LUT Python ohjelmointiopas 2023 - osa 9
Sisällysluettelo

Luku 9: Laadunvarmistus ja virheenkäsittely

Ohjelman suoritus ei aina etene suunnitellusti, vaan jotain virheellistä ja ennalta odottamatonta saattaa tapahtua ohjelmasi ajon aikana. Ajatellaan vaikka tilannetta, jossa yrität lukea tiedostoa, jota ei ole olemassa, tai muuttaa merkkijonoa kokonaisluvuksi. Tässä tilanteessa ohjelmasi aiheuttaa poikkeuksen, joka johtaa useimmiten siihen, että tulkki keskeyttää käynnissä olleen prosessin ja tulostaa virheilmoituksen. Tässä luvussa käymme läpi toimenpiteitä, joilla voimme ottaa kiinni näitä poikkeuksia ja virhetiloja sekä toipua niistä ilman, että tulkki keskeyttää ohjelman suorittamisen virheilmoitukseen. Usein järkevä lopputulos on kertoa käyttäjälle tapahtuneesta ja lopettaa ohjelman suoritus hallitusti. Näin käyttäjä tietää, mikä meni pieleen ja voi yrittää ratkaista asian sopivalla tavalla.

Virhetilanteiden ennaltaehkäisy

Olemme pyrkineet estämään virhetilanteita ennakolta jo aiemmin tässä oppaassa. Aina jakolaskujen yhteydessä tulisi tarkastaa, ettei jakaja ole 0 virhetilanteen estämiseksi. Toinen esimerkki virhetilanteen ennakoinnista on ennen aliohjelmakutsua tehtävä tarkastus, että listassa on alkioita tyypillisesti analysoitavaksi tai tulostettavaksi. Tässä tapauksessa ongelma ei ole yhtä ilmeinen kuin nollalla jaettaessa, mutta tarkistamalla sisällön olemassaolo ennen aliohjelmaan siirtymistä, siellä ei tarvitse tehdä useita erilaisia tarkastuksia tai palata aliohjelmasta ensimmäisenä olevan tarkastuksen jälkeen tms. Virheiden ennaltaehkäisyn tavoitteena on estää ongelmatilanteiden syntyminen mahdollisimman aikaisin, jolloin tyypillisesti riittää kertoa asiasta käyttäjälle eikä muuta tarvita.

Esimerkeissä 9.1 ja 9.2 näkyvät edellä mainitut tavat ehkäistä virheitä ennakolta. Käytännössä tämä tapahtuu ehdollisena koodina eli esimerkissä 9.1 jakajan ollessa nolla, ei suoriteta jakolaskua vaan kerrotaan käyttäjälle ongelmasta. Vastaavasti ennen tietojen tallennusta esimerkissä 9.2 tarkastetaan, että tallennettavassa listassa on alkioita ja jos näin ole, kerrotaan ongelmasta käyttäjälle.

Esimerkki 9.1. Nollalla jaon ennaltaehkäisy

```
Luku = 4
Jakaja = 0
if (Jakaja != 0):
    print(Luku / Jakaja)
else:
    print("Nollalla ei voi jakaa.")
```

Molemmat esimerkit 9.1 ja 9.2 ovat yksikertaisia tapauksia, joissa ongelma voidaan välttää yksinkertaisella tarkastuksella. Aina tämä ei onnistu ja jos se on tilanne, silloin asiaa pitää harkita tarkemmin ja toimia tarpeen mukaan. Lähtökohtana kuitenkin on, että helpointa on välttää ongelmat ennakolta ja kertoa käyttäjälle asiasta sekä pyrkiä jatkamaan ohjelman suoritusta havaitusta ongelmasta huolimatta.

Esimerkki 9.2. Tyhjän listan aiheuttamien ongelmien ennaltaehkäisy, ks. esimerkki 7.10

```
...
    print("1) Kysy tiedot")
    print("2) Tallenna tiedosto")
    print("3) Lue tiedosto")
...

...
    Tiedot = []
    while (Valinta != 0):
        Valinta = valikko()
        if (Valinta == 1):
            Tiedot = kysy(Tiedot)
        elif (Valinta == 2):
            if (len(Tiedot) > 0):
                tallenna(TiedostoNimi, Tiedot)
            else:
                print("Ei tietoja tallennettavaksi.")
        elif (Valinta == 3):
...
```

Esimerkin 9.2 toistorakenteessa aliohjelman suorittamatta jättäminen ei aiheuta ongelmaa vaan toistorakenne aloittaa alusta ja ohjelman suoritus jatkuu normaalisti. Ohjelman rakenteen suunnittelulla on keskeinen rooli minimoitaessa virhe- ja poikkeustilanteiden seurauksia.

Virhetilanteiden käsittely

Tyypillisiä ohjelmoinnin virhetilanteita

Olemme nyt oppaan luvussa 9, joten olet varmasti nähnyt erilaisia virheilmoituksia ohjelmia tehdessäsi. Tyypillisimpiä virheilmoituksia ovat kirjoitusvirheet koodia kirjoittaessasi ja olet varmaan joskus tehnyt esimerkissä 9.3 näkyvät virheet eli sisentänyt koodia väärin tai kirjoittanut jonkun funktionimen väärin esim. isolla kirjaimella. Kuten aiemmin on ollut puhetta, sisennysvirhe johtaa SyntaxError-virheeseen, jossa lukee tyypillisesti ”`unexpected indent`” kuvan 9.1 mukaisesti. Toisaalta isolla kirjaimella kirjoitettu `print`-käsky käyttö johtaa tulkin virheilmoituksiin interaktiivisessa ikkunassa, jossa lukee tyypillisesti esimerkin 9.3 Tuloste-kohdan mukaisesti ”`NameError: name 'Print' is not defined.`” tai jotain vastaavaa, ja tulkki saattaa myös yrittää auttaa sinua korjaamaan ongelman kysymällä ”`Did you mean: 'print'?`” Tällä kertaa meidän kannalta oleellista on, että nämä kaksi erilaista virhettä johtivat kahteen erilaiseen palautteeseen tulkin kannalta, joka kertoo meille ohjelman suorituksen estävästä virheestä ja mahdollisesti yrittää auttaa meitä ratkaisemaan sen.

Esimerkki 9.3. Kahden rivin ohjelma kahdella virheellä ja virheilmoituksella

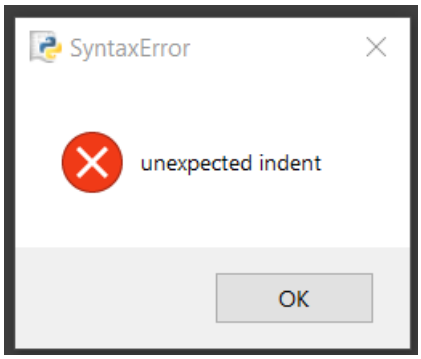
```
 print("moi")
Print("moi")
```

Tuloste

```
Traceback (most recent call last):
  File "tst.py", line 2, in <module>
    Print("moi")
NameError: name 'Print' is not defined. Did you mean: 'print'?
```

Esimerkin 9.3 sisennysvirheeseen liittyvä virheilmoitus näkyy kuvassa 9.3.

Kuva 9.1. Väärin sisennetyn print-käskyn aiheuttava virheilmoitus



Ohjelmoija voi siis tehdä virheitä ohjelmia kirjoittaessaan ja samoin käyttäjä voi tehdä virheitä syöttäessään tietoa ohjelmalle esimerkin 9.4 mukaisesti. Esimerkin 9.4 osassa 1 näkyy normaali kokonaislukutiedon kysyminen käyttäjältä sekä tuloksen tulostus näytölle, ja mikäli käyttäjä toimii annettujen ohjeiden mukaan, kaikki toimii ongelmitta. Tulosteen mukaisesti käyttäjälle voi kuitenkin tulla näppäilyvirhe ja hän saattaa syöttää ohjelmalle esimerkiksi kirjaimen kokonaisluvun sijasta. Tulkki toimii tässä yhteydessä samalla tavoin kuin aiemmin väärin kirjoitetun funktionimen kohdalla ja antaa virheilmoituksen ”`ValueError: invalid literal for int() with base 10: 'r'`”.

Esimerkin 9.4 osassa 2 on pyritty estämään tämä virhe lähtien siitä, että käyttäjän syöte on merkkijono ja sen onnistunut muutos kokonaisluvuksi edellyttää, että kaikki annetut merkit ovat numero-merkkejä eli merkkijono-muuttujan `isdigit()`-jäsenfunktion tulee palauttaa totuusarvo tosi. Jos näin on, ohjelma toimii ongelmitta ja vaihtoehtona on kertoa käyttäjälle, ettei annettu syöte ole muutettavissa kokonaisluvuksi.

Esimerkki 9.4.

```
# Osa 1. Käyttäjäsyöte voi johtaa virhetilanteeseen
Ika = int(input("Kuinka vanha olet: "))
print(Ika)

# Osa 2. Käyttäjäsyötteen tarkastus ennen sen käyttöä
Syote = input("Kuinka vanha olet: ")
if (Syote.isdigit() == True):
    Ika = int(Syote)
    print(Ika)
else:
    print("'{}' ei ole numero.".format(Syote))
```

Tuloste

```
Kuinka vanha olet: r
Traceback (most recent call last):
  File "C:\tst.py", line 2, in <module>
    Ika = int(input("Kuinka vanha olet: "))
ValueError: invalid literal for int() with base 10: 'r'
```

Esimerkin 9.4 osan 2 ratkaisu on toimiva ja hyvä ratkaisu tähän ongelmaan, joka onnistuu aiemmin läpikäydyillä konsepteilla. Käytännössä erilaisia ongelmia on kuitenkin useita erilaisia ja sen vuoksi asiaan on kehitetty monipuolinen ratkaisu, joka tunnetaan poikkeustenkäsittely-konseptina. Tutustutaan siihen seuraavaksi tarkemmin.

Poikkeustenkäsittely `try-except` -rakenteella

Ohjelmissa ongelmia voidaan estää tarkastamalla tietoja etukäteen ja tekemällä operaatioita vasta sen jälkeen. Käytännössä virhetilanteita on kuitenkin paljon erilaisia kuten on myös niiden syitä. Käyttäjä on yksi esimerkki ohjelman ulkopuolisesta asiasta, jota ei voida kontrolloida ohjelman toimesta. Toinen ohjelman ulkopuolinen järjestelmä on käyttöjärjestelmä, jota tarvitaan tiedostojenkäsittelyssä. Käytännössä ohjelma pyytää käyttöjärjestelmää kirjoittamaan tietoja nimettyyn tiedostoon ja vastaavasti ohjelma voi pyytää käyttöjärjestelmää lukemana nimetystä tiedostosta merkkejä halutun määrän. Tiedostonkäsittely voi olla mahdotonta useista erilaisista syistä, sillä luettavaa tiedostoa ei välttämättä ole (ilmoitetussa osoitteessa), käyttäjältä voi puuttua sen luku- ja/tai kirjoitusoikeus, tiedosto voi olla luettu jo loppuun, verkkoyhteys voi katketa kesken lukuoperaation tai jokin muu odottamaton tapahtuma voi estää tiedoston lukemiseen. Siksi ohjelmissa on pakko varautua poikkeuksiin eli epänormaaleihin tilanteisiin.

Olio-ohjelmissa virhetilanteet käsitellään tyypillisesti poikkeustenkäsittelijöillä, jotka hoitavat poikkeustilanteet poikkeuksina. Tämä mekanisti on rakennettu olio-ohjelmiin alusta alkaen mukaan, joten se on toimiva ja monipuolinen konsepti, joka mahdollistaa poikkeustenkäsittelijöiden tekemisen lisäksi sen, että ohjelmat voivat aiheuttaa eli ”nostaa” poikkeuksia. Koska tämä ei ole olio-ohjelmointiopas, emme mene tähän asiaan tämän tarkemmin vaan keskitymme poikkeustenkäsittelyyn käytännöllisestä näkökulmasta.

Esimerkin 9.4 osassa 2 teimme käyttäjäsyötteen tarkastuksen ennen sen käyttöä ja tämä on aina toimiva ratkaisu. Vaihtoehtoinen ratkaisu poikkeustenkäsittelijällä näkyy esimerkissä 9.5 eli käytämme nyt `try-except` -rakennetta, jossa `try` ja `except` -avainsanojen välissä on koodiosio, jota seurataan tarkemmin poikkeusten varalta. Mikäli tässä osiossa tapahtuu poikkeus, suoritetaan `except`-osiossa oleva poikkeustenkäsittelyyn tehty koodilohko. Huomaa, että `try` ja `except` -avainsanat ovat sisentämättä ja vastaavasti tämän poikkeustenkäsittelijän jälkeen normaali koodi on taas sisentämättä samalla tasolla kuin ne.

Esimerkin 9.5 tulosteessa näkyy, miten syötteellä 5 eli normaalilla kokonaisluvulla ohjelma jättää `except`-lohkossa olevan koodin suorittamatta, mutta käyttäjän antaessa kokonaisluvuksi merkkijonon `moi`, ohjelma suorittaa poikkeustenkäsittelijän koodirivit. Tässä esimerkissä poikkeustenkäsittelijää käytetään vain estämään ohjelman kaatuminen ja poikkeuksesta toipumiseen ei kiinnitetä huomiota, koska tässä tapauksessa riittää virhetilanteen ohitus.

Esimerkki 9.5. Poikkeustenkäsittelijä virhetilanteiden estämiseen

```
try:
    Ika = int(input("Kuinka vanha olet: "))
    print(Ika)
except:
    print("Syötteen muuttaminen numeroksi ei onnistunut.")
print("Ohjelma jatkuu normaalisti try-except -koodilohkon jälkeen.")
```

Tuloste

```
Kuinka vanha olet: 5
5
Ohjelma jatkuu normaalisti try-except -koodilohkon jälkeen.

Kuinka vanha olet: moi
Syötteen muuttaminen numeroksi ei onnistunut.
Ohjelma jatkuu normaalisti try-except -koodilohkon jälkeen.
```

Esimerkin 9.5 poikkeustenkäsittelijä on minimaalinen, mutta riittävä tässä yksinkertaisessa tapauksessa. Seuraavaksi katsomme, miten voimme valita meitä kiinnostavia poikkeuksia tai useita ja käsitellä niitä eri tavoin.

Poikkeuksia ja poikkeustenkäsittelijöitä

Esimerkissä 9.5 otimme kiinni poikkeuksen ja jatkoimme ohjelman suoritusta kiinnittämättä huomiota tapahtuneeseen virheeseen. Kuitenkin esimerkissä 9.3 tulkki kertoi virheen olevan `NameError` ja esimerkissä 9.4 virhe oli `ValueError`, eli virheitä on erilaisia ja ne liittyvät erilaisiin tilanteisiin. Käytännössä Python 3.11:ssa on 67 erilaista poikkeusta, jotka ovat nähtävissä tämän luvun lopussa olevassa luokkahierarkia -liitteessä. Tätä liitettä ei kannata opetella ulkoa tämän oppaan laajuudessa, mutta se havainnollistaa konseptin laajuutta ja poikkeuksien nimet kertovat, mihin kaikkeen poikkeuksia voi käyttää. Tässä oppaassa tavoitteemme on tutustua poikkeustenkäsittely-konseptiin ja siihen riittää tutustua muutamiin yleisiin poikkeuksiin, joita ovat mm. `NameError`, `ValueError`, `ZeroDivisionError`, `TypeError`, `IndexError`, `KeyboardInterrupt`, `SystemExit`, `OSError` ja `Exception`.

Esimerkki 9.6. Tyypillisiä poikkeuksia

```
## NameError eli nimessä virhe
try:
    Print("moi")
except NameError:
   print("Poikkeus NameError: Nimessä virhe.")

## ValueError - str vs. int
try:
    Luku = int("Merkkijono")
except ValueError:
    print("Poikkeus ValueError: Merkkijonon muuttaminen kokonaisluvuksi ei
onnistu.")

## ZeroDivisionError eli nollalla jako
try:
    print(5 / 0)
except ZeroDivisionError:
   print("Poikkeus ZeroDivisionError: Nollalla jako.")

## TypeError - str vs. int
try:
   Sana = "Erkki"
   Sana = Sana - 1
except TypeError:
   print("Poikkeus TypeError: Merkkijonon ja numeron laskenta ei
onnistu.")

## IndexError - lista ja indeksit
Lista = [1,2]
try:
   print(Lista[3])
except IndexError:
   print("Poikkeus IndexError: Listassa ei ole haettua indeksiä.")

## Tiedostonkäsittelyn poikkeus Exception
try:
   Tiedosto = open("eiole.txt", "r")
except OSError:
   print("Poikkeus OSError: Yritettiin avata tiedosto, jota ei ole.")
```

Tuloste

```
Poikkeus NameError: Nimessä virhe.
Poikkeus ValueError: Merkkijonon muuttaminen kokonaisluvuksi ei onnistu.
Poikkeus ZeroDivisionError: Nollalla jako.
Poikkeus TypeError: Merkkijonon ja numeron laskenta ei onnistu.
Poikkeus IndexError: Listassa ei ole haettua indeksiä.
Poikkeus OSError: Yritettiin avata tiedosto, jota ei ole.
```

Poikkeusten osalta kannattaa seurata ohjelmaa tehdessä tulevia poikkeuksia ja harkita poikkeustenkäsittelijöiden tarvetta sen pohjalta. Tulkki kertoo poikkeuksen nimen, jonka pohjalta siihen voi tutustua tarkemmin ja kirjoittaa poikkeuksenkäsittelijän tarpeen mukaan. Kokemuksen myötä itselle ja tehtäviin ohjelmiin liittyvät keskeiset poikkeukset oppii tunnistamaan, mutta konseptin laajuuden ja poikkeusten suuren määrän vuoksi asiaa kannattaa lähestyä käytännöllisesti. Tämän oppaan osalta kannattaa seurata tyyliohjeita, sillä ne on tehty tämän oppaan laajuus huomioon ottaen.

Poikkeusten suuren määrän ja moninaisuuden vuoksi on ilmeistä, ettei kaikki poikkeuksia voi käsitellä samalla tavoin. Siksi esimerkissä 9.7 näkyy `try-except` -rakenteen tarjoamia mahdollisuuksia eri poikkeusten käsittelyyn ja ryhmittelyyn. Mikäli useita eri poikkeuksia halutaan käsitellä samalla tavoin, voidaan ne ryhmitellä `except`-sanan perään suluilla ja pilkuilla eroteltuna. Huomaa myös, että `try-except` -rakenne mahdollistaa useiden haarojen tekemisen eri poikkeuksille sekä `else`-haaran käyttämisen. Tässä oppaassa näitä ei tulla käyttämään, mutta tarkempi poikkeuskäsittely edellyttää helposti tarkempaa tutustumista näihin vaihtoehtoihin.

Esimerkki 9.7. Poikkeuskäsittelijän variantteja

```
## Useita virheitä samassa rakenteessa, sulut ja pilkut
try:
   Tiedosto = open("eiole.txt", "r")
   x = 1/0
except (OSError, ZeroDivisionError):
   print("Tuli joku seuraavista poikkeuksista: OSError tai ZeroDivisionError.")


## Useita virheitä kaikki omissa haaroissaan
try:
   Tiedosto = open("eiole.txt", "r")
   x = 1/0
except OSError:
   print("Poikkeus: OSError.")
except ZeroDivisionError:
   print("Poikkeus: ZeroDivisionError.")

## else-rakenne
try:
    luku = int(input("Anna luku: "))
    print("Annoit luvun", luku)
except Exception:
    print("Tapahtui virhe.")
else:
    print("Ei huomattu virheitä.")
```

Esimerkin 9.7 mukaisesti poikkeuksia voi käsitellä eri tavoin, mutta tällaista poikkeuskäsittelijää rakentaessa tulee huomioida poikkeusliitteessä näkyvä hierarkia. Käytännössä `try-except` -rakenne ilman nimettyjä poikkeuksia ottaa kiinni kaikki poikkeukset ja estää mm. käyttäjän `KeyboardInterrupt`-keskeytyksen ja `SystemExit`-komennon toiminnan. Nopeasti, tai huolimattomasti, tehty

poikkeustenkäsittelijä voi siis estää ohjelman normaalin lopettamisen, joten poikkeuksia ei kannata käyttää tutustumatta niiden dokumentaatioon. Antamalla `except`-osassa seurattavaksi poikkeukseksi `Exception`-poikkeuksen, saamme kiinni hierarkian mukaisesti kaikki muut virheet menettämättä mahdollisuutta ohjelman keskeyttämisen käyttäjän toimesta.

Poikkeuskäsittelyn vaihtoehtoina lopetus ja toipuminen

Ongelman eli poikkeuksen käsittelyssä pitää harkita aina tapauskohtaisesti, mitä ongelman jälkeen on tehtävä eli voiko ohjelman suoritusta jatkaa vai onko se käytännössä mahdotonta ja pitääkö ohjelman suoritus lopettaa saman tien isompien ongelmien välttämiseksi. Kuten aiemmin oli puhetta virheiden ennaltaehkäisyn yhteydessä, tavoite on jatkaa ohjelman suoritusta ja esim. tarjota käyttäjälle mahdollisuus antaa jatkamiseen tarvittavat syötteet tms. Toisessa ääripäässä ohjelman ongelma voi estää ohjelman normaalin jatkamisen, jolloin sitä ei kannata edes yrittää vaan on selkeintä kertoa tilanne käyttäjälle ja lopettaa ohjelma.

Edellä keskityimme lähinnä jatkamaan ohjelman suoritusta ongelmasta huolimatta, eikä se ole aina järkevää. Python tarjoaa ohjelman lopettamiseen esimerkissä 9.8 näkyvän `try-finally` -rakenteen, jossa tavoite on mahdollisuuksien mukaan vapauttaa varattuja resursseja jne., ennen kuin ohjelma lopetetaan. Käytännössä tehtävät ”siivousoperaatiot” voivat olla vähäisiä, sillä esim. esimerkin 9.8 tapauksessa ohjelma saattaa kaatua tiedoston avaamiseen tai lukemiseen, joten siivousvaiheessa ei ole varmuutta siitä, onko tiedosto avattu vai ei. Näin ollen tämän oppaan laajuudessa on selkeintä lopettaa ohjelma `sys.exit(0)` -käskyllä, joka huolehtii mahdollisuuksien mukaan resurssien vapauttamisesta ja ilmoittaa käyttöjärjestelmälle, että ohjelma päättyi koodin mukaan hallitusti. Huomaa, että `finally`-osuus suoritetaan **aina**, siis myös siinä tapauksessa, ettei virhettä tapahdu ja siksi sen käyttöä ei suositella tässä oppaassa.

Esimerkki 9.8. Ohjelman lopetus poikkeuskäsittelijässä, try-finally

```
import sys
import time

try:
    Tiedosto = open("L09E4.txt", "r", encoding="utf-8")
    Rivi = Tiedosto.readline()
    while (len(Rivi) > 0):
        time.sleep(2)
        print(Rivi, end="")
        Rivi = Tiedosto.readline()
    Tiedosto.close()
finally:
    print("Ohjelma loppuu, siivotaan jäljet mahdollisuuksien mukaan.")
    sys.exit(0)
```

Esimerkin 9.9 osassa 1 näkyy yrite toipua käyttäjän virheestä kysymällä käyttäjältä uusi jakaja. Tässä yritteessä ongelma on, että jos käyttäjä antaa uudelleen sopimattoman syötteen, ohjelma kaatuu edelleen virheeseen. Siksi osassa 2 virheenkäsittelijä on laitettu toistorakenteeseen, josta pääsee pois vain antamalla hyväksyttävän syötteen. Ohjelma voi siis pakottaa käyttäjää toimimaan ohjeiden mukaisesti, mutta useimmilla tietokoneohjelmien käyttäjillä on kokemuksia siitä, ettei ohjelma hyväksy annettua syötettä eikä tilanteesta pääse eroon muuten kuin sulkemalla ohjelma/selainikkuna/tietokone/kännykkä ja aloittamalla alusta. Vaikka tämä lähestymistapa voi estää väärän tiedon syöttämisen ohjelmaan, tahtoo se johtaa myös tyytymättömiin käyttäjiin ja siksi sille kannattaa harkita muita vaihtoja.

Esimerkki 9.9. Ohjelman toipumisyrite poikkeuskäsittelijässä

```
# Osa 1. Suoraviivainen uusi yritys
Luku = 4
Jakaja = 0
try:
    print(Luku / Jakaja)
except ZeroDivisionError:
    Jakaja = int(input("Jakaja on nolla, anna uusi jakaja: "))
    print(Luku / Jakaja)
print("Ohjelman normaali suoritus jatkuu tästä.")

# Osa 2. Virheellisen syötteen käsittely toistorakenteessa [parempi
if(jakaja==0)]
Luku = 4
Jakaja = 0
while (True):
    try:
        print(Luku / Jakaja)
        break
    except ZeroDivisionError:
        Jakaja = int(input("Jakaja on nolla, anna uusi jakaja: "))
print("Ohjelman normaali suoritus jatkuu tästä.")
```

Virheenkäsittely tässä oppaassa

Tässä vaiheessa poikkeustenkäsittelyn iso kuva rupeaa hahmottumaan ja samalla sen aiheuttama työmäärä voi mietityttää. `try-except` -rakenne rupeaa muistuttamaan valintarakennetta monine haaroineen ja `else`-osioineen, olemme tutustuneet puoleen tusinaa erilaisia poikkeuksia ja Pythonin koko poikkeushierarkia sisältää 67 poikkeusta. Herää siis kysymys, että mahtuuko ohjelmaan enää mitään muuta kuin poikkeustenkäsittelyä? Siksi on hyvä muistaa, että kyseessä on *poikkeusten* käsittely ja etenkin tämän oppaan laajuudessa tehdyissä ohjelmissa se kuuluu sivurooliin. Kaupallisissa ohjelmistoissa tilanne on toinen ja esim. tuhansien pankki- ja tankkausautomaattien pitäminen toiminnassa keskeytyksettä 24/7 on tavoite, johon kannattaa panostaa oikeasti ja toteuttaa poikkeustilanteiden käsittely kattavasti.

Kuten tämän luvun alussa oli puhetta, ensimmäinen tavoitteemme oli virhetilanteiden ennaltaehkäisy. Esimerkissä 9.1 esiteltiin yksi tapa estää nollalla jako tarkastamalla jakajan arvo ennen jakolaskua. Tässä vaiheessa tuo tapa toimii edelleen, mutta sille on vaihtoehtona poikkeustenkäsittelijä, joka ottaa kiinni `ExceptZeroDivision` -poikkeuksen. Nollalla jaon voi siis estää kummalla tavalla tahansa. Toinen ennaltaehkäisyyn liittyvä asia oli aliohjelmakutsuun lähetettävät listat ja sen varmistaminen, että listaa hyödyntäviin aliohjelmiin, ts. analyysi tai tulostus, lähetetään vain alkioita sisältäviä listoja. Esimerkissä 9.2 katsoimme ratkaisun tähän ongelmaan eli listan pituuden tarkastamisen valintarakenteessa ennen aliohjelmakutsua.

Tässä luvussa tutustuimme virhetilanteiden käsittelyyn poikkeustenkäsittely-konseptilla tavoitteena auttaa sinua ymmärtämään sen tarve sekä sen toteuttaminen muutamiin keskeisiin asioihin liittyen. Käytännössä tämä tarkoittaa poikkeustenkäsittelyn toteuttamista aina tiedostonkäsittelyn yhteydessä siten, että käyttäjä saa tietää ongelmasta sekä siitä, missä se tapahtui ennen kuin ohjelma lopetetaan hallitusti. Näin käyttäjällä on realistinen mahdollisuus lähteä selvittämään ongelman syitä ja yrittää eliminoida ne. Tämän oppaan kannalta riittävä tiedostonkäsittelyn yhteydessä toteutettava poikkeustenkäsittely näkyy esimerkissä 9.10 eli kaikki tiedostonkäsittelyyn liittyvät käskyt `open`-käskystä `close`-käskyyn asti ovat saman poikkeuksenkäsittelijän sisällä, eikä se sisällä muita asiaan liittymättömiä käskyjä. Poikkeustenkäsittelijä ottaa kiinni `OSError`-poikkeuksen, joka liittyy käyttöjärjestelmän antamiin keskeytyksiin ja ohjelma voidaan tällöin lopettaa

hallitusti sen jälkeen, kun käyttäjälle on kerrottu minkä tiedoston käsittelyssä virhe tuli. Tiedostoa ei yritetä sulkea, koska virhe on voinut olla joko tiedoston avaamisen tai lukemisen yhteydessä.

Esimerkki 9.10. Minimaalinen tiedostonkäsittelyyn liittyvä poikkeustenkäsittely

```
import sys
Nimi = "data.txt"

try:
    Tiedosto = open(Nimi, "r")
    Rivi = Tiedosto.readline()
    print(Rivi, end="")
    Tiedosto.close()
except OSError:
    print("Tiedoston '{0:s}' käsittelyssä virhe, lopetetaan.".format(Nimi))
    sys.exit(0)
```

Yhteenveto

Osaamistavoitteet

Tämän luvun keskeiset asiat on nimetty alla olevassa listassa. Tarkista sen avulla, että tunnistat nämä asiat ja sinulla on käsitys siitä, mitä ne tarkoittavat. Mikäli joku käsite tuntuu oudolta, käy siihen liittyvät asiat uudestaan läpi. Nämä käsitteet ja asiat on hyvä muistaa ja tunnistaa, sillä seuraavaksi teemme niihin perustuvia ohjelmia.

- Virhetilanteiden ennaltaehkäisy
  - Nollalla jaon ennaltaehkäisy (E9.1)
  - Tyhjän listan aiheuttamien ongelmien ennaltaehkäisy (E9.2)
- Virhetilanteiden käsittely
  - Tyypillisiä ohjelmoinnin virhetilanteita (E9.3, 9.4)
  - Poikkeustenkäsittely try-except -rakenteella (E9.5)
  - Poikkeuksia ja poikkeustenkäsittelijöitä (E9.6, 9.7)
  - Poikkeuskäsittelyn vaihtoehtoina lopetus ja toipuminen (E9.8, 9.9)
- Virheenkäsittely tässä oppaassa (E9.10)
- Yhteenveto, kokoava esimerkki (E9.11)

Pienen Python-perusohjelman tyyliohjeet

Tähän lukuun liittyvät tyyliohjeet on koottu alle. Alla olevia ohjeita noudattamalla vältät yleisimmät virhetilanteisiin ja poikkeuksiin liittyvät ongelmat, ja perustellusta syystä niistä voi ja tulee poiketa. Lähtökohta ohjelman suoritusaikaisten ongelmien käsittelyssä on kertoa käyttäjälle ongelma ja miten sen voi ratkaista. Tässä oppaassa keskitytään alla oleviin asioihin esimerkkeinä ongelmankäsittelystä, eikä virheen- ja/tai poikkeustenkäsittelyä tehdä muutoin, ellei asia mainita erikseen tehtävänannossa.

Virhetilanteiden ennakoinnin ja poikkeustenkäsittelyn periaatteet

1. Ohjelma ei saa kaatua nollalla jakoon. Lähtökohtaisesti käyttäjälle kerrotaan ongelma ja ohjelman suoritus jatkuu normaalisti
2. Aliohjelmaan ei lähetetä tyhjää listaa, jos aliohjelma käyttää listassa olevia alkioita. Käyttäjälle kerrotaan ongelma ja ohjelman suoritus jatkuu normaalisti
3. Tiedostonkäsittely tehdään poikkeustenkäsittelyn sisällä ja poikkeuksen yhteydessä ohjelma lopetetaan

Virhetilanteiden ennakoinnin ja poikkeustenkäsittelyn toteutus

4. Nollalla jaon voi estää alla olevilla tavoilla, jonka jälkeen käyttäjälle kerrotaan ongelmasta ja ohjelman suoritusta jatketaan mahdollisuuksien mukaan. Jos jatkaminen ei onnistu, lopetetaan ohjelman suoritus `sys.exit(0)` -käskyllä
    a. Jakajan voi tarkastaa ehtolauseella
    b. Jakolasku voi olla poikkeustenkäsittelijän sisällä
5. Ennen aliohjelmakutsua tarkastetaan, että listassa on alkioita. Jos listassa on alkioita, ohjelma jatkuu normaalisti ja jos ei ole, kerrotaan tilanne käyttäjälle ja jatketaan ohjelman suoritusta normaalisti
6. Tiedostonkäsittely ja poikkeustenkäsittelijä ovat samassa aliohjelmassa siten, että tiedoston avaus, luku/kirjoitus ja sulkeminen ovat yhdessä poikkeustenkäsittelijässä
7. Poikkeustenkäsittelijän sisällä on vain suoraan tiedostonkäsittelyyn liittyviä käskyjä
8. Tiedostonkäsittelyssä ensimmäinen poikkeuksen sisällä oleva käsky on `open` ja viimeinen `close`
9. Tiedostonkäsittelyssä tarkkailtavaksi poikkeukseksi määritellään aina `OSError`
10. Tiedostonkäsittelypoikkeuksen yhteydessä käyttäjälle kerrotaan ongelma alla olevalla tulosteella ja ohjelma lopetetaan `sys.exit(0)`-käskyllä. Mitään siivoustoimenpiteitä ei yritetä tehdä, koska poikkeuksen tarkka syy ja siten sopivat operaatiot eivät ole tiedossa

Tyypillisimmät virheilmoitukset ovat seuraavat

11. `"Tiedoston 'TiedostoNimi' käsittelyssä virhe, lopetetaan."`

Luvut asiat kokoava esimerkki

Esimerkissä 9.11 näkyy tiedostonkäsittelyyn liittyvä poikkeustenkäsittely sekä tiedostoa kirjoitettaessa että luettaessa.

Esimerkki 9.11. Poikkeustenkäsittely tiedostoa kirjoitettaessa ja luettaessa

```
import sys
LUKUJA = 10

def kirjoitaTiedosto(Nimi): # 1 kokonaisluku per rivi, lopussa tyhjä rivi
    print("Kirjoitetaan tiedosto '{0:s}'.".format(Nimi))
    try:
        Tiedosto = open(Nimi, "w")
        for i in range(LUKUJA):
            Tiedosto.write(str(i)+'\n')
        Tiedosto.close()
    except OSError:
        print("Tiedoston '{0:s}' käsittelyssä virhe, lopetetaan.".format(Nimi))
        sys.exit(0)
    print("Kirjoitettu tiedosto '{0:s}'.".format(Nimi))
    return None

def lueTiedosto(Nimi):
    print("Luetaan tiedosto '{0:s}'.".format(Nimi))
    try:
        Tiedosto = open(Nimi, "r")
        Rivi = Tiedosto.readline()
        while (len(Rivi) > 0):
            print(Rivi[:-1])
            Rivi = Tiedosto.readline()
        Tiedosto.close()
    except OSError:
        print("Tiedoston '{0:s}' käsittelyssä virhe, lopetetaan.".format(Nimi))
        sys.exit(0)
    print("Luettu tiedosto '{0:s}'.".format(Nimi))
    return None

def paaohjelma():
    TiedostoNimi = "L09E11.txt"
    kirjoitaTiedosto(TiedostoNimi)
    lueTiedosto(TiedostoNimi)
    print("Kiitos ohjelman käytöstä.")
    return None

paaohjelma()
##########################################################################
# eof
```

Liite x. Python 3.11 poikkeushierarkia, The class hierarchy for built-in exceptions