Protseduurne programmeerimine ja funktsioon

Funktsioon on protseduurse programmeerimiskeele vahend (ja protseduurse programmeerimise põhimõte) programmi loogika organiseerimiseks kergemalt hallatavateks väiksemateks osadeks. Üks oluline eelis on võimalus sama programmikoodi korduvalt käivitada ja sel viisil hoida kogu programmi kood lühem. Lisaks saab jõudu ja aega kokku hoida sellelt, et vajaduse korral koodi muuta saab seda teha vaid ühes kohas ja ei pea otsima muudatuste kohta kogu koodist.

Alamprogrammiga / funktsiooniga seondub 2 olulist toimingut:

1. funktsiooni kirjeldamine ehk defineerimine – mida funktsioon teeb ja kuidas ta seda teeb
2. funktsiooni väljakutsumine ehk käivitamine – millal funktsioon peab tööle hakkama ja milliste algandmetega

Mõistetest üldiselt on juttu 4. loengu materjalides. Oluline on, et funktsioon peab olema enne kirjeldatud, kui teda välja kutsuda ehk kasutada saab. Ja enamasti kehtib see ka juhul, kui ühes programmifailis on mitme funktsiooni kirjeldused. Kui funktsioon A kutsub välja funktsiooni B, siis peab funktsiooni B kirjeldus olema failis eespool A omast.

Et funktsioonide kasutamine muudab programmilausete kirjas olevat täitmise järjekorda (ei alustata faili esimesest reast ja ei liiguta faili lõpuni), siis selliste programmide lugemisel on kasulik järgida tegelikku täitmisvoogu *(flow of execution)*. Sellest omakorda tuleneb vajadus, et mitte öelda kohustus, anda funktsioonidele sisu kirjeldavad nimed. Lugedes funktsiooni välkakutse puhul `LeiaKeskmine(jada)`, ei peagi ehk funktsiooni ennast uurima asuma, vaid üldise pildi saamiseks piisab ka nime usaldamisest.

Funktsioonid Pythonis

Pythonis ei eristata protseduuri ja funktsiooni kirjeldust. Funktsioon tagastab traditsiooniliselt ühe väärtuse pärast andmete töötlemist (seda saab funktsiooni sees käsuga organiseerida). Kui funktsioonist ei tagastata väärtust, siis ongi tegemist sisuliselt protseduuriga.

Funktsiooni kirjeldamine

Funktsiooni kirjeldamiseks on `def`-lause:

```
def funktsiooni_nimi(parameetrite loetelu):
    „Funktsioooni kirjeldus inimkeeles“
    Funktsiooni sisu
```

def – keele sõna, mis alustab funktsiooni kirjeldust

funktsiooni_nimi – programmeerija valitud nimi, mille kaudu funktsiooni hiljem välja kutsutakse (ehk käivitatakse); seega peaks nimi olema mõistlikult ja funktsiooni sisule vastavalt valitud

parameetrite loetelu – algandmed, mida funktsioon töötlema hakkab ja mis talle väljakutse kohast ette antakse (nt arv, mida juurima hakata)

Eelnev kolmik moodustab funktsiooni **päise** *(header)*.

„kirjeldus ...“ - string (nn *docstring*), mille abil kirjutatakse üles funktsiooni olulised omadused ja tegevused ehk mille jaoks see funktsioon kasulik on, mida ta leiab, väga soovitatav lisada; nendest saab automaatselt dokumentatsiooni genereerida

funktsiooni sisu – kood, mis tegelikult funktsiooni tegevuse määrab (nt programmi kood, mille abil arvust tema juur arvutatakse); kogu funktsiooni sisu peab olema kirjutatud taandega (ja võib sisaldada järgmisi taandeid, kui keele laused seda nõuavad.

Lisaks saab funktsioonis kasutada `return`-lauset, mille abil tagastatakse funktsioonist vastus (nt leitud ruutjuur):

```
return tulemus
```

Võib kirjutada ka ainult `return`, sel juhul tagastatakse tühi väärtus `None`.

Peale `return`-lauset olevat koodi ei täideta kunagi, sest lisaks vastuse tagastamisele on see ka käsk funktsiooni töö lõpetamiseks.

Reeglina peavad funktsioonid olema enne kirjeldatud, kui neid kasutama hakata saab. Seega paiknevad funktsioonide kirjeldused programmi alguses (funktsioone võib seal mitu tükki olla). Kui üks funktsioon teist funktsiooni

väljakutsuma peab, siis tuleb ka nende omavahelist järgnevust arvestada – väljakutsuja pärast väljakutsutavat. St kui hakata programmi koodi algusest lugema, siis jõudes funktsiooni väljakutseni peaks selge olema, mida ja kuidas see funktsioon teeb.

Funktsiooni väljakutsumisel luuakse uus sümbolite (nt muutujate nimede) tabel. St tekivad antud **funktsiooni lokaalsed muutujad**. Samasse tabelisse satuvad ka funktsiooni parameetrid. Kui sama funktsioon teist korda välja kutsutakse, luuakse uus tabel. Funktsioon võib küll kasutada peaprogrammi muutujaid, kuid nende väärtuseid ta muuta ei saa (täpsustuseks – võib proovida, kuid see ei mõju, sest luuakse lihtsalt uus funktsiooni lokaalne muutuja).

Funktsiooni saab ka teise funktsiooni sees kirjeldada, kuid sel juhul on ta kättesaadav ja väljakutsutav ka vaid selle sama funktsiooni seest.

Funktsiooni parameetrid

Parameetrite põhiülesanne on lähteandmete toomine funktsiooni. Kõige tavalisem parameetrite edastamise viis on nö **järjekorra alusel** *(Positional Arguments)* – funktsiooni kirjelduses ja funktsiooni väljakutses on formaalsed ja tegelikud parameetrid samas järjekorras. Sel juhul peab formaalsete ja tegelike parameetrite arv olema vastavuses.

Pythonis võib parameetrite arv olla ka muutuv. Ja seetõttu on veel mõned võimalused nende edastamiseks.

Parameetritele saab määrata **vaikeväärtused** *(Default Arguments)*. Seda tehakse funktsiooni kirjelduses omistusmärgi abil:

```
def astenda(arv, aste = 2):
```

See tähendab, et kui teist parameetrit (aste) funktsiooni väljakutses ei määrata, siis on astme väärtuseks 2. Väljakutse võib aga olla järgmine:

```
astenda(a1)       # a1 tõstetakse ruutu
astenda(a2,3)     # a2 tõstetakse kuupi
```

Kohustuslikud parameetrid tuleb panna esimesteks ja vaikeväärtustega parameetrid viimasteks.

Vaikeväärtuste kasutamine võib hõlbustada programmeerija tööd. Üks näide nende kasutamisest on mitmesuguste konstantsete väärtuste pruukimine – näiteks tulumaksu, käibemaksu ja tont-teab-veel-mis-maksu määrad. Kui funktsiooni kirjeldaja on need väärtused ära määranud, siis ei pea teine programmeerija nende arvude pärast oma pead vaevama, kui just väga vaja ei ole. Alternatiiviks on vastavate konstantide määramine funktsiooni sees.

Parameetreid võib käsitleda kui **võtmesõnu** *(Keyword Arguments)* ja kasutada väljakutse lauses formaalse parameetri nime koos talle omistatava väärtusega (väärtuseks võib olla ka muutuja). Näiteks on lubatud järgmised funktsiooni väljakutsed:

```
astenda(arv = a1, aste = 2) või
astenda(aste = 2, arv = a1)
```

(NB! Formaalsel ja tegelikul parameetril ei pruugi olla sama nimi, sest sama funktsiooni peab saama rakendada erinevatele väärtustele). Võtmesõnade kasutamisest võib olla kasu, kui on vaja osa parameetreid vahele jätta või muidu ei suudeta tuvastada, millises järjekorras nad olema peavad. Teinekord võib see tõsta ka programmi loetavust – on näha, millisesse parameetrisse milline väärtus pannakse, sest paljude parameetrite korral hakkab järjekord segamini minema.

Näide (funktsioon1.py):

```
# Funktsiooni algus
def kompliment (nimi):
    # Järgnev string on docstring (dokumentatsiooni genereerimiseks)
    "Funktsioon teeb komplimendi ilusa nime kohta"
    print("Tere ", nimi)
    print("Sul on väga armas nimi!")
# Funktsiooni kompliment() lõpp

# Peaprogrammi algus
print("Aga siit algab tegelikult programmi täitmine!")
eesnimi = input("Mis Su eesnimi on? ")
# Funktsioon kutsutakse välja kaks korda erinevate parameetritega
kompliment(eesnimi)
perenimi = input("Ja perekonnanimi? ")
kompliment(perenimi)
```



```
print("Oli tore Sinuga kohtuda!")
# Peaprogrammi lõpp
```

Muutujate kehtivuspiirkond ehk skoop

Muutuja skoop *(Variable Scope)* on osa programmist, kus muutuja deklaratsioon kehtib ehk kus muutuja on nähtav. Skoobi probleemid on olulised ja nendega tuleb osata arvestada. Nagu üldiselt, nii ka Pythonis saab muutujal olla lokaalne või globaalne skoop.

Lokaalsed muutujad kuuluvad reeglina mõne funktsiooni juurde ja on lokaalsed selle funktsiooni suhtes. Nad tekivad funktsiooni töö käigus ja kaovad siis, kui funktsioon töö lõpetab. **Globaalsed muutujad** „elavad“ seni kuni kogu programm töötab ja neid on võimalik kätte saada suvalisest funktsioonist, mis selle programmi käigus käivitatakse.

Nime otsides käitub Python järgmiselt (täitsa tavaliselt) – kõigepealt otsitakse nime lokaalsete nimede hulgast, kui sealt ei leitud, siis globaalsete hulgast ja kui sealt ka ei leitud, siis tekib viga NameError. Mis juhtub kui lokaalne ja globaalne muutuja kannavad sama nime? Lähtudes eelnevalt kirjeldatud loogikast saadakse enne kätte lokaalne muutuja ja tema väärtusega ka tegelema hakatakse.

Globaalsete muutujate tekitamiseks tuleb nad kirjeldada võtmesõna `global` kasutades. Lause `global` lisatakse funktsiooni sisse, enne vastava muutuja kasutamist (mõistlik oleks lisada funktsiooni algusesse kõigi kasutatavate globaalsete muutujate kohta, et globaalseid muutujaid sel viisil paremini üles leida).

```
global muutuja1 [, muutuja2, ...]
```

Samas peab kirjeldatav muutuja olema peaprogrammis deklareeritud (kasutusele võetud), muidu tekib viga. Siiski **globaalsete muutujate kasutamine ei ole hea komme**, eriti mitte nende muutmine. Täpsustuseks veel niipalju, et `global`-deklaratsiooni kasutamata on peaprogrammi muutujad nähtavad, st neid saab lugeda. Muudetavaks muutuvad nad aga peale deklareerimist. Kokkuvõttes – parem kui seda võimalust ei kasutata.

Moodulid

Moodulite kasutamine on meetod selleks, et programmi koodi loogiliselt organiseerida. Kui programmeerimiskeel lubab kasutada mooduleid, siis reeglina saab ühe programmi koodi jaotada füüsiliselt mitmesse faili ja sellega kogu koodi hallatavamaks muuta. Moodulid võimaldavad ka koodi **korduvkasutada** *(reuse)*, mis tarkvaratootmises on väga oluline. Ühte moodulisse saab koguda mitmed funktsioonid (mis tavaliselt omavahel loogiliselt seotud on). Moodulid võivad sisaldada ka klasse koos klasside atribuutide ja meetoditega, kui rääkida ja tegutseda objekt-orienteeritult. Moodulis olevaid funktsioone jms saab kasutada, kui moodul importida (mäletad, import math jms?)

Moodul on funktsioonide organiseerimine loogilisel tasemel ja füüsilisel tasemel moodustab moodul ühe faili. Pythoni moodul kannab laiendit `.py`, nagu tavaline programmgi. Moodul on ülesehitatud nagu tavaline Pythoni skript / programm. Ta sisaldab funktsioone ja võib lisaks sisaldada ka nö peaprogrammi, milles olevad laused käivitatakse üks kord mooduli kasutusele võtmisel. Need laused võivad näiteks algväärtustada mingeid muutujaid.

Moodulit saab kirjutada **bait-koodi** *(byte-compiled)*, siis saab tema laiendiks `.pyc`. See muudab mooduli laadimise veidi kiiremaks ja võimaldab mooduleid edastada nii, et nende sisu ei ole programmi tekstina loetav (ja muudetav).

Nimeruum

Nimeruum *(namespace)* on mõiste, mille abil luuakse ühesed seosed objektide ja nende nimede vahel. Reeglina moodustab moodul ühe nimeruumi. Peale mooduli importimist oma programmi kutsutakse tema funktsioone välja: `mooduli_nimi.funktsiooni_nimi()`

See määrab ära üheselt, millist funktsiooni kasutada. Erinevates moodulites võivad funktsioonide nimed korduda, kui programmis ei saa korraga kasutada mitut sama nimega moodulit.

Lisaks moodulite nimeruumidele räägitakse lokaalsest nimeruumist ja globaalsest nimeruumist. Näiteks konkreetses programmis kasutatavad muutujad satuvad lokaalsesse nimeruumi ja lokaalne nimeruum muutub, sest funktsioonide väljakutsed muudavad seda.

Nimeruum – see on üksühese vastavuse loomine objekti ja nime vahel: millises nimeruumis on milline objekt?

Skoop – see on objektide / muutujate nähtavus programmi täitmise konkreetsel hetkel: mida ma praegu näen? Kas ma seda näen?

Näide moodulist:


```
# -*- coding: ISO-8859-4 -*-
# Moodul "teisenda"
# Moodul sisaldab funktsioone erinevate ühikute vaheliste teisenduste jaoks

def inch2cm(yhikud):
    "Funktsioon teisendab sisendiks olevad tollid sentimeetriteks"
    tulemus = yhikud * 2.54
    return tulemus

def cm2inch(yhikud):
    "Funktsioon teisendab sisendiks olevad sentimeetrid tollideks"
    tulemus = yhikud * 1/2.54
    return tulemus
```

Ja moodulit kasutavast programmist:

```
# Programm, mis proovib kasutada moodulit teisenda.py
import teisenda
arv = eval(input("Sisesta suurus sentimeetrites "))
print("Teisenduse tulemus on",teisenda.cm2inch(arv),"tolli.")

arv = eval(input("Sisesta suurus tollides "))
print("Teisenduse tulemus on",teisenda.inch2cm(arv),"sentimeetrit.")
```

Eelnevad näited paiknevad veebis näidete kataloogis nimede `moodul1.py` ja `teisenda.py` all.

