Ohjelmoinnin peruskurssi Y1

CS-A1111

31.10.2018

Oppimistavoitteet: tämän luennon jälkeen

- Sinulla on peruskäsitys siitä, mitä hyvällä ohjelmointityylillä tarkoitetaan ja miksi sen käyttäminen on tärkeää.
- Osaat käsitellä ohjelmassa erilaisia virhetilanteita niin, että ohjelma ei kaadu esimerkiksi silloin, jos käyttäjä antaa pyydetyn luvun sijasta tekstiä.
- Tiedät, miten ohjelman käsitelemää tietoa voi säilyttää ohjelman suorituskerrasta toiseen käyttämällä hyväksi tiedostoja.
- Osaat kirjoittaa ohjelman, joka lukee rivejä tekstitiedostosta ja tekee luetuille riveille haluttuja asioita.
- Lisäksi tämän kerran kalvojen loppuosassa kerrotaan, millaisia vaiheita ohjelman suunnittelussa ja kirjoittamisessa on sekä esitellään lyhyesti niitä periaatteita, joiden mukaan ohjelman rakennetta voi suunnitella.
- Voit luennon aikana lähettää kysymyksiä ja kommentteja sivulla `http://presemo.aalto.fi/y1s2018`

Hyvä ohjelmointityyli

- Miksi ihmisen pitää pystyä lukemaan valmista ohjelmakoodia?
  - virheiden korjaus
  - ohjelman laajentaminen tai muuttaminen
  - pohja uutta ohjelmaa tehdessä.
- Hyvä ohjelmointityyli auttaa ohjelman lukemisessa ja ymmärtämisessä.

Mitä seuraava funktio tekee?

```
def f(t1,t2):
    i=0 #nollataan i
    x=0 #nollataan x
    y=len(t1) #y saa arvokseen t1:n pituuden
    while i<y: #jatketaan niin kauan, että i on y
        z=t1[i]-t2[i] #sijoitetaan erotus z:aan
        x+=z #lisataan z x:aan
        i+=1 #kasvatetaan i:ta yhdella
    if y>0:
        k=x/y #k:n arvoksi tulee x jaettuna y
    else:
        k=0.0 #nollataan k
    return k #palautetaan k:n arvo
```

Sama funktio kirjoitettuna toisin

```
def laske_erojen_keskiarvo(lista1, lista2):
    i = 0
    erojen_summa = 0
    alkioiden_lkm = len(lista1)
    while i < alkioiden_lkm:
        alkioiden_ero = lista1[i] - lista2[i]
        erojen_summa += alkioiden_ero
        i += 1
    if alkioiden_lkm > 0:
        keskiarvo = erojen_summa / alkioiden_lkm
    else:
        keskiarvo = 0.0
    return keskiarvo
```

- Kumpaa funktiota on helpompi ymmärtää?

Hyvään ohjelmointityyliin kuuluvia asioita

- Ohjelman järkevä jako funktioihin. (Katso luentokalvojen loppuosa.)
- Kuvaavat nimet muuttujilla, parametreilla, funktioilla jne.
- Järkevä kommentointi.
- Sisennykset.
- Tyhjien tilan järkevä käyttö.

Nimistä

- Muuttujat, parametrit ja funktiot on syytä nimetä niin, että ne kuvaavat mahdollisimman hyvin muuttujan tms. tarkoitusta.
- Esimerkiksi `korkeus` vs. `luku`.
- Yksikirjaimiset nimet ovat suositeltavia vain erikoistapauksissa:
  - Toistokäskyn kierroslaskureina.
  - Muuuttujan nimi on sama kuin esim. matematiikan tai fysiikan kaavassa yleisesti käytetty symboli.
- Muuttujien nimet on syytä kirjoittaa pienellä kirjaimella ja vakioiden nimet isolla kirjaimella.

Kommentit

- Selvittävät koodin lukijalle sen rakennetta ja toimintaa.
- Ohjelmatiedoston alkuun yleiskommentti, jossa kerrotaan ohjelman kirjoittaja ja se, mitä tiedostossa oleva ohjelma tekee.
- Jokaisen funktion otsikon yhteyteen kommentti, joka kertoo funktion tarkoituksen, parametrien merkityksen sekä funktion palauttaman arvon.
- Funktioiden sisällä on syytä kommentoida vain suurempia kokonaisuuksia ja epätavallisia ratkaisuja.
- Älä kommentoi asioita, jotka ovat itsestään selviä jokaiselle Python-kieltä osaavalle, esim.

```
i += 1   #i:ta kasvatetaan yhdella
```

Tyhjän tilan käyttö

- Tyhjät rivit helpottavat ohjelman rakenteen näkemistä.
- Kahden funktion väliin on aina syytä jättää 1–2 tyhjää riviä.
- Myös pidemmän funktion sisällä voi erottaa kokonaisuuksia tyhjillä riveillä.
- Älä kirjoita koodia, jossa joka toinen rivi on tyhjä — se vain hankaloittaa ohjelman lukemista.
- Erota operaattori ja operandit toisistaan välilyönnillä.
- Myös pilkun jälkeen kannattaa yleensä kirjoittaa välilyönti.

Poikkeukset

- Ohjelmaa suoritettaessa voidaan törmätä virhetilanteisiin.
- Osa virheistä johtuu ohjelmointivirheistä, mutta osaan ohjelmoija ei voi vaikuttaa (esim. väärän tyyppinen syöte).
- Virhetilanteiden käsittely if–else-rakenteen avulla tekee ohjelmasta helposti sekavan.
- Python tarjoaa virhetilanteiden käsittelyyn oman mekanismin, *poikkeukset*.
- Poikkeus voidaan käsitellä try–except-rakenteen avulla.

try–except-rakenne

```
try:
    # Jono kaskyja, joista jokin tai jotkin
    # voivat aiheuttaa poikkeuksen.
except poikkeuksen_tyyppi:
    # Kaskyja, jotka jotenkin selvittavat
    # virhetilanteen, jos on aiheutunut
    # poikkeuksen_tyyppi-tyyppinen poikkeus.
```

Virheelliseen syötteeseen varautuminen

- Ohjelma yrittää muuttaa käyttäjän syötteen luvuksi, mutta syöte on väärää tyyppiä $\Rightarrow$ aiheutuu `ValueError`-tyyppinen poikkeus.
- Vaihtoehtoja poikkeuksen käsittelemiseksi:
  - Käyttäjälle annetaan except-osassa selväsanainen virheilmoitus.
  - Käyttäjältä pyydetään uutta syötettä niin kauan, että hän antaa luvun.

Esimerkki: naulamuunnos

```
def main():
    NAULAKERROIN = 0.4536
    print("Muutan nauloina annetun massan kilogrammoiksi.")
    try:
        syote = input("Anna massa nauloina: ")
        naulat = int(syote)
        kilot = NAULAKERROIN * naulat
        print("Massa on {:.3f} kg".format(kilot))
    except ValueError:
        print("Virhe: et antanut nauloja kokonaislukuna.")

main()
```

Syötteen pyytäminen uudelleen

- Parempi versio pyytää käyttäjältä nauloja niin kauan, että hän antaa kokonaisluvun.
- Uutta pyyntöä ei sijoiteta except-osaan, vaan koko try–except-osa sijoitetaan toistokäskyn sisään.
- Toistokäskyn suoritusta jatketaan niin kauan, että on saatu luettua kelvollinen syöte.

Naulamuunnos: uusi koodi

```
def main():
    NAULAKERROIN = 0.4536
    print("Muutan nauloina annetun massa kilogrammoiksi.")
    luku_onnistui = False
    while not luku_onnistui:
        try:
            syote = input("Anna massa nauloina: ")
            naulat = int(syote)
            kilot = NAULAKERROIN * naulat
            print("Massa on {:.3f} kg".format(kilot))
            luku_onnistui = True
        except ValueError:
            print("Virhe: naulat pitää olla kokonaisluku.")
            print("Yrita uudelleen!")


main()
```

Välitehtävä 1

- Miksi naulojen pyytämistä uudelleen ei sijoitettu except-osan sisään?
- `http://presemo.aalto.fi/y1s2018`

Apufunktio luvun lukemiseen

- Jos samassa ohjelmassa luetaan kokonaislukuja useassa kohdassa, kannattaa kirjoittaa apufunktio kokonaisluvun lukemiseen.
- Funktio lukee ja palauttaa kokonaisluvun. Se pyytää käyttäjältä uutta kokonaislukua niin kauan, että saadaan kelvollinen kokonaisluku.
- Vastaavat apufunktiot voidaan kirjoittaa myös muuntyyppisten arvojen lukemiseen.

Kokonaisluvun lukeminen apufunktion avulla: koodi

```
def lue_kokonaisluku():
    luku_onnistui = False
    while not luku_onnistui:
        try:
            luku = int(input())
            luku_onnistui = True
        except ValueError:
            print("Virheellinen kokonaisluku!")
            print("Anna uusi!")
    return luku
```

Kokonaisluvun lukeminen apufunktion avulla: koodi jatkuu

```
def main():
    NAULAKERROIN = 0.4536
    print("Muutan nauloina annetun massa kilogrammoiksi.")
    print("Anna massa nauloina.")
    naulat = lue_kokonaisluku()
    kilot = NAULAKERROIN * naulat
    print("Massa on {:.3f} kg".format(kilot))


main()
```

Huomatuksia poikkeuksista

- try–except-rakenne voi sisältää useita except-osia erityyppisiä poikkeuksia varten. Tällöin poikkeuksen sattuessa siirrytään ensimmäiseen except-osaan, jonka poikkeuksen tyyppi vastaa aiheutunutta poikkeusta.
- Ohjelmoija voi myös itse aiheuttaa poikkeuksen (virhetilanteen sattuessa) `raise`-käskyllä (ei käsitellä tällä kurssilla).

Tiedostot

- Tiedostojen käsittelyä tarvitaan esimerkiksi seuraavissa tilanteissa:
  - Ohjelman käsittelemiä tietoa halutaan säilyttää ohjelman suorituskerrasta toiseen (esim. puhelinluettelo, opiskelijarekisteri).
  - Halutaan, että käyttäjän ei tarvitse syöttää ohjelman lähtötietoja jokaisella suorituskerralla (esim. mittaussarjan parametrit).
  - Ohjelman on käsiteltävä jonkun muun ohjelman tuottamaa dataa.
- Ohjelma lukee tarvittavat lähtötiedot tiedostosta.
- Jos ohjelma tekee tietoihin muutoksia ja muuttuneita tietoja halutaan käyttää seuraavalla suorituskerralla, ohjelma kirjoittaa muuttuneet tiedot tiedostoon.

Tekstitiedosto vs. binääritiedosto

- Tiedostot jaetaan tekstitiedostoihin ja binääritiedostoihin.
- Tekstitiedostossa tiedot on tallennettu merkkeinä, esimerkiksi luku 147 merkkeinä 1, 4 ja 7.
- Tekstitiedostoa voi muokata millä tahansa tekstieditorilla.
- Binääritiedostossa tiedot on esitetty binääriesitysmuodossa, esimerkiksi luku 147 vastaavana binäärilukuna.
- Binääritiedostoa ei yleensä pysty käsittelemään järkevästi tavallisella tekstieditorilla.
- Tällä kurssilla opetetaan ainoastaan tekstitiedostojen käsittely.

Tiedoston avaaminen

- Kerrottaan, mitä fyysistä tiedostoa ohjelman muuttuja vastaa.
- Samalla tietokoneen käyttöjärjestelmä varautuu käsittelemään ko. tiedostoa.
- Tätä kutsutaan *tiedoston avaamiseksi*, esimerkki

```
tiedostomuuttuja = open("teksti.txt","r")
```

- Ensimmäinen parametri on käsiteltävän tiedoston nimi käyttöjärjestelmässä. Tarvittaessa sen pitää sisältää polku tiedoston hakemistoon.
- Toinen parametri kertoo tiedoston käsittelytavan.

Lisää tiedoston avaamisesta

- Mahdollisia käsittelytapoja:
  - `r` lukeminen
  - `w` kirjoittaminen, vanha sisältö häviää
  - `a` kirjoittaminen, kirjoitetaan vanhan sisällön perään.
- Tiedoston avaaminen lukemista varten aiheuttaa poikkeuksen, jos tiedostoa ei ole tai sitä ei pystytä jostain muusta syystä lukemaan.
- Tiedoston käsittelyyn liittyvien poikkeusten "ylätyyppinä" on `OSError`-tyyppinen poikkeus. Se on syytä käsitellä try–except-rakenteella aina, kun luetaan tiedostosta tai kirjoitetaan tiedostoon.

Rivin lukeminen ja tiedoston sulkeminen

- Jos muuttuja `tiedostomuuttuja` viittaa lukemista varten avattuun tiedostoon, niin siitä voi lukea rivin kerrallaan metodin `readline` avulla seuraavasti:

  ```
  luettu_rivi = tiedostomuuttuja.readline()
  ```

  Luettu rivi sisältää myös sen lopussa olevan rivinvaihtomerkin.
- Seuraava `readline`-käsky lukee tiedoston seuraavan rivin jne.
- Jos tiedosto on jo luettu loppuun ja kutsutaan `readline`-metodia, se palauttaa arvona tyhjän merkkijonon `""`
- Kun tiedoston lukeminen päättyy, tiedosto pitää sulkea `close`-käskyllä:

  ```
  tiedostomuuttuja.close()
  ```

Esimerkkiohjelma tiedoston lukemisesta

```
def main():
    try:
        lahtotiedosto = open("tekstia.txt", "r")
        rivi = lahtotiedosto.readline()
        while rivi != "":
            print(rivi)
            rivi = lahtotiedosto.readline()
        lahtotiedosto.close()
    except OSError:
        print("Virhe tiedoston lukemisessa. Ohjelma paattyy.")


main()
```

Rivinvaihtomerkistä

- Edellisen kalvon ohjelma tulostaa ylimääräisen tyhjän rivin jokaisen rivin jälkeen.
- Tämä johtuu siitä, että tiedostoista luettujen rivien lopussa on rivinvaihtomerkki.
- Jos ylimääräiset rivinvaihdot voidaan välttää poistamalla rivinvaihtomerkki rivin lopusta ennen tulostamista.
- Yksi tapa poistaa rivinvaihtomerkki on käyttää metodia `rstrip`. Se poistaa kuitenkin myös muut "tyhjät merkit" rivin lopusta.

Parannettu versio tiedostonlukuohjelmasta

```
def main():
    nimi = input("Anna luettavan tiedoston nimi: ")
    try:
        lahtotiedosto = open(nimi, "r")
        rivi = lahtotiedosto.readline()
        while rivi != "":
            rivi = rivi.rstrip()
            print(rivi)
            rivi = lahtotiedosto.readline()
        lahtotiedosto.close()
    except OSError:
        print("Virhe tiedoston", nimi,
              "lukemisessa. Ohjelma paattyy.")

main()
```

Tiedoston rivien lukeminen for-käskyllä

- Jos ohjelman on luettava kaikki tiedoston rivit, on usein helpointa käydä ne läpi for-käskyn avulla.
- Käskyn yleinen muoto on

```
for rivimuuttuja in lahtotiedosto:
    tee jotain riville rivimuuttuja
```

- Käskyyn ei tarvitse kirjoittaa lainkaan rivin tiedostosta lukevaa käskyä (esim. `readline`).
- **Tärkeää:** jos ohjelmassa luetaan tiedoston rivit for-käskyn avulla, ei samassa ohjelmassa pidä käyttää `readline`-metodia.

Esimerkki tiedoston lukemisesta for-käskyllä

```
def main():
    nimi = input("Anna luettavan tiedoston nimi: ")
    try:
        lahtotiedosto = open(nimi, "r")
        for rivi in lahtotiedosto:
            rivi = rivi.rstrip()
            print(rivi)
        lahtotiedosto.close()
    except OSError:
        print("Virhe tiedoston", nimi,
              "lukemisessa. Ohjelma paattyy.")


main()
```

Lukujen lukeminen tiedostosta

- Python-tulkki palauttaa tiedostosta luetut rivit aina merkkijonoina.
- Jos tiedostossa on lukuja, pitää luetut rivit muuttaa tyypinmuunnoksella oikeantyyppiksiksi.
- Jos riviä ei voida muuttaa luvuksi, aiheutuu `ValueError`, joka on syytä käsitellä.
- Jos samalla rivillä on useita lukuja, pitää rivi ensin jakaa. Tyypinmuunnos tehdään vasta jaon tuloksena syntyneille luvuille.

Desimaalilukuja tiedostosta, koodi

```
def main():
    nimi = input("Mista tiedostosta lampotilat luetaan: ")
    summa = 0.0
    lkm = 0
    try:
        lampotiedosto = open(nimi, "r")
        for rivi in lampotiedosto:
            rivi = rivi.rstrip()
            lampotila = float(rivi)
            summa += lampotila
            lkm += 1
        lampotiedosto.close()
```

Desimaalilukuja tiedostosta, koodi jatkuu

```
        if lkm == 0:
            print("Tiedostossa ei ollut yhtaan lampotilaa.")
        else:
            keskiarvo = summa / lkm
            print("Lampotilojen keskiarvo on", keskiarvo)
    except OSError:
        print("Virhe tiedoston", nimi,
              "lukemisessa. Ohjelma paattyy.")
    except ValueError:
        print("Virheellinen rivi tiedostossa", nimi,
              "- ohjelma paattyy.")


main()
```

Lukujen lukeminen csv-tiedostosta

- Entä, jos tiedostossa on jokaisella rivillä lämpötilan lisäksi päivämäärä, jolloin lämpötila on mitattu?
- Rivillä on ensin päivämäärä, sitten pilkku ja sen jälkeen lämpötila.
- Tällaisia *csv-tiedostoja* voi tehdä helposti esimerkiksi taulukkolaskentaohjelmilla (csv = comma-separated values).

Lukeminen csv-tiedostosta, koodi

```
def main():
    nimi = input("Mista tiedostosta lampotilat luetaan: ")
    summa = 0.0
    lkm = 0
    try:
        lampotiedosto = open(nimi, "r")
        for rivi in lampotiedosto:
            rivi = rivi.rstrip()
            osat = rivi.split(",")
            if len(osat) == 2:
                lampotila = float(osat[1])
                summa += lampotila
                lkm += 1
            else:
                print("Virheellinen rivi", rivi)
        lampotiedosto.close()
```

Ohjelma jatkuu

```
        if lkm == 0:
            print("Tiedostossa ei ollut yhtaan lampotilaa.")
        else:
            keskiarvo = summa / lkm
            print("Lampotilojen keskiarvo on", keskiarvo)
    except OSError:
        print("Virhe tiedoston", nimi,
              "lukemisessa. Ohjelma paattyy.")
    except ValueError:
        print("Virheellinen rivi tiedostossa", nimi,
              "- ohjelma paattyy.")

main()
```

Ohjelmointiprojektin vaiheet

1. Määrittely
   - Mitä ohjelma täsmällisesti ottaen tekee?
   - Miten ohjelma kommunikoi käyttäjän kanssa?
2. Ohjelman suunnittelu (ohjelman rakenne ja ohjelman käyttämät tietorakenteet)
   - Mitä funktioita (tai muita laajempia osia) ohjelmassa on? Mitä parametreja funktioilla on ja millaisia arvoja ne palauttavat?
   - Mitä tietorakenteita (esim. listat, sanakirja jne) ohjelma käyttää?
3. Koodaus ohjelmointikielelle
4. Testaus (rinnakkain koodauksen kanssa)
   - Testataan aluksi mahdollisimman pieniä osia (esim. yksittäisiä funktioita) kerrallaan, sitten laajempia kokonaisuuksia.
5. Käyttöönotto
6. Ylläpito
   - Mahdollisten virheiden korjaus ja uusien omainaisuuksien lisääminen.

Suunnittelu: mitä funktioita ja tietorakenteita ohjelmaan tulee?

- Kirjoita kuvaus ohjelman toiminnasta.
- Millaisista osatehtävistä ohjelman toiminta koostuu?
- Yleensä kutakin osatehtävää varten kirjoitetaan oma funktio.
- Aloita ohjelman tärkeimmistä osatehtävistä ja tarkenna sitten näiden toimintaa. Tällöin saattaa osoittautua tarpeelliseksi määritellä uusia osatehtäviä.
- Lisäksi on mietittävä, mitä tietoja funktio tarvitsee muulta ohjelmalta (parametrit) ja mitä tietoja se tuottaa muulle ohjelmalle (paluuarvot).
- Huomaa: tämä lähestymistapa ei sovi olio-ohjelmointiin.
- Tietorakenteet: Mieti, mitä tietoa ohjelma joutuu käsittelemään ja missä muodossa se kannattaa tallentaa. Tarvitaanko esim. merkkijonoja, listoja. sanakirjoja tms. yksittäisiä lukuja esittävien muuttujien lisäksi?

Lisää funktioiden suunnittelusta

- Tavoitteena on se, että funktio näyttää ulkopuolelle mustalta laatikolta: funktion käyttäjän tarvitsee tietää, mitä lähtötietoja funktio tarvitsee ja mitä se palauttaa, mutta ei funktion toiminnan yksityiskohtia.
- Funktion sisäinen toteutus ei saa vaikuttaa muuhun ohjelmaan.
- Funktioiden pituus pitää suunnitella sopivaksi. Yhden rivin mittaisista käskyistä kannattaa yleensä tehdä funktioita vain silloin, jos ne laskevat jonkin matemaattisen lausekkeen arvon tai tarkastavat monimutkaisemman ehdon totuusarvon. Toisaalta liian pitkät funktiot vaikeuttavat ohjelman rakenteen ymmärtämistä.
- Funktion pitäisi olla loogisesti yhtenäinen kokonaisuus.

Esimerkki: valikkopohjainen puhelinluettelo

- Puhelinluettelo-ohjelma, joka säilyttää nimi- ja puhelinnumerotiedot tiedostossa.
- Käyttäjälle tulostetaan valikko, joka kertoo mahdolliset toimenpiteet. Käyttäjä valitsee valikosta aina yhden toimenpiteen kerrallaan, kunnes hän lopettaa ohjelman suorituksen.
- Kirjoitetaan oma funktio jokaista eri toimenpidettä (tietojen lukeminen tiedostosta numeron kysyminen, numeron lisäys, numeron muuttaminen, numeron poisto, tietojen tallentaminen tiedostoon) varten.
- Lisäksi kirjoitetaan oma funktio, joka tulostaa käyttäjälle valikon ja pyytää käyttäjän valinnan. Se palauttaa käyttäjän valinnan.
- Käytettävä puhelinluettelo välitetään sitä käsitteleville funktioille parametrina.
- Pääohjelma sisältää toistokäskyn, joka kutsuu aina valikon tulostavaa funktiota ja sen jälkeen valitsee suoritettavan funktion käyttäjän valinnan mukaan.