sisältö
for yhdiste, moolimassa in moolimassat.          :
    print("Yhdisteen {:s} moolimassa on {:.2f} g/mol".format(          ,          ))

# Poista XeF4 sanakirjasta
          moolimassat["XeF4"]

# Muuta yhdisteen XeF6 moolimassaksi 245.28
moolimassat[          ] = 245.28
```

Check

Sisäkkäiset tietorakenteet

Pythonin erilaisia tietorakenteita voi käyttää myös sisäkkäin. Jos listoja sisältävä lista kuulostaa erikoiselta, suosittelen vahvasti kokeilemaan allaolevia esimerkkejä Spyderissä ja kokeilemaan niiden muokkausta.

Sisäkkäiset listat

Listan alkio voi olla myös toinen lista:

```
# Määritellään lista, jossa kaksi alkiota. Kukin alkio on kolmen alkion lista.
lista = [[10, 20, 30], [1, 2, 3]]
print(lista[0][0])
print(lista[1][2])
```

tulostaa

```
10
3
```

Eli merkinnässä lista[1][0] ensimmäinen indeksi [1] viittaa ulomman listan toiseen alkioon [1, 2, 3] (indeksointi nollasta!). Toinen indeksi [2] viittaa sisemmän listan kolmanteen alkioon (indeksointi nollasta!).

Otetaan käytännöllisempi esimerkki. Kuvataan kemiallista yhdistettä listalla:

- Listan jokainen alkio on toinen lista
- Tämä lista sisältää alkuaineen symbolin ja sen määrän yhdisteessä

```
yhdiste_1 = [['C', 2], ['H', 6]]
yhdiste_2 = [['Ca', 1], ['Cl', 2]]

# Lisätään nyt kaikki yhdisteet yhteen listaan ja tulostetaan
yhdisteet = [yhdiste_1, yhdiste_2]
print(yhdisteet)
```

tulostaa

```
[[['C', 2], ['H', 6]], [['Ca', 1], ['Cl', 2]]]
```

Laajempi esimerkki

```
# Käydään läpi yhdisteet, tulostetaan ne ja etsitään hiilivedyt
yhdisteet = [['C', 2], ['H', 6]], [['Ca', 1], ['Cl', 2]]
for yhdiste in yhdisteet:
    # "yhdiste" on nyt esim. [['C', 2], ['H', 6]]
    # Alustetaan muuttujat ennen sisempää for-silmukkaa
    yhdisteen_kaava = ""
    on_hiili = on_vety = False
    # Käydään läpi kaikki yhdisteen alkuaineet
    for alkuaine in yhdiste:
        # "alkuaine" on nyt esim. ['C', 2]
        # Tulostetaan määrä vain, jos se on > 1
        if alkuaine[1] > 1:
            maara = str(alkuaine[1])
        else:
            maara = ""
        yhdisteen_kaava += alkuaine[0] + maara

        # Tarkistetaan, onko alkuaine hiili tai vety
        if alkuaine[0] == 'C':
            on_hiili = True
        elif alkuaine[0] == 'H':
            on_vety = True

    # Tulostetaan yhdisteen molekyylikaava
    if len(yhdiste) == 2 and on_hiili and on_vety:
        hiilivety_str = "on hiilivety"
    else:
        hiilivety_str = ""
    print("Yhdiste:", yhdisteen_kaava, hiilivety_str)
```

tulostaa

```
Yhdiste: C2H6 on hiilivety
Yhdiste: CaCl2
```

Matriisit listojen avulla

Sisäkkäisillä listoilla voisi periaatteessa kuvata matriiseja:

```
matriisi = [[2, 4],
            [5, 6]]
# Tulostetaan 1. rivin 2. alkio (indeksointi nollasta!)
print(matriisi[0][1]) # tulostaa 4
```

Käytännössä matriisilaskentaan käytetään kuitenkin NumPy-kirjaston array-tyyppiä, johon tutustutaan kierroksesta 4 lähtien.

Listat sanakirjojen sisällä

Sanakirjan arvot voivat olla listoja:

```
# Sanakirjan arvot voivat olla vaikka listoja:
yhdisteet = {"C2H6": ["C",  2,  "H", 6],
             "NaCl": ["Na", 1, "Cl", 1]
          # indeksi:   0    1    2   3
            }
print(yhdisteet["C2H6"])
print("Yhdisteessa C2H6 on", yhdisteet["C2H6"][3], "vetyatomia")
```

tulostaa

```
['C', 2, 'H', 6]
Yhdisteessa C2H6 on 6 vetyatomia
```

Sisäkkäiset sanakirjat

Sanakirjoja voi laittaa sisäkkäin:

```
tietokanta = {
              "C2H6": {"moolimassa": 30.07, "tiheys": 1.36},
              "NaCl": {"moolimassa": 58.44, "tiheys": 2.16}
             }
print("Etaanin tiheys on:", tietokanta["C2H6"]["tiheys"], "g/cm^3")
print("Ruokasuolan moolimassa on:", tietokanta["NaCl"]["moolimassa"], "g/mol")
```

tulostaa

```
Etaanin tiheys on: 1.36 g/cm^3
Ruokasuolan moolimassa on: 58.44 g/mol
```

Tehtävä 3.7.1

Mitä allaoleva koodi tulostaa?

```
G12 = [["H", "Li", "Na", "K", "Rb", "Cs"],
       ["Be", "Mg", "Ca", "Sr", "Ba"]]
print(G12[1])
```

- Mg
- Li
- ["H", "Li", "Na", "K", "Rb", "Cs"]
- ['Be', 'Mg', 'Ca', 'Sr', 'Ba']

Merkkijonojen käsittely listoina

Merkkijonot ovat läheistä sukua listoille. Merkkijonon voi muuntaa suoraan listaksi:

```
merkkijono_listana = list('Sana')
print("Merkkijono listana:", merkkijono_listana)
```

tulostaa

```
Merkkijono listana: ['S', 'a', 'n', 'a']
```

Merkkijonon voi siis itsessään ajatella olevan "lista merkkejä". Näin ollen myös merkkijonoja voi indeksoida ja siivuttaa:

```
teksti =  "Kemisti"
# indeksi: 0123456
print(teksti[0])
print(teksti[0:4])
```

tulostaa

```
K
Kemi
```

Merkkijonofunktiot

Pythonin dokumentaatiossa listataan useita merkkijonojen käsittelyyn tarkoitettuja funktioita. Tutustutaan tässä erikseen muutamaan funktioon, joilla voi tutkia merkkijonon sisältöä.

funktiolla *str.isdigit* voi etsiä numeroita:

```
# Käydään katuosoite läpi merkki kerrallaan ja poimitaan numerot
katuosoite = "Kemistintie 1"
numerot = ""
for merkki in katuosoite:
    if merkki.isdigit():
        numerot = numerot + merkki
print("Talon numero on", numerot)
```

tulostaa

```
Talon numero on 1
```

Funktiolla *str.isalpha* voi etsiä kirjaimia:

```
# Käydään postinumero läpi merkki kerrallaan ja poimitaan kirjaimet
postinumero = "02150 ESPOO"
kirjaimet = ""
for merkki in postinumero:
    if merkki.isalpha():
        kirjaimet = kirjaimet + merkki

print("Postitoimipaikka on", kirjaimet)
```

tulostaa

```
Postitoimipaikka on ESPOO
```

Funktioilla *str.isupper* ja *str.islower* voi tutkia, onko merkki iso vain pieni kirjain. *str.isspace* kertoo, onko merkki "whitespace", eli esimerkiksi välilyönti, tabulaattori tai rivinvaihto:

```
# Kerätään alkuainesymbolit listaan
teksti = "Sc Ti V Cr Mn Co Fe Ni Cu Zn"
alkuaineet = []
apujono = ""
# Käydään teksti läpi kirjain kerrallaan
for merkki in teksti:
    # Alkuaineen symboli alkaa aina isolla kirjaimella
    if merkki.isupper():
        # Iso kirjain talteen
        apujono = merkki
    elif merkki.islower():
        # Lisätään pieni kirjain ison alkukirjaimen perään
        apujono = apujono + merkki
    elif merkki.isspace():
        # Välilyönti erottaa symbolit, eli apujono sisältää nyt alkuainesymbolin
        # Symboli on joko (a) iso kirjain + pieni kirjain tai (b) vain iso kirjain
        alkuaineet.append(apujono)
        apujono = ""
print(alkuaineet)
```

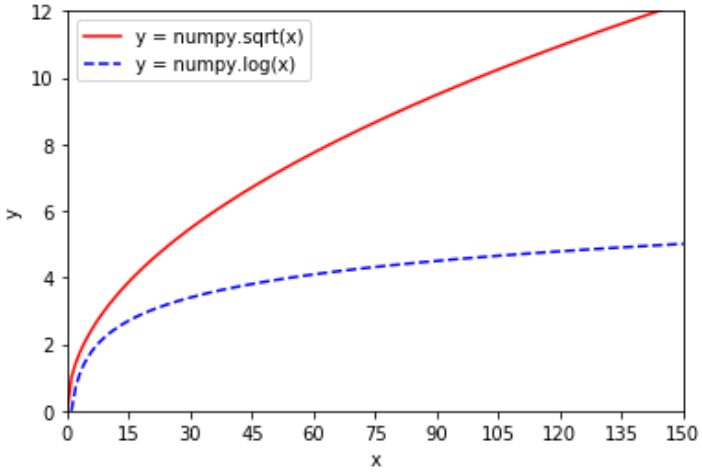
tulostaa

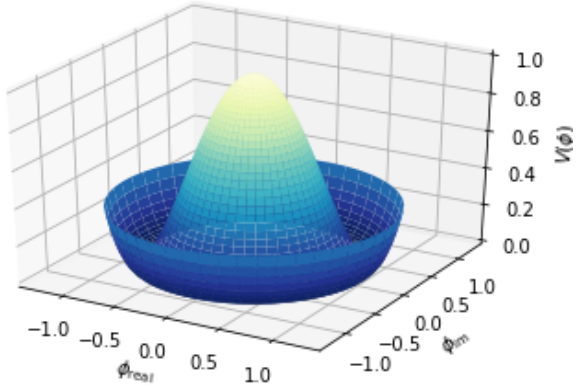
```
['Sc', 'Ti', 'V', 'Cr', 'Mn', 'Co', 'Fe', 'Ni', 'Cu']
```

Kierros 4

Neljännellä kierroksella otamme käyttöön **numpy**-kirjaston, joka sisältää luonnontieteissä ja tekniikassa erityisen hyödyllisen tietorakenteen, eli taulukon (**array**). Taulukoiden avulla voimme helposti ja tehokkaasti kuvata vektoreita, matriiseja ja mitä tahansa N-ulotteisia datajoukkoja. Tutustumme myös tietotyyppeihin, joilla on helppo käsitellä polynomeja.

Lisäksi otamme käyttöön **matplotlib**-kirjaston, jolla pystymme visualisoimaan ja analysoimaan dataa. Upeita kuvaajia luvassa!





Tehtävä 4.0.1.

Tästä alkaa kierros 4. Sitä ennen kierroksen 3 pikakertaus.

---

1 / 8

Onko tietorakenne "pH" lista, sanakirja, monikko vai joukko: pH = [1.5, 2.5, 4.5, 5.5, 6.5]

Your answer Check

Listan pituuden palauttava funktio? (nimi ilman sulkeita

Your answer Chec

NumPy-kirjasto

NumPy-kirjasto sisältää numeerisen laskennan kannalta keskeisiä työkaluja:

- *ndarray* (tai *array*) -tietorakenne eli *taulukko*. Taulukoilla voidaan kuvata esimerkiksi vektoreita, matriiseja ja mitä tahansa moniulotteisia datajoukkoja. Numpy-taulukot soveltuvat erittäin hyvin esimerkiksi mittausdatan käsittelyyn ja ne mahdollistavat numeerisen laskennan aivan eri tasolla kuin tavalliset listat.
- Lukuisia funktioita taulukoiden käsittelyyn:
  - Matemaattiset perusfunktiot (sin, cos, exp)
  - Lineaarialgebra (matriisit ja vektorit)
  - Tilastolliset funktiot, polynomit, datan sovitus, jne.
- Numpy-taulukot ja niiden käsittelyyn liittyvät funktiot on toteutettu mahdollisimman tehokkaasti ja ne soveltuvat hyvinkin raskaaseen laskentaan

Jos haluat tehdä numeerista laskentaa Pythonilla, käytä NumPy-kirjastoa. NumPy + SciPy + Matplotlib –yhdistelmällä voi korvata monessa asiassa Matlabin.

Matplotlib-kirjastoa käytämme jo tällä kierroksella kuvaajien tekemiseen. SciPy:stä opimme lisää kurssin viimeisellä kierroksella.

Jos haluat oppia NumPystä enemmän kuin tämän kurssin puitteissa on mahdolista, suosittelen Nicolas P. Rougierin materiaaleja:

- NumPy-tutoriaali: http://www.labri.fr/perso/nrougier/teaching/numpy/numpy.html
- From Python to NumPy-kirja: http://www.labri.fr/perso/nrougier/from-python-to-numpy/index.html

NumPy-taulukot (*array*)

NumPy-taulukoita ja muita NumPyn ominaisuuksia käytettäessä ohjelmaan pitää aina tuoda *numpy*-moduuli **import**-käskyllä. Tällä kurssilla moduuli tuodaan aina seuraavalla käskyllä, jolloin NumPyn funktioita voi kutsua lyhennettä *np* käyttäen:

```
import numpy as np
```

Taulukoiden luominen ja alkioihin viittaaminen

Taulukoita (*array*) voi luoda suoraviivaisesti *numpy.array*-funktion avulla.

Yksiulotteiset taulukot (vektorit)

Luodaan yksiulotteinen neljan alkion taulukko (eli **vektori**)

```
vektori = np.array([10, 20, 30, 40])
# indeksi:          0   1   2   3
```

Taulukon *ulottuvuus* (engl. *dimension*) tarkoittaa, montako indeksiä tarvitaan yhden alkion osoittamiseen. Yksiulotteisen esimerkkivektorin tapauksessa tarvitsemme vain yhden indeksin, joka saa arvot 0-3.

Taulukon alkiohin viittaaminen toimii kuten listojen kanssa (muista, että ensimmainen indeksi on 0!):

```
eka = vektori[0] # 10
toka = vektori[1] # 20
vika_1 = vektori[3] # 40
vika_2 = vektori[-1] # 40
```

Kaksiulotteiset taulukot (matriisit)

Luodaan kaksiulotteinen kahden rivin ja kolmen sarakkeen taulukko (eli **matriisi**):

```
matriisi = np.array([[10, 20, 30],
                     [40, 50, 60]])
```

Erona tavallisiin listoihin kaksiulotteisten NumPy-taulukkojen alkioihin voi viitata käytännöllisellä *taulukko[rivi, sarake]* -merkinnällä:

```
ekan_rivin_eka_sarake = M[0, 0] # 10
tokan_rivin_toka_sarake = M[1, 1] # 50
tokan_rivin_vika_sarake = M[1, -1] # 60
```

Myös tavallisista listoista tuttu merkintä *taulukko[rivi][sarake]* toimii, mutta se on kömpelömpi käyttää.

Kolmiulotteiset taulukot

Luodaan viimeisenä esimerkkinä kolmiulotteinen 2 x 2 x 3 taulukko:

```
T = np.array([[[1, 2, 3],
               [4, 5, 6]],

              [[10, 20, 30],
               [40, 50, 60]
              ]])
alkio = T[1, 1, 0] # 40
```

Taulukon ulottuvuuksien tarkastelu (ndim, shape, len)

Minkä tahansa NumPy-taulukon ulottuvuuksien määrän voi selvittää *ndim*-funktiolla:

```
matriisi = np.array([[10, 20, 30],
                     [40, 50, 60]])
matriisin_ulottuvuus = np.ndim(matriisi) # palauttaa kokonaisluvun 2
```

shape-funktio palauttaa taulukon ulottuvuuksien dimensiot

```
# Käytetään yllä määriteltyä taulukkoa T
T_dimensiot = np.shape(T) # palauttaa monikon (2, 2, 3)
```

Tuttu *len*-funktio palauttaa taulukon halutun ulottuvuuden pituuden:

```
# Käytetään yllä määriteltyjä taulukoita vektori ja T
vektorin_pituus = len(vektori) # palauttaa kokonaisluvun 4
T_kolmas_ulottuvuus_pituus = len(T[0, 0]) # palauttaa kokonaisluvun 3
```

Listojen muuntaminen taulukoiksi

numpy.array-funktio muuntaa siis tavallisen listan NumPy-taulukoksi. Seuraavassa esimerkissä vektori *v1* luodaan kokonaislukujen listasta [1, 2, 3, 4, 5]:

```
v1 = np.array([1, 2, 3, 4, 5])
```

Saman muunnoksen voi tehdä myös olemassaoleville listoille (tai monikoille):

```
lista = [1.1, 2.2, 3.3, 4.4]
v2 = np.array(lista)
# nyt v2 on array([ 1.1,  2.2,  3.3,  4.4])
M1 = np.array([lista, lista])
# Nyt M1 on kaksiulotteinen taulukko
# array([[ 1.1,  2.2,  3.3,  4.4],
#        [ 1.1,  2.2,  3.3,  4.4]])
```

Nollilla alustetun taulukon luominen

Tyhjän taulukon voi luoda *numpy.zeros*-funktiolla:

```
vektori = np.zeros(8) # Kahdeksan alkiota pitkä vektori täynnä nollia
matriisi = np.zeros((9, 9)) # 9x9 matriisi täynnä nollia. zeros-funktion parametri on tässä monikko (9, 9).
```

Moniulotteisen taulukon tapauksessa *numpy.zeros*-funktion parametrin *shape* tulee olla monikko tai lista.

Taulukoiden luominen *arange*- ja *linspace*-funktioilla

Kokonaislukuja sisältäviä taulukoita on kätevä luoda *range*-funktiota vastaavalla *numpy.arange*-funktiolla:

```
v3 = np.arange(1, 10)
# np.arange(alku, loppu)
# Nyt v3 on array([1, 2, 3, 4, 5, 6, 7, 8, 9])
# Huomaa, että viimenen alkio on loppu-1 (kuten range-funktiolla)
```

Myös muoto *np.arange(alku, loppu, askel)* on käytössä:

```
v4 = np.arange(2, 11, 2)
# Nyt v4 on array([ 2,  4,  6,  8, 10])
```

Liukulukujen kanssa käytetään *numpy.linspace*-funktiota, jota kutsutaan *numpy.linspace(start, stop, num)*. Parametri *start* on ensimmäinen luku, *stop* on viimeinen luku ja *num* on lukujen määrä välillä *start..stop*. **Huomaa**, etta oletuksena *stop*-luku tulee mukaan, toisin kuin *arange*-funktiossa!

```
v5 = np.linspace(0.1, 1, 10)
# v5 on array([ 0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
v6 = np.linspace(-0.2, 0.2, 6)
# v6 on array([-0.2 , -0.12, -0.04,  0.04,  0.12,  0.2 ])
```

Jos tarvitset tasavälisiä lukuja logaritmisella asteikolla, voit käyttää funktiota *numpy.logspace*.

Tehtävä 4.2.1

Määritellään NumPy-taulukko:

tiheydet = np.array([0.81, 0.75, 0.88, 0.94, 1.08, 1.24, 2.34, 1.00])

Mitä on tiheydet[3]?

- 3
- 0.88
- 0.94
- 1.08

NumPy-taulukoiden siivuttaminen

Luodaan ensin uusi yksiulotteinen taulukko (vektori):

```
import numpy as np

v = np.arange(100, 1100, 100)
# v on array([ 100,  200,  300,  400,  500,  600,  700,  800,  900, 1000])
# indeksi:     0     1     2     3     4     5     6     7     8    9
```

NumPy-taulukoiden siivuttaminen toimii samaan tapaan kuin listojen siivuttaminen. Merkinnässä [*start:stop:askel*], *stop*-alkio ei siis kuulu enää siivuun. *askel*-osuus ei ole pakollinen. Siivutetaan yllä luotu vektori v:

```
siivu1 = v[0:4]
# siivu1 on array([100, 200, 300, 400])
siivu2 = v[5:7]
# siivu2 on array([600, 700])
siivu3 = v[0:7:2]
# siivu3 on array([100, 300, 500, 700])
```

NumPy-tarjoaa myös erittain käytannöllisen **:** -indeksoinnin Matlabin tapaan. **:** -indeksi tarkoittaa kyseisen ulottuvuuden kaikkia indeksejä:

```
# Määritellään 3 x 4 matriisi M
M = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
# Matriisin M kaikkien rivien kolmas sarake
A = M[:, 2] # array([ 3,  7, 11])
# Matriisin M toisen rivin kaikki sarakkeet
B1 = M[1, :] # array([5, 6, 7, 8])
# Rivin siivuttamisen voi tehdä myös merkinnällä, jossa sarakkeet jätetään pois
B2 = M[1]    # array([5, 6, 7, 8])
```

Laskuoperaatiot NumPy-taulukoilla

Luodaan ensin uusi taulukko (yksiulotteinen vektori):

```
import numpy as np

v1 = np.arange(100, 1001, 100)
# v1 on array([ 100,  200,  300,  400,  500,  600,  700,  800,  900, 1000])
```

NumPy-taulukoiden ja yksittäisten lukuarvojen laskuoperaatiot onnistuvat suoraviivaisesti. NumPy suorittaa laskuoperaation jokaiselle alkiolle:

```
v2 = v1 + 1    # array([ 101,  201,  301,  401,  501,  601,  701,  801,  901, 1001])
v3 = v1 * 2    # array([ 200,  400,  600,  800, 1000, 1200, 1400, 1600, 1800, 2000])
v4 = v1 / 100  # array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
v42 = 100 / v1 # array([ 1.,  0.5,  0.33333333, 0.25,  0.2, 0.16666667, 0.14285714, 0.125, 0.11111111, 0.1])
```

NumPy-taulukoita voi myos lisätä, kertoa ja jakaa keskenään. Operaatiot tehdaan alkioittain, joten taulukoiden tulee olla samankokoisia!

```
v5 = v2 + v2  # array([ 202,  402,  602,  802, 1002, 1202, 1402, 1602, 1802, 2002])
v6 = v4 * v4  # array([   1.,    4.,    9.,   16.,   25.,   36.,   49.,   64.,   81.,  100.])
v7 = v3 / v1  # array([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
```

Seuraavassa esimerkissä matriisin *M* sarakkeiden määrä (4) täsmää vektorin *a* pituuden (4) kanssa. Jokainen rivivektori kerrotaan alkioittain vektorilla *a*:

```
M = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
a = np.array([1, 2, 3, 4])
tulo = M * a
# Ensimmäinen rivi on siis [1*1, 2*2, 3*3, 4*4]
# array([[ 1,  4,  9, 16],
#        [ 5, 12, 21, 32],
#        [ 9, 20, 33, 48]])
```

Huomaa, että esimerkin kertolasku *M * a* on aivan eri asia kuin oikea matriisitulo! Siitä lisää seuraavassa luvussa.

Laskutoimitukset ilman silmukoita

Tarkastellaan esimerkin avulla, kuinka NumPy-taulukoiden kanssa toimitaan listoihin verrattuna. Aiemmin olemme oppineet, kuinka kahdesta listasta lasketaan tuloksia kolmanteen:

```
massat      = [2.2,    4.1,   5.6,    1.2,     6.7]
moolimassat = [18.015, 58.44, 74.55, 81.408, 144.645]
ainemaarat = []
for massa, moolimassa in zip(massat, moolimassat):
    ainemaarat.append(massa / moolimassa)
```

NumPyllä for-silmukkaa ei tarvita:

```
import numpy as np
massat      = np.array([2.2,    4.1,   5.6,    1.2,     6.7])
moolimassat = np.array([18.015, 58.44, 74.55, 81.408, 144.645])
ainemaarat_np = massat / moolimassat
```

NumPy jakaa siis taulukon *massat* jokaisen alkion taulukon *moolimassat* vastaavalla alkiolla ja lopputulos on uusi taulukko *ainemaarat*.

Molemmissa tapauksissa lasketut ainemäärät ovat samat. Mutta Numpyn tapa on merkittävästi nopeampi, varsinkin kun kyseessä on vähänkin isompi datamäärä. NumPyn lähestymistapaa kutsutaan *vektoroinniksi*.

Laajempi esimerkki

Toteutetaan funktio *ainemaara* NumPy-taulukoiden avulla:

```
import numpy as np

def ainemaara(m, M):
    # m, M: massa (g) ja moolimassa (g/mol)
    # Kukin parametri voi olla joko NumPy-taulukko tai yksittäinen luku
    # Oletetaan, että kaikki parametrit ovat kelvollisia lukuarvoja
    # Paluuarvo: ainemäärä(t)
    # Jos yksikin parametri on NumPy-taulukko, paluuarvo on NumPy-taulukko
    return m / M

# Luodaan kolmen alkiota sisältävät taulukot massoista ja moolimassoista
massat = np.array([3.2, 0.5, 2.2])
moolimassat = np.array([58.44, 42.394, 120.921])
                      # NaCl   LiCl    RbCl
# Lasketaan ja tulostetaan ainemäärät. Funktio palauttaa taulukon.
n1 = ainemaara(massat, moolimassat)
print(n1)

# Funktio toimii myös yksittäisillä lukuarvoilla
# Tässä tapauksessa funktio palauttaa yksittäisen liukuluvun
n2 = ainemaara(3.2, 58.44)
print(n2)

# Funktio toimii myös jos toinen parametri on yksittäinen luku ja toinen taulukko
# Tässä tapauksessa funktio palauttaa taulukon
n3 = ainemaara(3.2, moolimassat)
print(n3)
```

tulostaa

```
[0.05475702 0.01179412 0.0181937 ]
0.05475701574264203
[0.05475702 0.07548238 0.02646356]
```

NumPyn matemaattiset funktiot

Aloitetaan luomalla yksiulotteinen taulukko *v*:

```
import numpy as np
v = np.arange(1, 11)
# v on array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Summaus ja tulo

NumPyn matemaattisille funktioille annetaan parametrina taulukko, jolloin funktio suorittaa halutun matemaattisen operaation **kaikille taulukon alkioille**.

Taulukoiden alkioiden summaus ja tulo onnistuu *numpy.sum* ja *numpy.prod*-funktioilla:

```
summa = np.sum(v) # Tulos on 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
tulo = np.prod(v) # Tulos on 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 = 3628800
```

Laskutoimituksia voi tehdä myös taulukosta leikatuille siivuille:

```
# Määritellään 3 x 4 matriisi M
M = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

ekan_rivin_summa = np.sum(M[0])          # 1 + 2 + 3 + 4 = 10
vikan_sarakkeen_summa = np.sum(M[:, -1]) # 4 + 8 + 12 = 24
```

Matemaattiset funktiot

Lista NumPyn matemaattisista funktioista: https://docs.scipy.org/doc/numpy/reference/routines.math.html

Esimerkkejä vektorille *v*:

```
log_kympit = np.log10(v)
# array([ 0.        ,  0.30103   ,  0.47712125,  0.60205999,  0.69897   ,
#         0.77815125,  0.84509804,  0.90308999,  0.95424251,  1.        ])
asteet = np.linspace(0, 360, 9)
# array([   0.,   45.,   90.,  135.,  180.,  225.,  270.,  315.,  360.])
radiaanit =  np.radians(asteet)
sinit = np.sin(radiaanit)
```

Tilastolliset funktiot

Lista NumPyn tilastollisista funktioista: https://docs.scipy.org/doc/numpy/reference/routines.statistics.html

Esimerkkejä vektorille *v*:

```
keskiarvo = np.mean(v) # 5.5
suurin = np.amax(v) # 10
pienin = np.amin(v) # 1
```

Taulukoiden yhtäsuuruuden vertailu alkioittain

Kahden taulukon yhtäsuuruutta voi verrata *numpy.allclose*-funktiolla, joka vertaa taulukoita alkioittain:

```
T1 = np.linspace(1.0, 5.0, 5)
# array([ 1.,  2.,  3.,  4.,  5.])
T2 = np.linspace(1.00001, 5.0, 5)
# array([ 1.00001  ,  2.0000075,  3.000005 ,  4.0000025,  5.       ])
if np.allclose(T1, T2, rtol = 0.01):
    print("Samat")
# Taulukoiden T1 ja T2 alkiot ovat yhtäsuuret 1% tarkkuudella, joten tulostuu "Samat"
```

Lineaarialgebra

Lista NumPyn lineaarialgebran funktioista: https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Matriisitulon voi laskea *numpy.dot*-funktiolla

```
# Määritellään kaksi 2x2 neliömatriisia A ja B
A = np.array([[2, 1],
              [1, 2]])
B = np.array([[2, 2],
              [3, 3]])
# Matriisitulo C = AB
C = np.dot(A, B)
#array([[7, 7],
#       [8, 8]])

# Muistathan, että vaihdantalaki ei päde matriisien kertolaskussa!
# Matriisitulo D = BA
D = np.dot(B, A)
# array([[6, 6],
#        [9, 9]])
```

Eli np.dot(A, B) != np.dot(B, A)

Matplotlib-kirjasto

Matplotlib-kirjasto sisältää erittäin monipuoliset työkalut erilaisten kuvaajien tekemiseen:

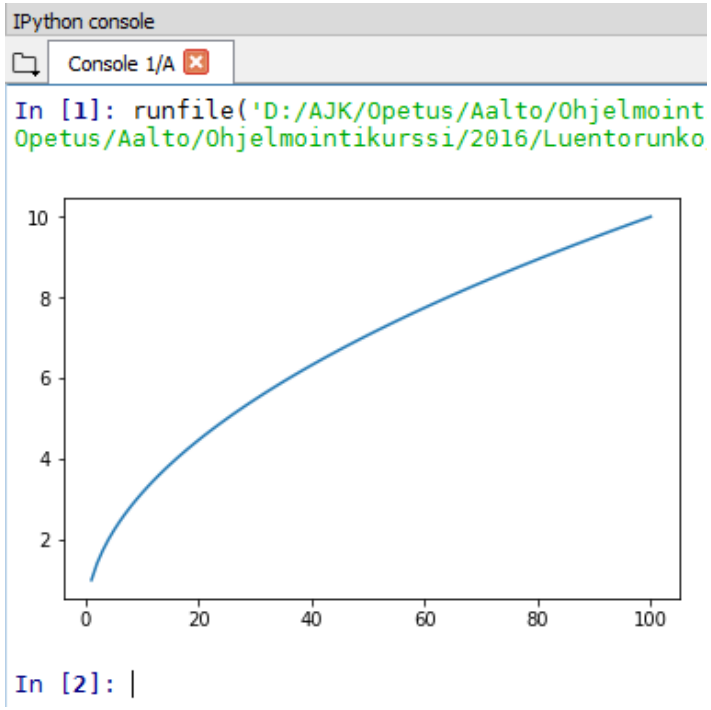
- Katso esim. http://matplotlib.org/gallery.html (sisältää esimerkkikoodeja)
- Erinomainen tutoriaali: http://www.labri.fr/perso/nrougier/teaching/matplotlib/#introduction

Tällä kurssilla tutustumme *matplotlib.pyplot*-moduuliin, joka mahdollistaa kuvaajien piirtämisen hieman Matlabin tapaan.

Kun teet Matplotlib-tehtäviä Spyderissä, kuvaajien pitäisi aueta siististi suoraan IPython-konsoliin:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.arange(1, 101)
Y = np.sqrt(X)
plt.plot(X, Y)
plt.show()
```

lopputulos:



Jos et käytä Spyderiä vaan jotain muuta kehitysympäristöä (esim. Eclipse), sinun pitää itse selvittää, miten Matplotlib-kuvaajat toimivat sen kanssa. Voi olla, että ne avautuvat omina ikkunoinaan.

matplotlib.pyplot-moduuli

Matplotlib-kuvaajia piirrettäessä ohjelmaan pitää aina tuoda *matplotlib.pyplot*-moduuli **import**-käskyllä. Matplotlibiä käytettäessä tarvitaan useimmiten myös NumPy-moduuli. Tällä kurssilla nämä moduulit tuodaan aina seuraavilla käskyllä, jolloin voidaan käyttää lyhenteitä *np* ja *plt*:

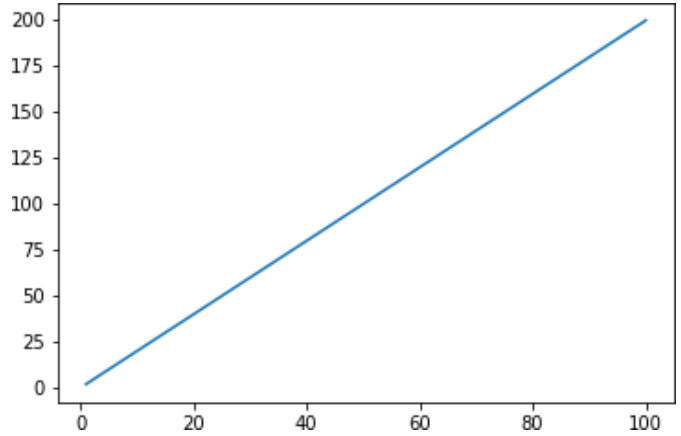
```
import numpy as np
import matplotlib.pyplot as plt
```

Yksinkertainen kuvaaja oletusasetuksilla

Aloitetaan piirtämällä yksinkertainen kuvaaja funktiolle f(x) = 2x oletusasetuksia käyttäen

```
import numpy as np
import matplotlib.pyplot as plt

# Luodaan datat funktiolle f(x) = 2x
X = np.linspace(1, 100, 100)
Y = X * 2
# Luodaan kuvaaja datoja X ja Y käyttäen
plt.plot(X, Y)
# Näytetään kuvaaja
plt.show()
```

Lopputulos:

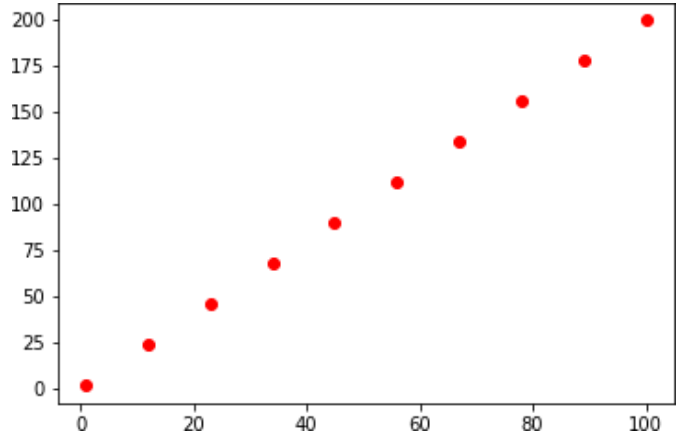


Kuvaajan oletusasetuksien muuttaminen

matplotlib.pyplot.plot-funktion parametrejä muokkaamalla voi vaikuttaa kuvaajan ulkonäköön. Lyhyt yhteenveto parametrien mahdollisista arvoista seuraavassa luvussa.

```
import numpy as np
import matplotlib.pyplot as plt

# Luodaan datat
X = np.linspace(1, 100, 10)
Y = X * 2
# Luodaan kuvaaja
# - Vaihdetaan tyyliksi pisteet ('o')
# - Vaihdetaan väri color-parametrilla punaiseksi
plt.plot(X, Y, 'o', color = 'red')
# Näytetään kuvaaja
plt.show()
```

Lopputulos:

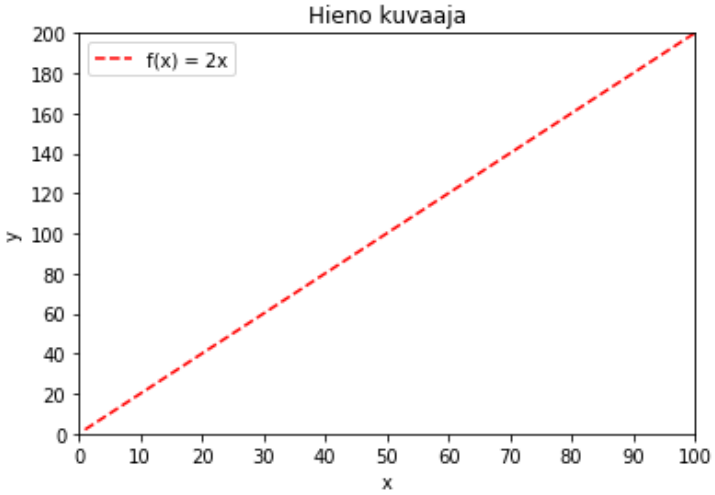


Akseleiden muokkaaminen, selitteiden lisääminen ja kuvien tallentaminen

Muokataan esimerkkikuvaajaa edelleen

```
import numpy as np
import matplotlib.pyplot as plt

# Luodaan datat
X = np.linspace(1, 100, 100)
Y = X * 2
# Luodaan katkoviiva-kuvaaja (mukana väri ja teksti 'f(x) = 2x' selitettä varten)
plt.plot(X, Y, '--', color = 'red', label = 'f(x) = 2x')
# Nimetään akselit
plt.xlabel('x')
plt.ylabel('y')
# Asetetaan akselivali
plt.xlim(0, 100)
plt.ylim(0, 200)
# Asetetaan akselin numerot (ticks)
plt.xticks(np.arange(0, 101, 10))
plt.yticks(np.arange(0, 201, 20))
# Lisätään selite (legend). Se käyttää plot-funktion label-parametriä.
plt.legend(loc = 'upper left')
# Lisätään otsikko (title)
plt.title('Hieno kuvaaja')
# Tallennetaan kuvaaja myös png ja PDF-muodossa
plt.savefig("kuvaaja.png", dpi = 300)
plt.savefig("kuvaaja.pdf")
# Lopuksi piirretaan kuvaaja
plt.show()
```

Lopputulos:



Kuvaajan voi siis **tallentaa tiedostoon** *plt.savefig*-funktiolla. Kuvaajat tallentuvat samaan hakemistoon, missä ohjelma ajetaan (paitsi jos annat *plt.savefig*-funktiolle kokonaisen tiedostopolkun, esimerkiksi "C:\Users\pipetti\hienokuva.pdf").

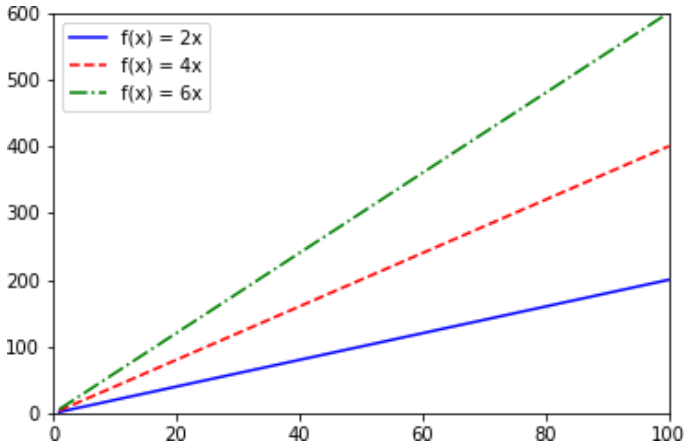
Huom! *plt.savefig*-funktiota pitää kutsua **ennen** *plt.show*-funktiota. Matplotlib päättelee tiedoston tyypin sen päätteestä. Tyypillisiä vaihtoehtoja ovat **pdf**, **eps** ja **png** (png-kuvien tarkkuutta voi nostaa dpi-parametrilla).

Useampi kuvaaja samassa akselistossa

Matplotlibillä on helppo piirtää useita kuvaajia samaan akselistoon. Riittää, kun kutsuu *plt.plot*-funktiota useamman kerran:

```
import numpy as np
import matplotlib.pyplot as plt

# Luodaan kolmet XY-datat
X1 = np.linspace(1, 100, 100)
Y1 = X1 * 2
X2 = np.linspace(1, 100, 100)
Y2 = X2 * 4
X3 = np.linspace(1, 100, 100)
Y3 = X3 * 6

# Luodaan kolme kuvaajaa
# Huomaa lyhennetty merkintä, jossa yhdistetty väri ja viivan tyyppi
plt.plot(X1, Y1, 'b-', label = 'f(x) = 2x')
plt.plot(X2, Y2, 'r--', label = 'f(x) = 4x')
plt.plot(X3, Y3, 'g-.', label = 'f(x) = 6x')

# Asetetaan akselien rajat ja luodaan selite
plt.xlim(0, 100)
plt.ylim(0, 600)
plt.legend(loc = 'upper left')

# Näytetään kuvaajat
plt.show()
```

Lopputulos:



Muuntyyppiset kuvaajat

Matplotlib.pyplot-moduuli sisältää valtavasti erilaisia toiminnallisuuksia. Erityyppisiä kuvaajia on lukuisia. Hyödyllisiä kuvaajatyyppejä ovat varmasti esimerkiksi pyplot.scatter (XY-pistekuvaaja) ja pyplot.bar (pylväsdiagrammi).

Lisätietoja:

- http://matplotlib.org/gallery.html
- http://www.labri.fr/perso/nrougier/teaching/matplotlib/#introduction
- https://matplotlib.org/devdocs/api/_as_gen/matplotlib.pyplot.html

Matplotlib-määritelmiä

Ajantasainen yhteenveto matplotlib.pyplot.plot-funktion parametreista löytyy osoitteesta: https://matplotlib.org/devdocs/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot

Alla lyhyt yhteenveto.

Viivatyypit:

character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker
'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

Värit (katso myös http://matplotlib.org/api/colors_api.html)

character	color
‘b’	blue
‘g’	green
‘r’	red
‘c’	cyan
‘m’	magenta
‘y’	yellow
‘k’	black
‘w’	white

Selitteen (legend) sijainti (plt.legend-funktio)

Location String	Location Code
‘best’	0
‘upper right’	1
‘upper left’	2
‘lower left’	3
‘lower right’	4
‘right’	5
‘center left’	6
‘center right’	7
‘lower center’	8
‘upper center’	9
‘center’	10

NumPy-polynomit

Kemian tekniikassa haluamme usein sovittaa polynomeja mittausdataan. Polynomien käsittelyssä voimme käyttää *numpy.poly1d*-toimintoa (varsinaisesti kyseessä on "luokka", joita käsitellään olio-ohjelmoinnin yhteydessä kurssin 6. kierroksella).

Polynomien luominen

```
import numpy as np
import matplotlib.pyplot as plt

# Luodaan polynomi x^2 + 4x + 3
# Kutsutaan poly1d:tä kertoimilla [1, 4, 3], eli 1*x^2 + 4*x^1 + 3*x^0
pol = np.poly1d([1, 4, 3])
print("Polynomi on (eksponentit ylärivillä):\n", pol)
```

tulostaa

```
Polynomi on (eksponentit ylärivillä):
    2
1 x + 4 x + 3
```

Polynomien perusominaisuudet

```
# Polynomin arvon laskeminen yksittäisessä pisteessä
pol = np.poly1d([1, 4, 3])
print("Arvo pisteessa x = 5:", pol(5))
# Laskee siis 5^2 + 5*4 + 3 = 48

# Polynomin juuret (nollakohdat): pol.r
print("Juuret:", pol.r)

# Polynomin derivaatta: pol.deriv
# pol.deriv palauttaa uuden polynomin (poly1d-olion )
der = pol.deriv()
print("Derivaatta:", der)
print("Derivaatan arvo pisteessä x = 5:", der(5))
```

tulostaa

```
Arvo pisteessa x = 5: 48
Juuret: [-3. -1.]
Derivaatta:
2 x + 4
Derivaatan arvo pisteessä x = 5: 14
```

Polynomeilla laskeminen ja kuvaajien piirtäminen

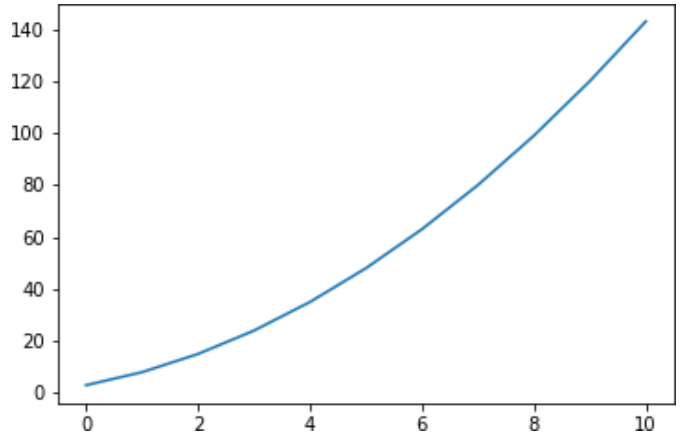
```
# Polynomilla voi tehdä laskutoimituksia
pol = np.poly1d([1, 4, 3])
print(pol**2)
```

tulostaa (eksponentit ylärivillä)

```
   4     3      2
1 x + 8 x + 22 x + 24 x + 9
```

```
# Polynomin arvon voi laskea myös useassa pisteessä
X = np.arange(0, 11) # X on array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
Y = pol(X) # Y on array([  3,   8,  15,  24,  35,  48,  63,  80,  99, 120, 143])

# Luodaan polynomin kuvaaja
plt.plot(X, Y)
plt.show()
```

Kuvaaja on:

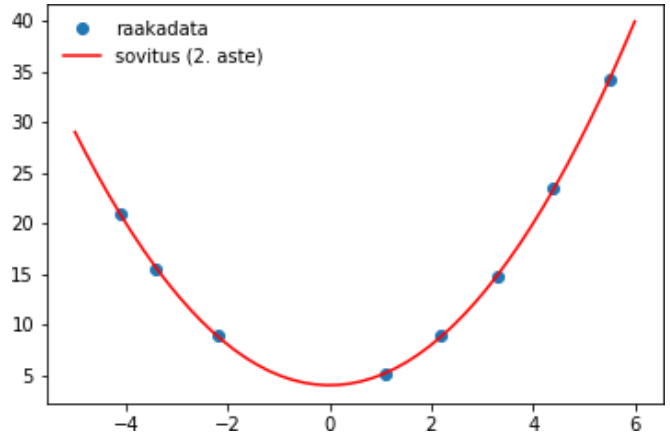


Polynomisovitukset

Polynomisovituksia voi tehdä np.polyfit-funktiolla. Tarkastellaan esimerkkiä:

```
import numpy as np
import matplotlib.pyplot as plt

# Luodaan XY-datat
X = np.array([-4.1, -3.4, -2.2, 1.1, 2.2, 3.3, 4.4, 5.5])
Y = np.array([20.91, 15.46, 8.94, 5.11, 8.94, 14.79, 23.46, 34.15])
# Luodaan raakadatan kuvaaja
plt.plot(X, Y, 'o', label = 'raakadata')

# Sovitusfunktio: np.polyfit(xdata, ydata, polynomin_aste)
# Tehdään raakadatalle toisen asteen polynomisovitus:
kertoimet = np.polyfit(X, Y, 2)
# kertoimet on nyt NumPy-taulukko, joka sisältää sovitetun polynomin kertoimet [a, b, c]:
# array([ 0.9996726 , -0.00626338,  4.00940658])
#         a * x^2      b * x        c * 1

# Tehdään kertoimista sovituspolynomi (np.poly1d)
sovitus = np.poly1d(kertoimet)
# sovitus on nyt poly1d-olio, jolla on sovitetun polynomin kertoimet:
# poly1d([ 0.9996726 , -0.00626338,  4.00940658])
# HUOMAA, että merkintä sovitus[2] palauttaa 2. asteen termin kertoimen 0.9996726!
# np.polyfit- funktion tapauksessa 2. asteen termi on kertoimet[0]!

# Lasketaan sovituspolynomin y-arvot usealle x-arvolle
pol_X = np.linspace(-5.0, 6.0, 50)
pol_Y = sovitus(pol_X)

# Luodaan sovituspolynomin kuvaaja
# Kuvaaja lisätään plt.plot-funktiolla samaan kuvaan raakadatan kanssa
plt.plot(pol_X, pol_Y, color = 'red', label = 'sovitus (2. aste)')
plt.legend(loc = 'upper left', frameon = False)
plt.show()
```

Lopputulos:



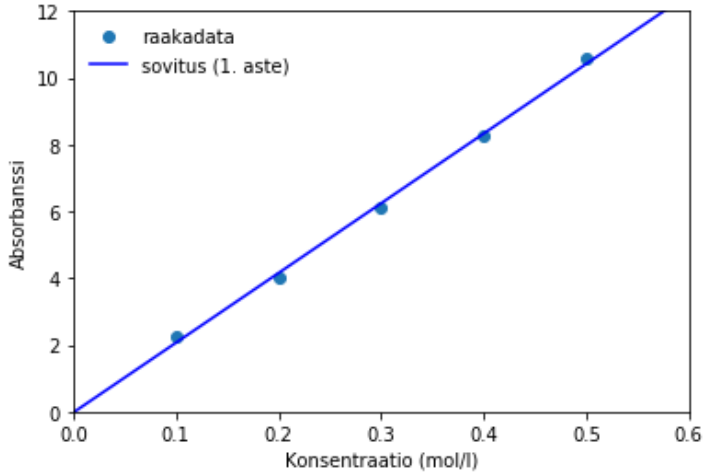
Korrelaatiokerroin

Suorien sovittamisen yhteydessä voi laskea X- ja Y-datojen välisen Pearsonin korrelaatiokertoimen *np.corrcoef*-funktiolla. Kahden muuttujan välinen korrelaatio kertoo mahdollisesta lineaarisesta riippuvuudesta. Jos korrelaatiokerroin on lähellä arvoa 1, voidaan toisen muuttujan arvo arvioida tietämällä vain toisen arvo. Mitä lähempänä korrelaatiokerroin on nollaa, sitä enemmän arvioituun arvoon liittyy epävarmuutta.

numpy.corrcoef -funktiolle annetaan X- ja Y-datapisteet, jolloin se palauttaa korrelaatiokertoimet 2x2 taulukkona [[xx, xy], [yx, yy]]. Useimmiten riittää, että poimimme taulukosta xy-korrelaation (indeksi [0, 1]).

```
import numpy as np
import matplotlib.pyplot as plt

# Luodaan taulukot mittausarvoista
konsentraatiot = np.array([0.1, 0.2, 0.3, 0.4, 0.5])
absorbanssi = np.array([2.24, 4.02, 6.11, 8.27, 10.56])

# Luodaan kuvaaja raakadatoista
plt.plot(konsentraatiot, absorbanssi, 'o', label = 'raakadata')

# Sovitusfunktio: np.polyfit(xdata, ydata, polynomin_aste)
# Tehdään 1.  asteen polynomisovitus XY-dataan:
kertoimet = np.polyfit(konsentraatiot, absorbanssi, 1)
# kertoimet on nyt NumPy-taulukko, joka sisältää sovitetun polynomin kertoimet

# Tehdään kertoimista polynomi np.poly1d
polynomi = np.poly1d(kertoimet)
# Lasketaan y:n arvot usealle x:n arvolle
pol_X = np.linspace(0.0, 0.6, 60)
pol_Y = polynomi(pol_X)

# Luodaan sovituspolynomin kuvaaja
# Kuvaaja lisätään plot-funktiolla samaan kuvaan raakadatan kanssa
plt.plot(pol_X, pol_Y, color = 'blue', label = 'sovitus (1. aste)')

# Kuvaajan asetukset
plt.xlabel('Konsentraatio (mol/l)')
plt.ylabel('Absorbanssi')
plt.xlim(0, 0.6)
plt.ylim(0, 12)
plt.legend(loc = 'upper left', frameon = False)

plt.show()

# Lasketaan Pearsonin korrelaatiokerroin ja sen neliö
R = np.corrcoef(konsentraatiot, absorbanssi)[0, 1]
R_toiseen = R**2
print("Sovituksen R^2 on:", round(R_toiseen, 3))
```

Lopputuloksena saadaan kuvaaja:



ja tulostus

```
Sovituksen R^2 on: 0.998
```

Eli tässä tapauksessa X- ja Y-arvojen välillä on erittäin vahva lineaarinen korrelaatio (absorbanssi on suoraan verrannollinen konsentraatioon).

Kierros 5

Viidennellä kierroksella tutustumme tiedostojen käyttöön Pythonissa. Datan lukeminen tiedostosta ja kirjoittaminen tiedostoon on aivan keskeinen tehtävä ohjelmoinnissa. Tutustumme tavallisiin tekstitiedostoihin ja JSON-tiedostoihin. JSON-tiedostoformaatti on erittäin käytännöllinen esim. tietorakenteiden tallentamisessa.

Kierroksen aiheisiin kuuluu myös virheenkäsittely. Ohjelmien suorituksessa tulee usein vastaan virhetilanteita, joista pitäisi selvitä kunnialla (käyttäjä antaa merkkijonon, vaikka piti antaa luku, avattava datatiedosto onkin tyhjä, jne...). Opettelemme käyttämään try-except-finally -rakenteita virheenkäsittelyyn.

Tiedostojen avaaminen ja käsittely

Tiedostojen avaaminen *open*-funktiolla

Tiedostot avataan Pythonissa *open*-funktiolla, jota kutsutaan näin:

```
tiedosto = open(tiedoston_nimi, tila)
```

Esimerkiksi komento

```
datat = open("data.txt", "r")
```

avaa tiedoston *data.txt* lukemista varten (parametrin *tila* arvo on "r", eli read).

Oletuksena avattava tiedosto avataan samasta hakemistosta, missä ohjelmaa suoritetaan. Tyypillisesti tämä on sama hakemisto, missä ohjelman .py-tiedosto sijaitsee.

Voit antaa avattaessa myös kokonaisen tiedostopolun:

```
datat = open("Z:\datat\data.txt", "r")
```

Parametrin *tila* tyypillisimmät arvot ovat

- "r" eli **read**: avataan tiedosto lukemista varten:
- "w" eli **write**: avataan tiedosto kirjoittamista varten:
  - Jos tiedostoa ei ole olemassa, *open* luo uuden tiedoston
  - Jos tiedosto on jo olemassa, *open* luo uuden tyhjän tiedoston **olemassaolevan tiedoston päälle**!
- "a" eli **append**: avataan tiedosto kirjoittamista varten:
  - Jos tiedostoa ei ole olemassa, *open* luo uuden tiedoston
  - Jos tiedosto on jo olemassa, tiedostoon kirjoitettavat tiedot lisätään sen loppuun (ei tyhjennä tiedostoa kuten ”w”)

Tiedostojen käsittely ja sulkeminen

open-funktio joka palauttaa ns. tiedosto-olion. jonka avulla tiedostoa voi käsitellä. Avataan tiedosto mittaukset.txt lukemista varten:

```
mittaukset = open("Z:\fyke\mittaukset.txt", "r")
```

Muuttuja *mittaukset* on nyt *tiedosto-olio*, jonka avulla tiedostoa käsitellään. Esimerkiksi ensimmäisen rivin lukeminen tiedostosta tapahtuu näin:

```
rivi = mittaukset.readline()
```

Tiedoston lukemisesta ja tiedostoon kirjoittamisesta lisää yksityiskohtia seuraavassa luvussa.

Huom! Kun tiedoston käsittely lopetetaan se pitää sulkea *close*-funktiolla:

```
mittaukset.close()
```

Tiedoston sulkeminen on erittäin tärkeää! Jos kirjoitat tiedostoon, mutta jätät tiedoston sulkematta, tiedot eivät välttämättä tallennu!

Eli yhteenvetona tiedoston avaaminen, 1. rivin lukeminen ja tiedoston sulkeminen:

```
mittaukset = open("mittaukset.txt", "r")
rivi1 = mittaukset.readline()
mittaukset.close()
print("Eka rivi:", rivi1)
```

Datan lukeminen ja kirjoittaminen

Luodaan tiedosto *halogeenit.txt*, joka sisältää halogeenien symbolit:

```
halogeenit = ['F', 'Cl', 'Br', 'I']
tiedosto = open("halogeenit.txt", "w")
for halogeeni in halogeenit:
    # Kirjoitetaan jokainen symboli omalle rivilleen (\n on rivinvaihto)
    tiedosto.write(halogeeni + "\n")
# Lopuksi suljetaan tiedosto!
tiedosto.close()
```

Luetaan seuraavaksi juuri luomamme tiedoston sisältö rivi kerrallaan **for**-silmukan avulla:

```
halogeenit1 = []
halogeenit2 = []
tiedosto = open("halogeenit.txt", "r")
for rivi in tiedosto:
    halogeenit1.append(rivi)
    # str.rstrip()-funktio poistaa rivinvaihdon rivin lopusta
    halogeenit2.append(rivi.rstrip())

# Suljetaan tiedosto!
tiedosto.close()

print(halogeenit1)
print(halogeenit2)
```

Koodi tulostaa:

```
['F\n', 'Cl\n', 'Br\n', 'I\n']
['F', 'Cl', 'Br', 'I']
```

Eli *str.rstrip*-funktiolla päästin eroon tiedostossa olleista rivinvaihtomerkistä, jotka Python sisällyttää lukemiinsa riveihin.

str.split-funktio

Luodaan tiedosto *neliot.txt*, joka sisältää numerot 1-100 ja niiden neliöt välilyönnillä erotettuna. Tiedoston rakenne on siis (kolme ensimmäistä riviä):

```
1 1
2 4
3 9
```

Huomaa, miten *str.format*-funktiota voi hyödyntää myös tiedostoon kirjoitettaessa:

```
tiedosto = open("neliot.txt", "w")
for i in range(1, 101):
    tiedosto.write("{:d} {:d}\n".format(i, i * i))
tiedosto.close()
```

Avataan luotu tiedosto lukemista varten ja hyödynnetään *str.split*-funktiota:

```
tiedosto = open("neliot.txt", "r")
for rivi in tiedosto:
    # rivi on siis merkkijono, esim. "3 9\n". Funktio str.split() palauttaa
    # listan, johon merkkijonon sanat on pilkottu alkioiksi.
    # Jos siis rivi on "3 9\n", rivi.split() palauttaa listan ["3", "9"]
    # str.split-funktio siivoaa myös rivinvaihdot pois
    # Funktion palauttamat arvot voi lukea suoraan muuttujiin:
    luku, nelio = rivi.split()
    # Toinen vaihtoehto olisi lukea arvot listaan:
    # datat = rivi.split() # datat[0] == luku ja datat[1] == nelio

    # Luvut ovat nyt edelleen merkkijonoina. Ne voisi muuntaa
    # kokonaisluvuiksi int()-funktiolla, mutta nyt riittää tulostus
    print("Luvun", luku, "nelio on", nelio)
tiedosto.close()
```

Koodi tulostaa (vain kolme ensimmäistä ja kolme viimeistä riviä näkyvissä):

```
Luvun 1 neliö on 1
Luvun 2 neliö on 4
Luvun 3 neliö on 9
...
Luvun 98 neliö on 9604
Luvun 99 neliö on 9801
Luvun 100 neliö on 10000
```

Numeroarvojen lukeminen tiedostosta

Meillä on käytössämme datatiedosto *moolimassat.txt*, joka sisältää kullakin rivillä yhdisteen nimen, moolimassan (g/mol) ja massan (g) välilyönnillä erotettuna (tiedoston kaksi ensimmäistä riviä):

```
H2O  18.015 2.3
NaCl 58.44  4.5
```

Luetaan nyt tiedoston sisältö niin, että voimme hyödyntää lukuarvoja laskennassa

```
tiedosto = open("moolimassat.txt", "r")
for rivi in tiedosto:
    datat = rivi.split()
    # datat on nyt kolmen merkkijonon lista, esim.: [H2O, 18.015, 2.3]
    nimi = datat[0]
    moolimassa = float(datat[1])
    massa = float(datat[2])
    ainemaara = massa / moolimassa
    print("Yhdisteen {} ainemaara on {:4.3f}".format(nimi, ainemaara))

# Lopuksi suljetaan tiedosto
tiedosto.close()
```

Koodi tulostaa (vain kolme ensimmäistä riviä näkyvissä):

```
Yhdisteen H2O ainemaara on 0.128
Yhdisteen NaCl ainemaara on 0.077
Yhdisteen KF ainemaara on 0.114
...
```

Toinen esimerkki numeroarvojen lukemisesta

Meillä on käytössämme datatiedosto temps.txt, joka sisältää kullakin rivillä alkuaineen symbolin, nimen, sulamispisteen (°C) ja kiehumispisteen (°C). Tiedoston kaksi ensimmäistä riviä:

```
Sc, skandium   , 1541.0 ,2830
Ti  , titaani  , 1668.0,3287
```

Huomaa, että tiedot ovat nyt pilkulla eroteltuna ja sisältävät ylimääräisiä välilyöntejä. Luetaan nyt tiedoston sisältö hyödyntämällä *str.split*-funktion *sep*-parametriä, jolla voi määrätä datapisteiden välisen erottimen. Lisäksi tarvitsemme *str.strip*-funktiota ylimääräisten välilyöntien poistamiseen.

```
metallit = []
tiedosto = open("temps.txt", "r")
for rivi in tiedosto:
    datat = rivi.split(sep = ',')
    # datat on nyt neljän merkkijonon lista, esim.: ["Sc", " skandium   ", "1541.0 ", "2830"]
    # Käytetään lisäksi str.strip()-funktiota, joka karsii tyhjät merkit
    # (välilyönnit, rivinvaihdot) merkkijonon vasemmalta ja oikealta puolelta.
    # esim "  testi  \n".strip() -> "testi"
    symboli = datat[0].strip()
    nimi = datat[1].strip()
    # sulamispiste ja kiehumispiste liukulukuina. str.strip()-funktiota ei tarvita,
    # float osaa jättää ylimääräiset välilyönnit huomioimatta
    sulamispiste = float(datat[2])
    kiehumispiste = float(datat[3])

    print("Alkuaineen {:2s} sulamispiste on {:4.0f} C "
          "ja kiehumispiste {:4.0f} C".format(symboli, sulamispiste, kiehumispiste))

# Lopuksi suljetaan tiedosto!
tiedosto.close()
```

Koodi tulostaa (vain kolme ensimmäistä riviä näkyvissä):

```
Alkuaineen Sc sulamispiste on 1541 C ja kiehumispiste 2830 C
Alkuaineen Ti sulamispiste on 1668 C ja kiehumispiste 3287 C
Alkuaineen V  sulamispiste on 1910 C ja kiehumispiste 3407 C
...
```

JSON-tiedostot

Pythonissa voi hyödyntää JSON-muotoisia tiedostoja, joiden avulla on hyvin helppoa tallentaa dataa tiedostoihin ja lukea sitä (JSON = JavaScript Object Notation). JSON on ohjelmointikielestä riippumaton, avoin ja standardoitu tiedostoformaatti, jota voi käyttää useiden eri ohjelmointikielien kanssa. JSON-tiedostot ovat tekstimuotoisia ja ihmisten luettavissa (ja muokattavissa).

JSON-tiedostojen luominen

Luodaan JSON-tiedosto, jonne tallennamme listoja sisältävän sanakirjan:

```
# JSON-tiedostoja käytettäessä tarvitaan json-moduuli
import json

# Määritellään listoja sisältävä sanakirja
alkuaineet = {'O': ['Happi', 15.999], 'C': ['Hiili', 12.011], 'N': ['Typpi', 14.007]}

# Luodaan tiedosto
tiedosto = open("alkuaineet.json", "w")
# Kirjoittaminen json.tiedostoon hoidetaan json.dump()-funktiolla:
# json.dump(TALLENNETTAVA_DATA, TIEDOSTO-OLIO, indent = 4)
# indent = 4 -parametrilla tiedot tallentuvat selkeässä muodossa
json.dump(alkuaineet, tiedosto, indent = 4)
tiedosto.close()
```

Lopputuloksena saatava tiedosto *moolimassat.json* näyttää tältä:

```
{
    "O": [
        "Happi",
        15.999
    ],
    "C": [
        "Hiili",
        12.011
    ],
    "N": [
        "Typpi",
        14.007
    ]
}
```

JSON-tiedostomuodon käyttäminen on suositeltavaa, kun haluat tallentaa monimutkaisempia tietorakenteita. Muuttujatyypit **int**, **float**, **str**, **boolean** sekä tietorakenteet **list** ja **dict** voi tallentaa sellaisinaan JSON-tiedostomuodossa.

JSON-tiedostojen lukeminen

Tietojen lukeminen yllä luodusta *alkuaineet.json*-tiedostosta on näin helppoa:

```
import json
tiedosto = open("alkuaineet.json", "r")
# Tiedot ladataan json.load(TIEDOSTO-OLIO) -funktiolla
alkuaineet2 = json.load(tiedosto)
# alkuaineet2 on nyt sanakirja
tiedosto.close()

# Hyödynnetään vielä luettuja tietoja
for alkuaine, tiedot in alkuaineet.items():
    # alkuaine on merkkijono (esim. "O"), tiedot on kahden alkion lista [nimi, atomiopaino]
    print("Alkuaineen {:s} tiedot: Nimi = {:s}, atomipaino = {:6.3f}".format(alkuaine, tiedot[0], tiedot[1]))
```

Koodi tulostaa:

```
Alkuaineen O tiedot: Nimi = Happi, atomipaino = 15.999
Alkuaineen C tiedot: Nimi = Hiili, atomipaino = 12.011
Alkuaineen N tiedot: Nimi = Typpi, atomipaino = 14.007
```

Tiedostojen helppo käsittely NumPy:llä

NumPy-kirjasto sisältää käteviä funktioita tiedostojen käsittelyyn. Nämä ovat hyödyllisiä etenkin numeerista dataa luettaessa. Tekstitiedostoja voi lukea ja kirjoittaa *numpy.loadtxt*- ja *numpy.savetxt*-funktioilla. Tarkastellaan esimerkkiä, jossa meillä on tilavuus- ja painedataa tiedostossa painedata.txt seuraavassa muodossa (vain kommenttirivi ja kolme ensimmäistä datariviä näkyvissä):

```
# V (dm^3)    p (Pa)
0.21          1856455
0.31          1176944
0.41          838490
```

Esimerkki, kuinka tiedoston voi lukea *numpy.loadtxt*-funktiolla ja tulokset voi kirjoittaa tiedostoon *numpy.savetxt*-funktiolla:

```
import numpy as np

Vp_data = np.loadtxt("painedata.txt")
# Vp_data on nyt NumPy-taulukko, jossa on kaksi saraketta: V ja p
R = 8.3144598            # J K^-1 mol ^-1
n = 0.05                 # mol
V = Vp_data[:, 0] / 1000 # 1. sarake (V, dm^3). Muunnetaan dm^3 -> m^3
p = Vp_data[:, 1]        # 2. sarake (p, Pa)
T = V * p / (n * R)      # K

np.savetxt("T.txt", T, fmt="%.3f", header = "T (K)")
```

Tiedostoja ei siis tarvitse avata ja sulkea, koska NumPy hoitaa nämä puolestasi. Tiedoston *T.txt* neljä ensimmäistä riviä:

```
# T (K)
937.777
877.634
826.947
```

header-parametrin arvo tulee siis tiedoston alkuun kommenttiriviksi. Ja toisaalta *loadtxt* osasi automaattisesti jättää #-merkillä alkavan kommenttirivin lukematta.

fmt-parametrin rakenne on periaatteessa sama kuin *str.format*-funktiolla, mutta kaarisulut korvautuvat %-merkillä, eikä :-merkkiä käytetä. Lisätietoja *numpy.savetxt*-funktion ohjeesta.

Huom! Jos tarkastelet *savetxt*-tiedoston luomaa tiedostoa Windowsissa, rivinvaihdot eivät välttämättä näy oikein esimerkiksi Notepadissä. Tämä johtuu siitä, etteivät monet Windows-ohjelmat ymmärrä \n-rivinvaihtoa oikealla tavalla. Tiedosto näkyy oikein esim. Spyderissä.

numpy.column_stack

Kun sinulla on useita yksiulotteisia taulukoita, jotka haluat tallentaa sarakkeina tiedostoon, numpy.column_stack on hyvin hyödyllinen funktio. Laajennetaan ylläolevaa esimerkkiä niin, että tallennamme tulostiedostoon myös alkuperäiset tilavuus- ja painedatat.

```
import numpy as np

Vp_data = np.loadtxt("painedata.txt")
R = 8.3144598            # J K^-1 mol ^-1
n = 0.05                 # mol
V = Vp_data[:, 0] / 1000 # 1. sarake (V, dm^3). Muunnetaan dm^3 -> m^3
p = Vp_data[:, 1]        # 2. sarake (p, Pa)
T = V * p / (n * R)      # K
# Käytetään np.column_stack -funktiota, jolla yksiulotteiset
# NumPy-taulukot voi liittää yhteen kaksiulotteisen taulukon sarakkeiksi
VpT_data = np.column_stack([V * 1000, p, T]) # Tilavuudet m^3 -> dm^3

np.savetxt("VpT.txt", VpT_data, fmt="%10.3f %10.0f %10.1f",
           header = "V (dm^3)    p (Pa)       T (K)")
```

Huomaa, miten fmt-parametrille annetaan oma muotoiluparametri jokaiselle sarakkeelle. Tiedoston *VpT.txt* neljä ensimmäistä riviä:

```
# V (dm^3)    p (Pa)       T (K)
     0.210    1856455      937.8
     0.310    1176944      877.6
     0.410     838490      826.9
```

Virhetilanteiden käsittely (try-except-finally)

Poikkeukset

Hyvässä ohjelmakoodissa varaudutaan aina erilaisiin virhetilanteisiin, kuten

- Käyttäjän antama virheellinen syöte
- Tiedoston avaaminen epäonnistuu

Pythonissa virhetilanteet hoidetaan **poikkeusten** (engl. exception) avulla.

Oletkin varmasti jo kohdannut lukuisia poikkeuksia kurssin aikana. Esimerkiksi jos olet yrittänyt muuntaa vääränlaista merkkijonoa lukuarvoksi:

```
luku = int("kolme")
```

Python on ilmoittanut ValueError-nimisestä poikkeuksesta (IPython-konsolin tuloste Spyderissä)

```
ValueError: invalid literal for int() with base 10: 'kolme'
```

Poikkeusten "nappaaminen" ohjelmakoodissa

Virhetilanteessa Python "nostaa" (engl. raise) poikkeuksen. Poikkeuksen voi "napata" (engl. catch) ja käsitellä, jolloin se ei johda ohjelman suorituksen keskeytymiseen.

Poikkeusten nappaamiseen ja käsittelemiseen käytetään **try-except** -rakennetta:

```
while True:
    try:
        luku = float(input("Anna liukuluku:\n"))
        # Jos suoritus jatkui tänne, käyttäjä antoi kelvollisen liukuluvun
        break
    except ValueError:
        # Napataan ValueError-poikkeus (virheellinen lukuarvo)
        print("Virheellinen lukuarvo!")
        # Virhe on nyt käsitelty ja ohjelman suoritus palaa while-silmukan alkuun

print("Annoit luvun", luku)
```

Esimerkkisuoritus:

```
Anna liukuluku:
> kolme piste kaksi
Virheellinen lukuarvo!

Anna liukuluku:
> 3.2
Annoit luvun 3.2
```

Poikkeus tiedostoa avattaessa

Tiedostoja käsitellessä voi tulla vastaan virhetilanteita (esimerkiksi yritetään avata tiedostoa, jota ei ole olemassa). Tällöin pitää napata virhe OSError:

```
# Luetaan rivit tiedostosta
try:
    tiedosto = open("rivit.txt", "r")
    for rivi in tiedosto:
        print(rivi)
except OSError:
    print("Virhe avattaessa tiedostoa rivit.txt")
```

Sisäkkäiset try-lausekkeet

Monesti tarvitaan sisäkkäisiä try-lausekkeita, joilla hoidetaan erityyppiset virheet. Hyvä nyrkkisääntö on, että try-avainsana tulisi olla mahdollisimman lähellä riviä, jossa odotat virheen tapahtuvan (esimerkiksi alla).

```
nimi = "luku.txt"
try:
    # Yritetään avata tiedosto, tämä voi johtaa virheeseen
    tiedosto = open(nimi, "r")
    # Tiedosto aukesi onnistuneesti, luetaan siitä
    for rivi in tiedosto:
        try:
            # Yritetään muuntaa teksti liukuluvuksi
            luku = float(rivi)
            # Onnistui, tulostetaan luku
            print("Tiedosto sisälsi luvun", luku)
        except ValueError:
            # float()-funktio aiheutti ValueError-virheen
            print("Virheellinen lukuarvo {:s} tiedostossa {:s}".format(rivi.strip(), nimi))

    # Suljetaan lopuksi tiedosto
    tiedosto.close()
except OSError:
    # Tänne päädytään, jos open-funktio epäonnistui
    print("Virhe avattaessa tiedostoa", nimi)
```

Jos kaikki menee hyvin, ohjelma tulostaa:

```
Tiedosto sisälsi luvun 1.0
```

Jos tiedostoa ei ole olemassa, ohjelma tulostaa:

```
Virhe luettaessa tiedostoa luku.txt
```

Jos tiedostossa on virheellisiä lukuja, ohjelma tulostaa esimerkiksi

```
Virheellinen lukuarvo 1.0a tiedostossa luku.txt
Tiedosto sisälsi luvun 2.0
```

try-except-finally

try-except-finally-rakenteella voidaan varmistaa, että jokin asia tehdään varmasti, vaikka virheitä syntyisi. Luetaan lukuarvo tiedostosta ja napataan poikkeukset:

```
nimi = "teksti.txt"
try:
    # Yritetään avata tiedosto, tämä voi johtaa virheeseen
    tiedosto = open(nimi, "r")
    try:
        # Tiedosto aukesi onnistuneesti, yritetään lukea rivi
        teksti = tiedosto.read()
        # Onnistui, tulostetaan sisältö
        print("Tiedosto sisälsi tekstin", teksti)
    except OSError:
        # read()-funktio aiheutti OSError-virheen, tiedostoa ei voi lukea
        print("Tiedostoa {:s} ei voitu lukea".format(nimi))
    finally:
        # Suljetaan tiedosto riippumatta siitä, oliko virheitä vai ei
        print("Suljetaan tiedosto")
        tiedosto.close()
except OSError:
    # Tänne päädytään, jos open-funktio epäonnistui.
    print("Virhe avattaessa tiedostoa", nimi)
    # Tiedostoa ei avattu, joten sitä ei tarvitse myöskään sulkea
```

Jos kaikki menee hyvin, ohjelma tulostaa:

```
Tiedosto sisälsi teksti keukeu
Suljetaan tiedosto
```

Jos tiedostoa ei ole olemassa, ohjelma tulostaa:

```
Virhe avattaessa tiedostoa teksti.txt
```

Jos tiedostossa on virheellisiä lukuja, ohjelma tulostaa

```
Tiedostoa teksti.txt ei voitu lukea
Suljetaan tiedosto
```

Ohjelma siis sulkee viimeisenä tekonaan tiedoston. Tämä on tärkeää ja tiedostojen kanssa tuleekin aina käyttää **try-finally** -rakennetta. Helpoiten tämän vaatimuksen voi kuitata käyttämällä **with**-lausetta.

Tiedostojen avaaminen with-lausekkeella

Pythonissa on kätevä **with**-lauseke, joka kutsuu automaattisesti close-funktiota ja näin try-finally-rakennetta ei tarvita. Luetaan tiedosto moolimassat.json käyttäen **with**-lauseketta:

```
with open("moolimassat.json", "r") as tiedosto:
    moolimassat = json.load(tiedosto)
```

with-lauseke korvaa siis seuraavan try-finally -rakenteen:

```
tiedosto = open("moolimassat.json", "r")
try:
    moolimassat = json.load(tiedosto)
finally:
    # Tämä osio suoritetaan aina
    tiedosto.close()
```

Koska myös *open*-funktion mahdolliset virheet on tärkeää käsitellä, with-lausekkeesta on parasta käyttää seuraavaa muotoa:

```
try:
    with open("moolimassat.json", "r") as tiedosto:
        moolimassat = json.load(tiedosto)
except OSError:
    print("Tiedoston avaaminen epaonnistui!")
```

Tämä viimeinen esimerkki on **minimivaatimus** virheenkäsittelylle tiedostoja avattaessa!

try-except-else(-finally)

try-except-rakenteeseen voi yhdistää myös else-osan, joka suoritetaan siinä tapauksessa, että try-osio ei aiheuttanut poikkeuksia:

```
try:
    luku = float(input("Anna luku:\n"))
except ValueError:
    print("Virheellinen luku")
else:
    print("Annoit luvun", luku)
```

try-except-else-rakenteeseen voi yhdistää vielä finally-osan, jossa esimerkiksi suljetaan avatut tiedostot.

Lista Pythonin poikkeuksista

Mistä tietää, mikä poikkeus pitäisi napata? Tässä on listä Pythonin sisäänrakennetuista poikkeuksista. Tällä kurssilla tärkeimmät poikkeukset ovat OSError ja ValueError. Monimutkaisemmissa ohjelmissa pitää ottaa huomioon erilaisten kirjastojen poikkeustilanteet. Oikeassa ohjelmistossa, jonka tehtävä on esimerkiksi valvoa kriittistä tuotantoprosessia, voikin olla enemmän virheenkäsittelykoodia kuin varsinaista toiminnallista koodia!

Kierros 6

Kuudes ja viimeinen kierros sisältää uutena asiana **Scipy**-kirjaston, jossa on valtava määrä työkaluja tieteellistä laskentaa varten.

Kierroksen oppimateriaalissä käsitellään myös **olio-ohjelmointia**. Tämä on lisämateriaalia ja kierroksen B-tehtävät käsittelevät tätä aihepiiriä. Olio-ohjelmointi on hyvin tärkeä lähestymistapa modernissa ohjelmistotuotannossa, mutta lyhyellä peruskurssilla ehdimme käydä sitä läpi vain pintaraapaisun verran. Syvällisemmin aiheeseen pääsee perehtymään esimerkiksi kurssilla CS-A1121 - Ohjelmoinnin peruskurssi Y2.

SciPy

SciPy on Pythonilla luotu tieteellisen laskennan infrastruktuuri, joka on vapaasti kaikken Python-ohjelmoijien käytettävissä. SciPy on laaja kokonaisuus ja olemmekin jo käyttäneet osia siitä:

- NumPy-kirjasto on osa SciPyä ja SciPyn eri toiminnot hyödyntävät hyvin paljon NumPy-taulukoita
- Matplotlib-kirjasto on osa SciPyä
- Jopa Spyderin interaktiivinen IPython-konsoli on osa Scipyä

SciPyn dokumentaatio löytyy osoitteesta https://docs.scipy.org/doc/scipy/reference/ ja samasta paikasta löytyy myös tutoriaali SciPyn erilaisista alamoduuleista. Alamoduuleja on toistakymmentä ja tällä kurssilla tutustumme niistä vain neljään:

- scipy.constants: Luonnonvakiot. Sanakirjan scipy.constants.physical_constants tietolähde on CODATA-tietokanta, jota tälläkin kurssilla olemme hyödyntäneet. Helppo tapa ottaa luonnonvakioiden tarkimmat tunnetut arvot käyttöön!
- scipy.stats: Tilastolliset työkalut, esimerkiksi lineaarinen regressio (scipy.stats.linregress)
- scipy.linalg: Lineaarialgebra, esimerkiksi matriisien käsittely ja yhtälönryhmän ratkaiseminen (scipy.linalg.solve)
- scipy.integrate: Integrointi, erityisesti differentiaaliyhtälöiden numeerinen integrointi (scipy.integrate.odeint)

Oppimateriaalin seuraavissa kappaleissa annetaan käytännön esimerkkejä näiden alamoduulien hyödyntämisestä.

scipy.constants

Moduuli *scipy.constants* sisältää luonnonvakioita, joista yleisimmät voi ottaa käyttöön suoraan tuomalla pelkän *scipy.constants*-moduulin ohjelmaan:

```
import scipy.constants
print("Kaasuvakion R arvo on {:.7f} J K^-1 mol^-1".format(scipy.constants.R))
```

tulostaa

```
Kaasuvakion R arvo on 8.3144598 J K^-1 mol^-1
```

physical_constants-sanakirja

scipy.constants sisältää myös sanakirjan *scipy.constants.physical_constants*, jonka muoto on:

```
physical_constants[name] = (arvo_liukulukuna, yksikkö_merkkijonona, epävarmuus_liukulukuna)
```

Sanakirjan avain on siis luonnonvakio ja arvo on kolmen alkion monikko (voit ajatella sitä listana). Sanakirja sisältää laajan valikoiman luonnonvakioita, joiden arvot tulevat CODATA-tietokannasta. Esimerkki sanakirjan käytöstä:

```
from scipy.constants import physical_constants as pc
R = pc["molar gas constant"][0]
R_yksikko = pc["molar gas constant"][1]
R_epavarmuus = pc["molar gas constant"][2]
print("Kaasuvakion R arvo on {:.7f} {:s}".format(R, R_yksikko))
print("Arvon epävarmuus on {:.7f} {:s}".format(R_epavarmuus, R_yksikko))
```

tulostaa

```
Kaasuvakion R arvo on 8.3144598 J mol^-1 K^-1
Arvon epävarmuus on 0.0000048 J mol^-1 K^-1
```

Yksikkömuunnokset

Moduuli sisältää myös arvoja yksikkömuunnoksia varten:

```
import scipy.constants

kcal_mol = float(input("Anna energia yksikoissä kcal/mol:\n"))
kJ_mol = kcal_mol * scipy.constants.calorie
print("Antamasi energia on SI-yksiköissä {:.3f} kJ/mol".format(kJ_mol))
```

Tulostaa

```
Anna energia yksikoissä kcal/mol:
> 2.5
Antamasi energia on SI-yksiköissä 10.460 kJ/mol
```

scipy.stats

scipy.stats-moduuli sisältää valtavan määrän erilaisia tilastollisia funktioita ja jakaumafunktioita. Tämän kurssin puitteissa tutustumme vain *scipy.stats.linregress*-funktioon, jolla voi tehdä lineaarisen regressioanalyysin esimerkiksi mittausdatoille. Käytännössähän kyse on samasta suoran yhtälön sovituksesta, mitä olemme jo tehneet numpy.polyfit-funktion avulla 1. asteen polynomeille. *linregress*-funktio on kuitenkin suunniteltu juuri lineaariseen regressioon ja se myös palauttaa enemmän tietoja sovituksesta. Funktio palauttaa esimerkiksi korrelaatiokertoimen ja keskivirheen, emmekä tarvitse *np.corrcoef*-funktiota. Lisäksi *linregress* on laskennallisesti tehokkaampi hyvin suurille datajoukoille.

Tutustutaan *linregress*-funktioon esimerkin avulla. Meillä on käytössämme tiedosto T_p_data.txt, jossa on esitetty paine lämpötilan funktiona kaasumaiselle yhdisteelle (n = 0.65 mol). Mittausolosuhteet ovat sellaiset, että voimme odottaa ideaalikaasulain olevan voimassa.Tehtävänä on ratkaista astian tilavuus V.

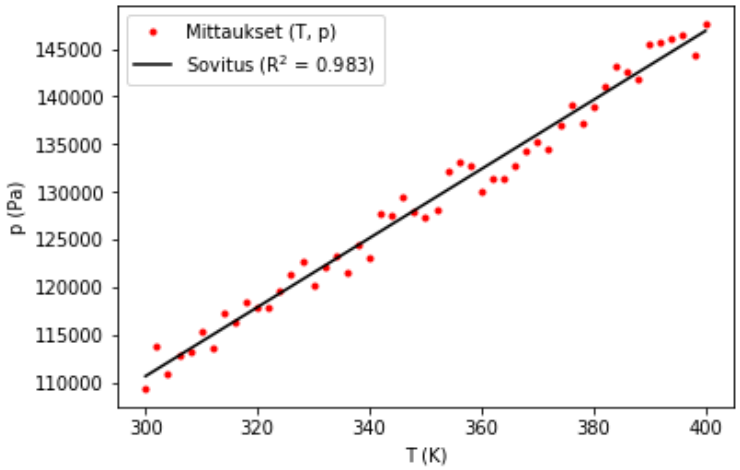
- $pV = nRT$, joten $p = nRT / V$
- Kun paine esitetään lämpötilan funktiona ja mittausdatat sovitetaan suoran yhtälöön, sovitussuoran kulmakerroin on nR/V. Eli $V = nR / kulmakerroin$.
- Luetaan siis datat tiedostosta, tehdään niille lineaarinen regressio *linregress*-funktiolla ja ratkaistaan tilavuus V.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress
from scipy.constants import R

n = 0.65 # mol
try:
    p_T_datat = np.loadtxt("T_p_data.txt")
except OSError:
    print("Tiedoston p_T_data.txt avaaminen epaonnistui")
else:
    T = p_T_datat[:, 0]
    p = p_T_datat[:, 1]

    # Lineaarinen regressio. T == x, p == y
    slope, intercept, r_value, p_value, std_err = linregress(T, p)

    # Sovitussuoran yhtälö: y = slope * x + intercept
    # Lasketaan sovitussuoran arvot mitatuille x-arvoille (taulukko T)
    p_sovitetut = slope * T + intercept

    # Piirretään mittausdatat ja sovitussuora
    plt.plot(T, p, '.', color = 'red', label = 'Mittaukset (T, p)')
    # r_value on korrelaatiokerroin
    teksti = "Sovitus (R$^2$ = {:.3f})".format(r_value**2)
    plt.plot(T, p_sovitetut, '-', color = 'black', label = teksti)

    plt.xlabel('T (K)')
    plt.ylabel('p (Pa)')
    plt.legend(loc = 'upper left')

    # Ratkaistaan V == n * R / slope
    # Tulostetaan V ja kulmakertoimen keskivirhe std_err
    print("V: {:.3f} m^3".format(n * R / slope))
    print("Kulmakertoimen keskivirhe: {:.1f}".format(std_err))
```

Koodi tulostaa

```
V: 0.015 m^3
Kulmakertoimen keskivirhe: 6.9
```

ja piirtää allaolevan kuvaajan


Mittaukset (T, p)
Sovitus (R^2 = 0.983)
p (Pa)
T (K)
145000
140000
135000
130000
125000
120000
115000
110000
300
320
340
360
380
400

scipy.linalg (Lineaarialgebra)

Moduuli scipy.linalg sisältää suuren määrän lineaarialgebraan liittyviä työkaluja (esim. matriisioperaatiot, ominaisarvo-ongelmien ratkaiseminen). Kappaleessa NumPyn matemaattiset funktiot mainittiin aiemmin moduuli numpy.linalg, joka sisältää samoja työkaluja. SciPyn lineaarialgebramoduuli on huomattavasti NumPy-moduulia laajempi.

Tämän kurssin puitteissa tutustumme vain funktioon *scipy.linalg.solve*, jolla voi ratkaista lineaarisia yhtälöryhmiä.

Hieman teoriaa

Tutustutaan yhtälöryhmien ratkaisemiseen Wikipedian sisältämän esimerkin avulla.

Määritellään lineaarinen kolmen yhtälön yhtälöryhmä:

$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + \tfrac{1}{2}y - z &= 0 \end{aligned}$$

Yhtälöryhmässä on siis kolme (tuntematonta) muuttujaa *x*, *y* ja *z*. Ratkaistaan muuttujien arvot muuntamalla yhtälöryhmä matriisiyhtälöksi ja soveltamalla *scipy.linalg.solve*-funktiota.

Yhtälöryhmä, jossa on *m* kappaletta yhtälöitä ja *n* kappaletta muuttujia voidaan kirjoittaa yleisessä muodossa

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m, \end{aligned}$$

missä x_1, x_2, ..., x_n ovat yhtälöryhmän tuntemattomat muuttujat,
a_{11}, a_{12}, ..., a_{mn} ovat yhtälöryhmän kertoimet ja
b_1, b_2, ..., b_m ovat yhtälöryhmän vakiotermit.

Yhtälöryhmän ylläoleva yleinen muoto voidaan kirjoittaa myös vektorimuodossa

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

missä kertoimia ja vakiotermejä kuvataan sarakevektoreilla. Tämä vektorimuoto taas voidaan kirjoittaa matriisiyhtälönä

$$A\mathbf{x} = \mathbf{b}$$

missä

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

A on siis neliömatriisi, **x** ja **b** vektoreita. Nyt kun yhtälöryhmä on saatu matriisimuotoon, sen ratkaisemisessa voidaan hyödyntää lineaarialgebraa. Emme käsittele teoriaa tämän enempää vaan toteamme vain, että kun meillä on yhtälöryhmä matriisimuodossa ***A*****x** = **b**, voimme ratkaista sen *scipy.linalg.solve-funktiolla* näin helposti:

```
x = scipy.linalg.solve(A, b)
```

Esimerkki

Käytetään esimerkissä yllä esiteltyä yhtälöryhmää

$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + \tfrac{1}{2}y - z &= 0 \end{aligned}$$

Nyt siis muuttujat x, y ja z vastaavat yhtälöryhmän yleisen muodon muuttujia x_1, x_2 ja x_3. Ratkaistaan tuntemattomat:

```
import numpy as np
from scipy.linalg import solve

# A on 3x3 neliömatriisi
A = np.array([[ 3,   2, -1],
              [ 2,  -2,  4],
              [-1, 1/2, -1]])

# b on kolmen alkion vektori
b = np.array([1, -2, 0])

# Ratkaistaan tuntemattomat muuttujat
x = solve(A, b)
print(x)
```

Koodi tulostaa

```
[ 1. -2. -2.]
```

Eli
$x_1 = x = 1$,
$x_2 = y = -2$ ja
$x_3 = z = -2$

Ratkaisu on täsmälleen sama kuin Wikipediassa:

$$\begin{aligned} x &= 1 \\ y &= -2 \\ z &= -2 \end{aligned}$$

scipy.integrate.odeint

Kemian tekniikassa haluamme usein ratkaista (integroida) differentiaaliyhtälöitä numeerisesti. SciPyn integrate-alamoduuli sisältää useita funktioita tätä varten. Tällä kurssilla tutustutaan scipy.integrate.odeint-funktioon.

Otetaan esimerkkinä klassinen esimerkki, eli bakteeripopulaation eksponentiaalinen kasvu. Esimerkkisysteemimme osalta tiedetään, että bakteeripopulaation kasvunopeus *dy*/*dt* on suoraan verrannollinen populaation kokoon *y* ajan hetkellä *t*:

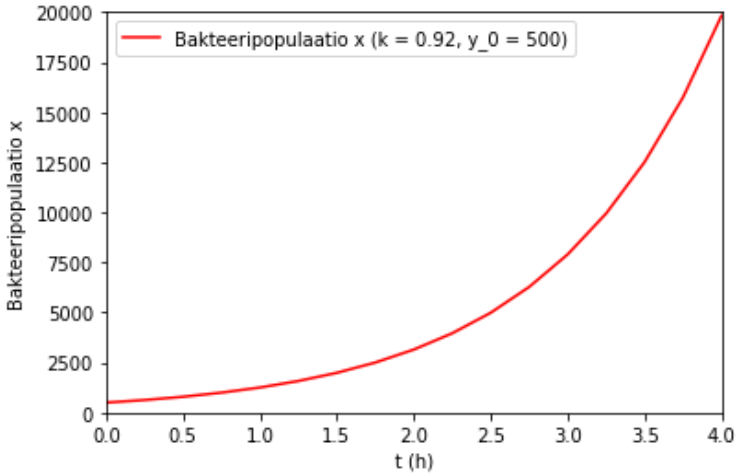
```
dy/dt = k * y
```

Tässä tapauksessa differentiaaliyhtälö on itse asiassa hyvin helppo ratkaista analyyttisesti suoralla integroinnilla (ks. Wikipedia-sivu). Mutta havainnollistetaan tämän suoraviivaisen esimerkin avulla, kuinka differentiaaliyhtälön numeerinen integrointi onnistuu SciPyllä.

Aja alla oleva esimerkki Spyderissä. Huomaa, miten *dy*/*dt* on määritelty funktiona *f_dy_dt* ja miten tämä funktio annetaan *odeint*-funktion parametriksi. Lisäksi *odeint* saa parametrinä muuttujan *y* alkuarvon *y_0* ja tutkittavat ajanhetket taulukossa *t*. Käytännössä siis *odeint* siis kutsuu funktiota *f_dy_dt* kaikilla ajan hetkillä *t* ja antaa parametrinä myös senhetkisen populaation *y*.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate

k = 0.92  # kasvunopeus
y_0 = 500 # bakteeripopulaatio alussa

def f_dy_dt(y, tx):
    # Differentiaaliyhtälön määritelmä eksponentiaaliselle kasvulle
    # Parametri y on populaatio ajan hetkellä t
    # Huomaa, että aikaparametria tx ei tarvita tässä tapauksessa, mutta
    # funktion määritelmän täytyy sisältää se, jotta odeint hyväksyy funktion.
    dy_dt = k * y
    return dy_dt

# Simuloidaan ajanhetket t = [0, 4] h
max_t = 4 # tuntia
t = np.linspace(0, max_t, max_t * 4 + 1) # pisteet 0.25 h välein

# Differentiaaliyhtälön dy_dt numeerinen integrointi
# y_0 -> populaation alkuarvo
# t  -> ajanhetket, joissa populaation määrä lasketaan funktion f_dy_dt avulla
# paluuarvo "pop" on taulukko. np.shape(pop) on (len(t), len(y_0)), eli (17,1)
pop = scipy.integrate.odeint(f_dy_dt, y_0, t)

# Piirretaan kuvaaja
teksti = "Bakteeripopulaatio x (k = {}, y_0 = {})".format(k, y_0)
plt.plot(t, pop[:, 0], color = 'red', label = teksti)
plt.xlim(0, max_t)
plt.ylim(0.0, 20000)
plt.xlabel('t (h)')
plt.ylabel('Bakteeripopulaatio x')
plt.legend()
plt.show()
```

Lopputuloksena on seuraavan näköinen kuvaaja:



Huomaa, että *odeint*-funktion parametri *y_0* voi olla myös vektori. Tällöin myös derivaatat laskeva funktio saa parametrikseen vektorin *y* ja *odeint* palauttaa taulukon, jossa on yhtä monta saraketta kuin vektorissa y_0 on alkioita.

Olio-ohjelmointia 1

Aiemmin tällä kurssilla olemme tutustuneet erilaisiin **tietorakenteisiin** (listat, sanakirjat, monikot, NumPy-taulukot) ja **funktioihin**, joilla näitä tietorakenteita voi käsitellä (esim. *max*(lista)).

Tutustutaan lopuksi lyhyesti **olioihin** (*engl.* object). Oliot ovat periaatteessa tietorakenteita, jotka sisältävät myös tietojen käsittelyyn tarkoitettuja funktioita.

Luokan määritteleminen ja olioiden luominen

Jotta voimme luoda uuden olion, meidän pitää ensin määritellä luokka (engl. *class*) joka kuvaa olion ominaisuudet. Määritellään luokka *Molekyyli* (luokkien nimet kirjoitetaan isolla alkukirjaimella):

```
class Molekyyli:
    def __init__(self, kaava, moolimassa):
        self.kaava = kaava
        self.moolimassa = moolimassa

    def laske_ainemaara(self, massa):
        return massa / self.moolimassa
```

Molekyyli-luokka sisältää kaksi funktiomäärittelyä. Näitä funktioita kutsutaan **metodeiksi** (engl. method) erotuksena tavallisista funktioista, jotka eivät kuulu mihinkään luokkaan.

Molekyyli-luokka sisältää **käynnistysmetodin** *__init__* ja metodin *laske_ainemaara*. Huomaa, että molempien metodien ensimmäinen parametri on **self**. Tämä parametri viittaa aina olioon itseensä. Python hoitaa *self*-parametrin automaattisesti, eli sitä ei anneta, kun metodia kutsutaan.

Käynnistysmetodissa *__init__* luodaan oliolle kaksi **kenttää**: *kaava* ja *moolimassa*. Kenttiin pitää viitata metodin *self*-parametrin avulla.

Katsotaan, mitä määrittelemällämme luokalla voidaan nyt tehdä. Luodaan Molekyyli-luokkaan pohjautuvat oliot *metaani* ja *etaani*:

```
metaani = Molekyyli("CH4", 16.04)
etaani = Molekyyli("C2H6", 30.07)
```

Käsky *Molekyyli*("CH4", 16.04) tarkoittaa, että *Molekyyli*-luokan *__init__*-metodia kutsutaan parametreilla "CH4" ja 16.04 (self-parametria ei anneta, mutta Python antaa sen __init__-metodille automaattisesti). Käsky palauttaa uuden olion, jonka kentät *kaava* ja *moolimassa* on täytetty arvoilla "CH4" ja 16.04.

Kokonainen olioesimerkki

Luokkamäärittelyn pohjalta voi siis luoda mielivaltaisen määrän uusia olioita. Katsotaan kokonaisen esimerkin avulla, miten olioiden kenttiä voi lukea ja miten niiden metodeja käytetään:

```
class Molekyyli:
    def __init__(self, kaava, moolimassa):
        self.kaava = kaava
        self.moolimassa = moolimassa

    def laske_ainemaara(self, massa):
        return massa / self.moolimassa

metaani = Molekyyli("CH4", 16.04)
etaani = Molekyyli("C2H6", 30.07)

# Käytetään olioiden kenttiä
print("Metaanin molekyylikaava on", metaani.kaava)
print("Etaanin molekyylikaava on", etaani.kaava)
print("Metaanin moolimassa on", metaani.moolimassa, "g/mol")
print("Etaanin moolimassa on", etaani.moolimassa, "g/mol")

# Käytetään laske_ainamaara-metodia.
# Huomaa, että self-parametria ei anneta
n_metaani = metaani.laske_ainemaara(5.0) # 5 g metaania
n_etaani = etaani.laske_ainemaara(7.0)   # 7 g etaania
print("5 g metaania on", round(n_metaani, 3), "mol")
print("7 g etaania on", round(n_etaani, 3), "mol")
```

tulostaa

```
Metaanin molekyylikaava on CH4
Etaanin molekyylikaava on C2H6
Metaanin moolimassa on 16.04 g/mol
Etaanin moolimassa on 30.07 g/mol
5 g metaania on 0.312 mol
7 g metaania on 0.233 mol
```

Huomaa, miten *laske_ainemaara*-metodissa self.moolimassa viittaa kunkin olion omaan moolimassa-kenttään. Sillä on siis eri arvo metaanille ja etaanille. Näin ainemäärä lasketaan oikein kullekin oliolle. Parametri massa taas määritetään aina metodia kutsuttaessa.

Kannattaa tutustua esimerkkiin huolella. Esimerkki on yksinkertainen, mutta sen tarkoituksena on havainnollistaa, miten olioiden avulla voidaan yhdistää tietorakenteet ja funktiot samaan pakettiin. *self*-parametrin käyttö on olio-ohjelmoinnin avainkäsitteitä.

Yllättävä käänne

Olemme itse asiassa käyttäneet olioita aivan koko kurssin ajan! Pythonissa oikeastaan **kaikki** asiat ovat olioita. Esimerkiksi *int* ja *float* -tyyppiset muuttujat tai *list* ja *dict* -tietorakenteet ovat kaikki olioita, jotka sisältävät myös metodeja kyseisten olioiden käsittelemiseksi:

```
# float-olio sisältää esimerkiksi is_integer()-metodin
liukuluku = 3.14
print(liukuluku.is_integer())
liukuluku_int = 3.0
print(liukuluku_int.is_integer())
```

tulostaa

```
False
True
```

list-tietorakenne sisältävää useita metodeja, joita olemmekin jo käyttäneet

```
lista = [1, 2, 3]
print(lista.count(1))
lista.append(1)
print(lista.count(1))
```

tulostaa

```
1
2
```

Olio-ohjelmointia 2

Luokkamuuttujat

Edellisen luvun esimerkissä *Molekyyli*-luokalla oli kaksi kenttää, *kaava* ja *moolimassa*. Jokaisella *Molekyyli*-luokan pohjalta luodulla oliolla on omat arvonsa näissä kentissä. Joskus voi olla kuitenkin tarpeen säilyttää tietoa, joka on kaikille luokan olioille yhteistä. Silloin voidaan hyödyntää **luokkamuuttujia**.

Lisätään *Molekyyli*-luokkaan luokkamuuttuja *maara* (määrä), jolla pidetään kirjaa *Molekyyli*-luokkaan perustuvien olioiden määrästä:

```
class Molekyyli:
    maara = 0
    def __init__(self, kaava, moolimassa):
        Molekyyli.maara += 1
        self.kaava = kaava
        self.moolimassa = moolimassa

    def laske_ainemaara(self, massa):
        return massa / self.moolimassa

metaani = Molekyyli("CH4", 16.04)
etaani = Molekyyli("C2H6", 30.07)
propaani = Molekyyli("C3H8", 44.10)

print("Olet luonut {} molekyyliä".format(Molekyyli.maara))
```

tulostaa

```
Olet luonut 3 molekyyliä
```

Luokkamuuttuja *maara* määritellään siis luokan metodien ulkopuolella. Se saa arvon 0, kun ohjelma käynnistyy. Kun luokan pohjalta luodaan uusi olio, käynnistysmetodi kasvattaa luokkamuuttujan arvoa yhdellä. Huomaa, että luokkamuuttujaan on viitattava luokan nimen avulla (*Molekyyli.maara*) sekä luokan metodien sisällä että luokkamäärittelyn ulkopuolella.

Olioiden säilöminen tietorakenteisiin

Oliot ovat jo itsessään kätevä tapa tietojen säilömiseksi. Meno muuttuu kuitenkin vielä jännittävämmäksi kun olioita aletaan tunkea tietorakenteisiin:

```
class Molekyyli:
    maara = 0
    def __init__(self, kaava, moolimassa):
        Molekyyli.maara += 1
        self.kaava = kaava
        self.moolimassa = moolimassa

    def laske_ainemaara(self, massa):
        return massa / self.moolimassa

metaani = Molekyyli("CH4", 16.04)
etaani = Molekyyli("C2H6", 30.07)
propaani = Molekyyli("C3H8", 44.10)
butaani = Molekyyli("C4H10", 58.12)

hiilivedyt_lista = [metaani, etaani, propaani, butaani]

print("Olet luonut {} molekyyliä:".format(Molekyyli.maara))
for alkio in hiilivedyt_lista:
    print(alkio.kaava)
```

tulostaa

```
Olet luonut 4 molekyyliä:
CH4
C2H6
C3H8
C4H10
```

Lista olioita on erittäin kätevä tapa korvata aiemmin kurssilla käytetyt rakenteet, joissa listojen sisään on tungettu toisia listoja. Olioiden avulla tiedot on helpompi säilöä ja niihin on helpompi päästä käsiksi.

Olio-ohjelmointia 3

Viimeisenä olioesimerkkinä on luokka *Alkuaine*. Luokalla on käynnistysmetodin lisäksi kolme metodia *on_kiintea*, *on_neste* ja *on_kaasu*, joilla voi helposti tarkastella alkuaineen olomuotoa tietyssä lämpötilassa. Lisäksi luokalla on erikoismetodi *__str__*, jonka tarkoitus on palauttaa luokan oliota kuvaava merkkijono. Tämä merkkijono tulostuu esimerkiksi jos *print*-funktiolle annetaan parametriksi luokan olio.

```
class Alkuaine:
    def __init__(self, Z, symboli, nimi, atomipaino,
                 sulamispiste, kiehumispiste):
        self.Z = Z
        self.symboli = symboli
        self.nimi = nimi
        self.atomipaino = atomipaino
        self.sulamispiste = sulamispiste   # K
        self.kiehumispiste = kiehumispiste # K

    def __str__(self):
        return("{:s} ({:s}): atomipaino = {:.2f}"
               .format(self.symboli, self.nimi, self.atomipaino))
```

```
    def on_kiintea(self, T):
        # Palauttaa True, jos alkuaine on kiinteä lampotilassa T (K)
        return T < self.sulamispiste

    def on_neste(self, T):
        # Palauttaa True, jos alkuaine on neste lampotilassa T (K)
        return T > self.sulamispiste and T < self.kiehumispiste

    def on_kaasu(self, T):
        # Palauttaa True, jos alkuaine on kaasu lampotilassa T (K)
        return T > self.kiehumispiste

sinkki = Alkuaine(30, 'Zn', 'sinkki', 65.38, 693, 1180)
kadmium = Alkuaine(48, 'Cd', 'kadmium', 112.411, 594, 1040)
elohopea = Alkuaine(80, 'Hg', 'elohopea', 200.59, 234, 630)

T = 600 # K
for metalli in [sinkki, kadmium, elohopea]:
    if metalli.on_neste(T):
        print("{} on neste lampotilassa {} K".format(metalli.symboli, T))
    else:
        print("{} ei ole neste lampotilassa {} K".format(metalli.symboli, T))

# Käytetään __str__ -metodia kutsumalla print-funktiota
print("---------------------------------")
print(sinkki)
```

tulostaa

```
Zn ei ole neste lämpotilassa 600 K
Cd on neste lämpotilassa 600 K
Hg on neste lämpotilassa 600 K
---------------------------------
Zn (sinkki): atomipaino = 65.38
```

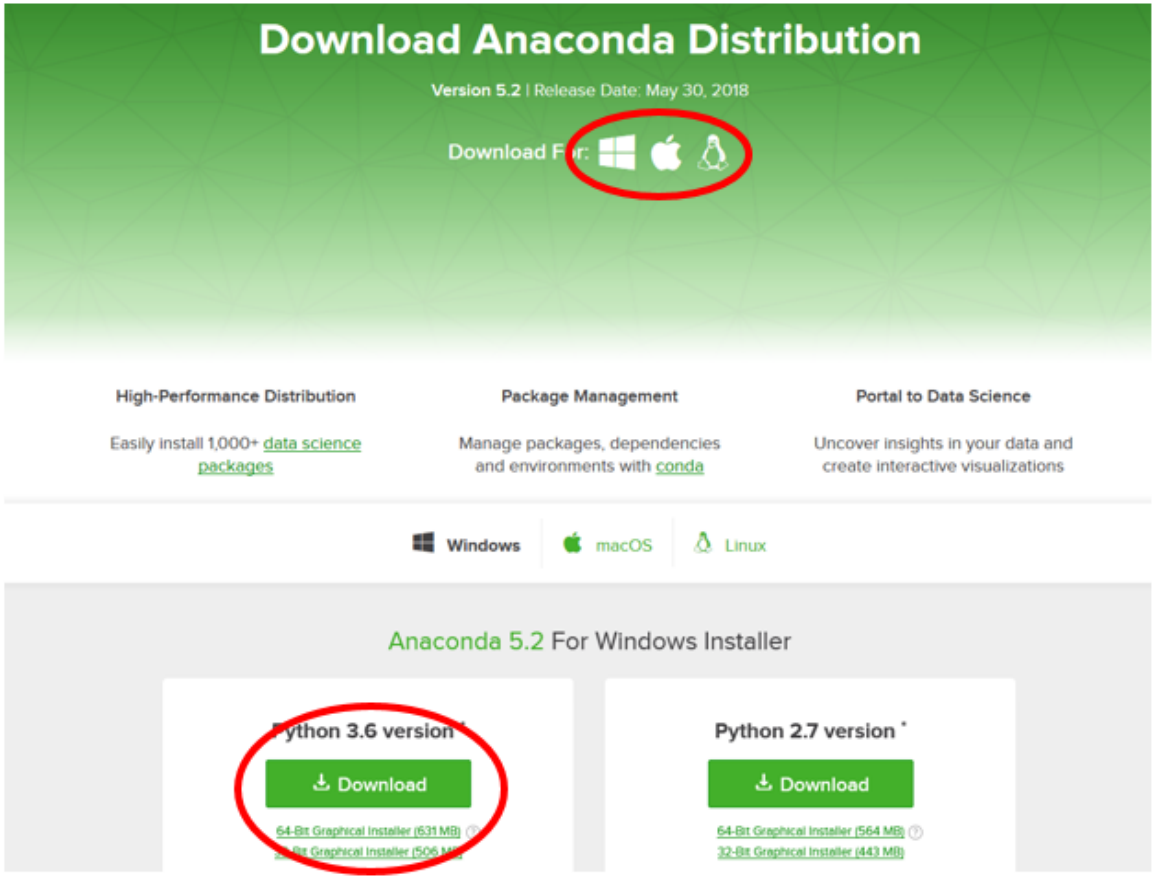
Lisämateriaalia

Tämä kappale sisältää yleistä lisämateriaalia Python-ohjelmointiin liittyvistä aiheista.

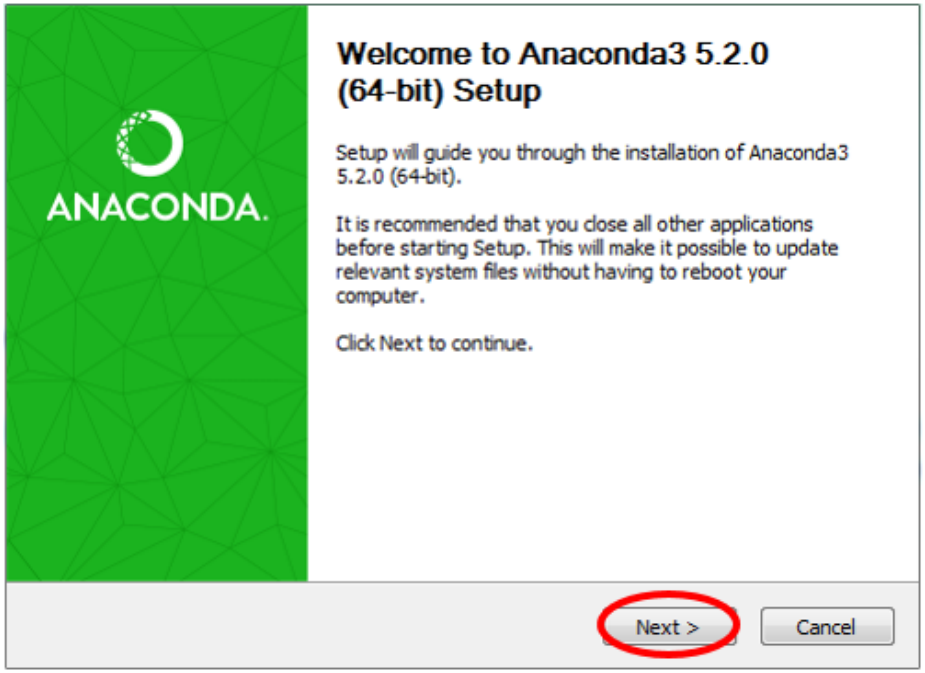
Anacondan asennusohje

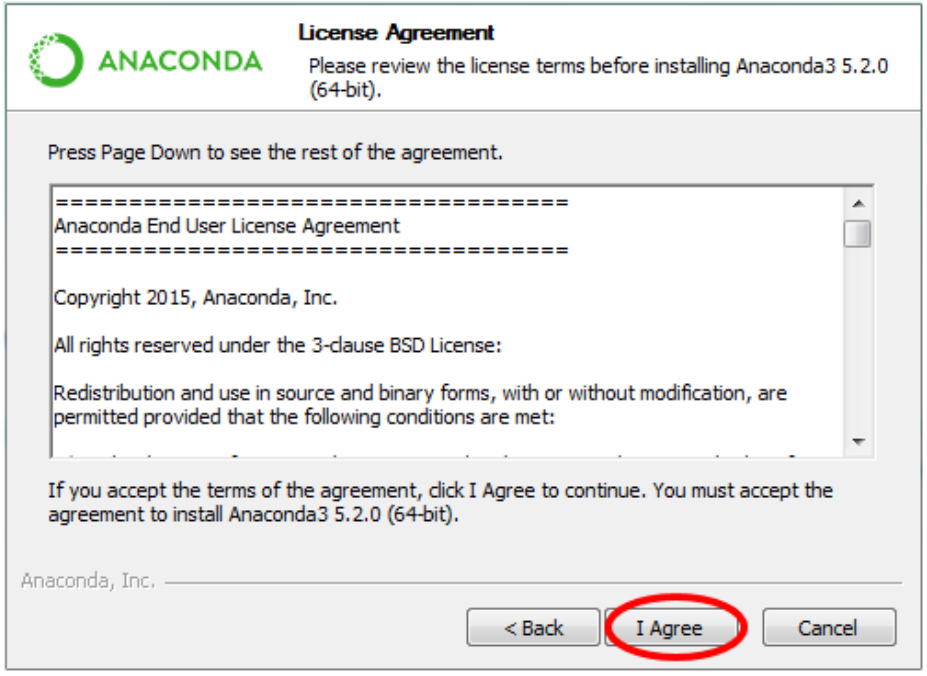
Aloita menemällä sivulle https://www.anaconda.com/download/

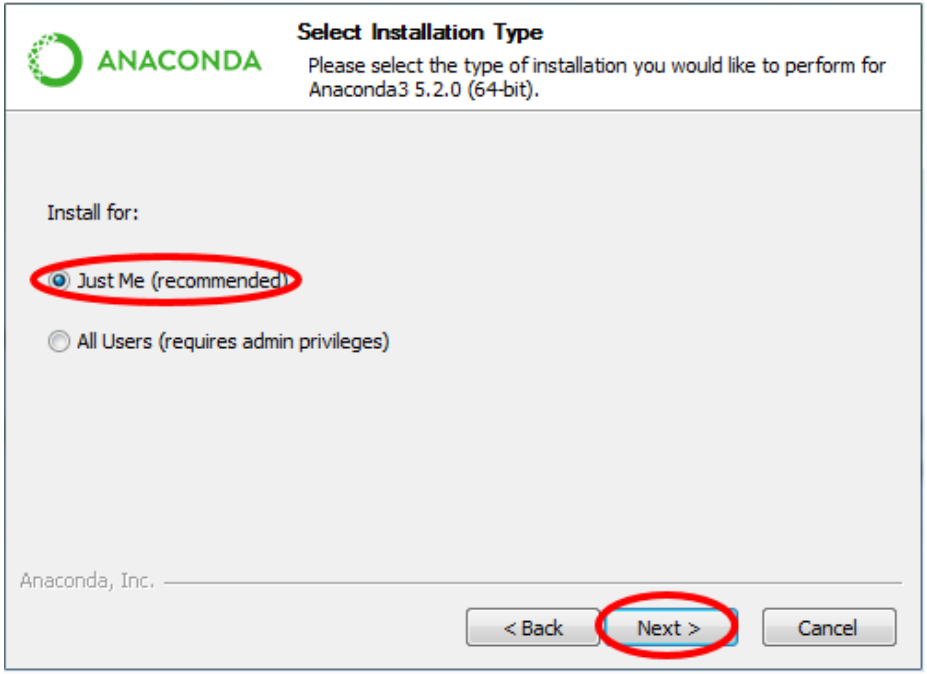
Valitse käyttöjärjestelmä, jota käytät ja lataa "**Python 3.7 version**" (esimerkkikuvassa näkyy vanha versio 3.6).



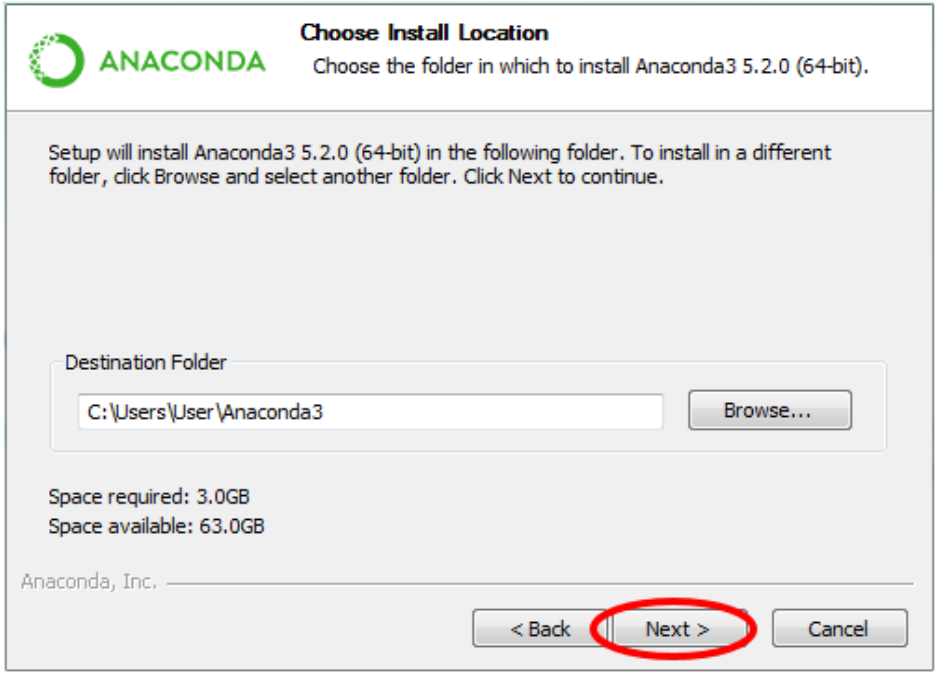
Kun tiedosto on latautunut, käynnistä asennusohjelma (esim. Anaconda3-5.2.0-Windows-x86_64.exe) ja seuraa asennusohjetta.

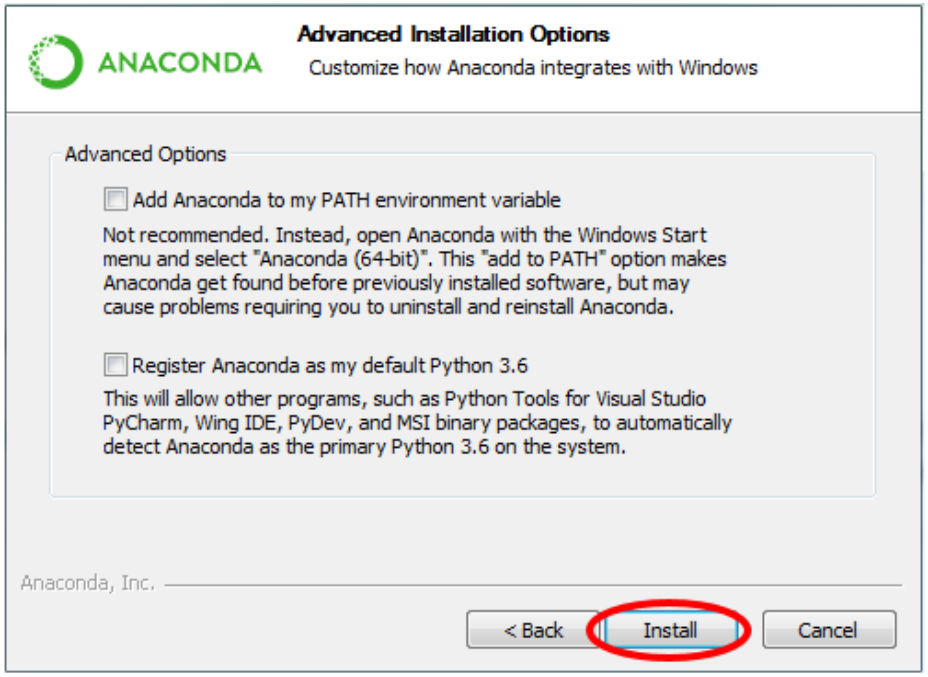

ANACONDA.
Welcome to Anaconda3 5.2.0 (64-bit) Setup
Setup will guide you through the installation of Anaconda3 5.2.0 (64-bit).
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Next >
Cancel

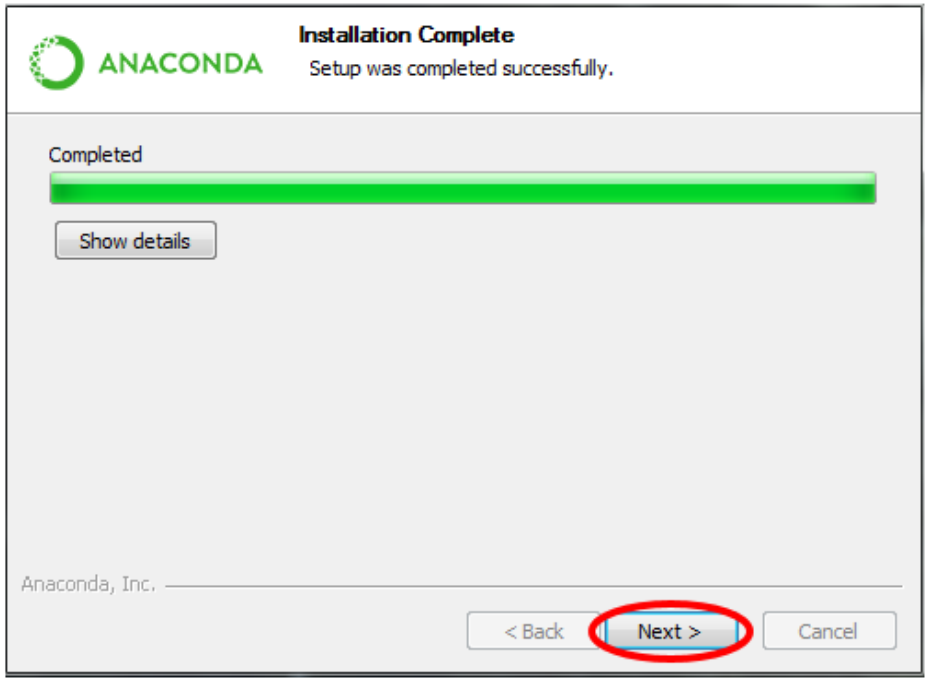

ANACONDA
License Agreement
Please review the license terms before installing Anaconda3 5.2.0 (64-bit).
Press Page Down to see the rest of the agreement.
Anaconda End User License Agreement
Copyright 2015, Anaconda, Inc.
All rights reserved under the 3-clause BSD License:
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 5.2.0 (64-bit).
Anaconda, Inc.
< Back
I Agree
Cancel

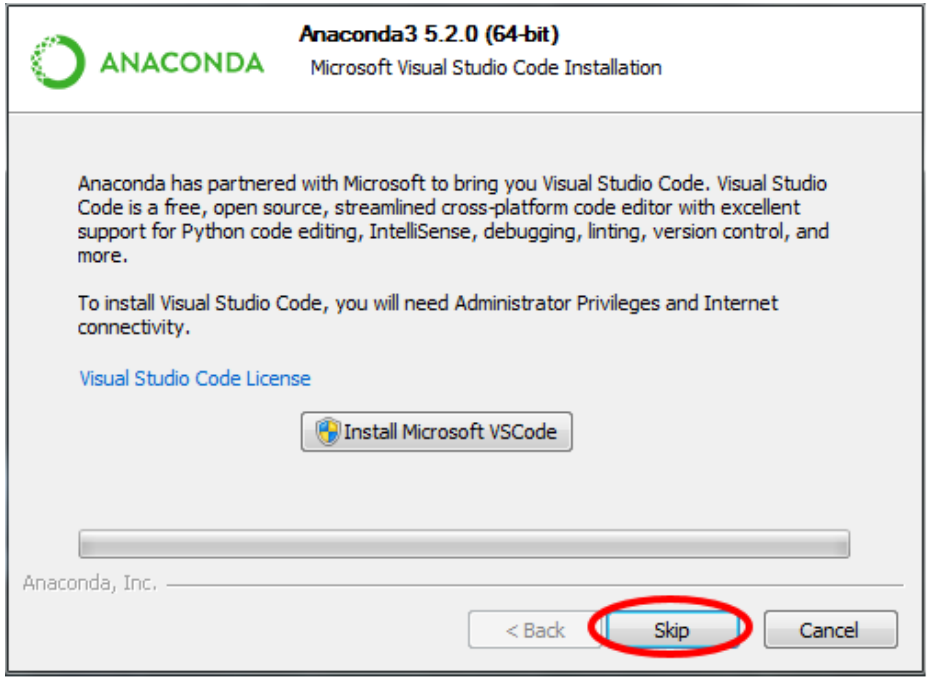


*** HUOM! TÄRKEÄÄ TIETOA ASENNUSKANSIOTA KOSKIEN*** Kun valitset sijainnin, jonne Anaconda asennetaan, pidä huoli, että polussa **ei ole välilyöntejä** (ks. kuva alla). Windows 10 -koneilla hyvä asennuskansio on esimerkiksi C:\Users\<käyttäjätunnus>\Anaconda3

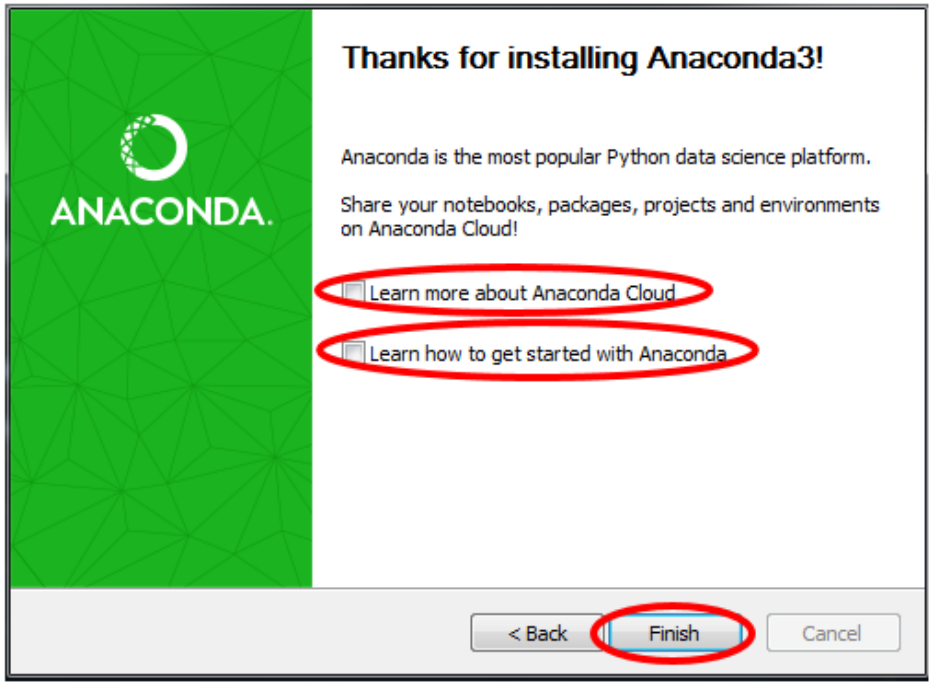

ANACONDA
Choose Install Location
Choose the folder in which to install Anaconda3 5.2.0 (64-bit).
Setup will install Anaconda3 5.2.0 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
C:\Users\User\Anaconda3
Browse...
Space required: 3.0GB
Space available: 63.0GB
Anaconda, Inc.
< Back
Next >
Cancel


ANACONDA
Advanced Installation Options
Customize how Anaconda integrates with Windows
Advanced Options
Add Anaconda to my PATH environment variable
Not recommended. Instead, open Anaconda with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.
Register Anaconda as my default Python 3.6
This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.6 on the system.
Anaconda, Inc.
< Back
Install
Cancel


ANACONDA
Installation Complete
Setup was completed successfully.
Completed
Show details
Anaconda, Inc.
< Back
Next >
Cancel


ANACONDA
Anaconda3 5.2.0 (64-bit)
Microsoft Visual Studio Code Installation
Anaconda has partnered with Microsoft to bring you Visual Studio Code. Visual Studio Code is a free, open source, streamlined cross-platform code editor with excellent support for Python code editing, IntelliSense, debugging, linting, version control, and more.
To install Visual Studio Code, you will need Administrator Privileges and Internet connectivity.
Visual Studio Code License
Install Microsoft VSCode
Anaconda, Inc.
< Back
Skip
Cancel

Viimeisessä asennusruudussa poista valinta kahdesta vapaaehtoisesta valintaruudusta, klikkaa ”Finish” nappia ja Anaconda on asennettu.



Spyder-kehitysympäristön saat avattua esimerkiksi kirjoittamalla **spyder** Windowsin aloitusvalikkoon ja avaamalla sovelluksen.

Virheiden etsiminen ja korjaaminen

Virheiden löytäminen ja käsittely kuuluu jokaisen ohjelmoijan perustaitoihin ja on välttämätöntä suuria ohjelmia kirjoittaessa. Tällä kurssilla ei luoda ohjelmia, jotka vaativat huomattavaa virheiden käsittelyä tai debuggaamista. Edettäessä suurempiin ohjelmiin, erityisesti graafisen käyttöliittymän sisältäviin ohjelmiin ja sekä vaativampiin (”pikkutarkkoihin”) kieliin (C, C++), on virheiden käsittely ja debuggaaminen erittäin oleellista.

Virheen löytäminen alkaa **traceback**-viestistä. Traceback on punainen virheilmoitus, joka kertoo syyn ohjelman kaatumiseen. Aluksi viesti voi näyttää heprealta, mutta kun sitä oppii lukemaan, se on erittäin hyödyllinen apuväline virheiden löytämisessä.

Esimerkki 1

Katsotaan ensin yksinkertaista tracebackiä, joka syntyy koodista

```
print(x)
```

Traceback on:

```
Traceback (most recent call last):
File "C:/Users/Omistaja/Desktop/ErrorExample1.py", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

Tracebackin ensimmäinen rivi ilmoittaa meille virheen sijainnin. Tässä tapauksessa virhe tapahtui tiedostossa *ErrorExample1.py* rivillä 1. Huomaa miten virheviestissä lukee polku, jossa tiedosto sijaitsee, pelkän nimen sijaan.

Seuraava rivi kertoo meille, mitä kyseisellä rivillä lukee. Tässä tapauksessa koodin pätkä, mikä aiheuttaa virheen on *print(x)*.

Viimeinen rivi tracebackissä kertoo meille mikä virhe on kyseessä. Kyseinen virhe on siis *NameError*, joka johtuu siitä, että muuttujaa x ei ole määritelty.

Esimerkki 2

Entäs sitten hieman monimutkaisempi traceback.

Traceback viesti on syntynyt seuraavasta koodista:

```
def palautaAlkio(lista, alkio):
    return lista[alkio]
lista = [1, 2, 3]
print(palautaAlkio(lista,3))
```

Selkeyden vuoksi jaotellaan traceback osiin:

```
Traceback (most recent call last):
File "<ipython-input-38-055a27710ada>", line 1, in <module><
    runfile('C:/Users/User/Desktop/ErrorExample.py', wdir='C:/Users/Sammako/Desktop')
File "C:\Users\User\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 705, in runfile
    execfile(filename, namespace)
File "C:\Users\User\Anaconda3\lib\site-packages\spyder\utils\site\sitecustomize.py", line 102, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)
```

Yllä olevat viestit käsittelevät ohjelman kääntämistä, eikä niihin syvennytä tällä kurssilla.

```
File "C:/Users/User/Desktop/ErrorExample.py", line 5, in <module>
    print(palautaAlkio(lista, 3))
```

Jälleen kerran tracebackistä selviää ohjelman polku, rivi, sekä rivin sisältö

```
File "C:/Users/User/Desktop/ErrorExample.py", line 2, in palautaAlkio
    return lista[alkio]
```

Koska rivi, joka aiheuttaa virheen on funktiokutsu, ilmoittaa traceback vielä erikseen virheen sijaitsevan funktiossa nimeltä palautaAlkio, sekä rivin, jossa virhe sijaitsee sekä, sisällön.

Tässä esimerkissä funktio, jossa virhe on, sijaitsee samassa tiedostossa kuin sen kutsu. Tapauksissa, jossa ohjelma kutsuu useita eri funktiota, useista eri tiedostoista, tämän kaltainen viesti on erittäin hyödyllinen.

```

```

```
IndexError: list index out of range
```

Virhe on *IndexError* eli ohjelma yrittää kutsua listan alkiota, jota ei ole olemassa.

On mahdollista, että virhe aiheutuu kirjastossa kuten numpy. Tällöin traceback saattaa ilmoittaa virheen sijaitsevan esim. rivillä 1000, jossain satunnaisessa tiedostossa. Tällöin täytyy etsiä viimeisin rivi tracback:ssä, joka sijaitsee luomassasi tiedostossa.

Esimerkki 3

Tracebackistä ei kuitenkaan aina ole apua. Otetaan esimerkiksi koodi

```
def kerro_kymmenella(numero):
    return numero * 0
numero = 5
jako = 1 / kerro_kymmenella(numero)
print(jako)
```

Käyttäjä on luonut funktion, jonka pitäisi kertoa numero kymmenellä, mutta teki kirjoitusvirheen, minkä seurauksesta funktio palauttaa aina nollan. Kun tarkastelemme saatua tracebackiä, huomaamme ongelman olevan rivillä 6, sekä ongelman johtuvan nollalla jakamisesta.

Koska tehty virhe ei suoranaisesti aiheuta virhettä, vaan antaa väärän tuloksen, joka myöhemmin aiheuttaa virheen, ei traceback ole yhtä hyödyllinen tässä tilanteessa. On myös tilanteita, kuten graafiset käyttöliittymät, jotka eivät aina kaatuessaan luo tracebackiä. Miten siis löytää virhe, kun sen sijaintia ei tiedetä?

Virheen löytäminen

print-funktiot ovat yksinkertainen tapa löytää mahdollisia virheitä koodista. Alla oleva koodi kaatuu, mutta koska virhe on ikuinen silmukka ei traceback-viestiä synny.

```
numero = 5
kertoma = 1
print("testi")
while numero > 1:
    kertoma *= numero
print("testi")
```

Kun koodi ajetaan *print*-funktioiden kanssa, huomataan ensimmäisen tulostuvan, mutta toisen ei. Tästä on helppo päätellä, että virhe on while-silmukassa. Kun virheen sijainti tiedetään, on helppo huomata virheen johtuvan siitä, että muuttujan *numero* arvoa ei vähennetä silmukassa.

Virheiden käsittelystä kerrotaan lisää muualla oppimateriaalissa.