

Protseduurne programmeerimine ja funktsioon

Alamprogramm on protseduurse programmeerimiskeele vahend (ja protseduurse programmeerimise põhimõiste) programmi loogika jaotamiseks kergemini hallatavateks väiksemateks osadeks. Üks oluline eelis on võimalus sama programmikoodi (alamprogrammi) korduvalt käivitada ja sel viisil hoida kogu programmi kood lühem ja paremini jälgitav.

Mõistetest üldiselt (nt mis on **protseduur** ja mis on **funktsioon**) saab lugeda materjalis „Alamprogramm kui protseduurse programmeerimise alus“. Et Pythoniga on seotud funktsioon, siis edasises jutus on kasutusel just see mõiste.

Oluline on, et funktsioon peab olema enne kirjeldatud, kui teda välja kutsuda ehk kasutada saab. Tavaliselt kehtib see ka juhul, kui ühes programmifailis on mitme funktsiooni kirjeldused. Kui funktsioon A kutsub välja funktsiooni B, siis peab funktsiooni B kirjeldus paiknema failis A kirjeldusest eespool. Pythonis siiski see kitsendus puudub. Veelgi enam – Pythonis võib funktsioonid ja peaprogrammi kõik segiläbi kirja panna. See siiski ei ole aksepteeritav käitumine, sest sel viisil ei muutu kood paremini loetavaks.

Funktsiooniga seondub kaks olulist toimingut:

1. funktsiooni kirjeldamine ehk defineerimine – funktsiooni kood.
2. funktsiooni kutsumine ehk käivitamine – millal ja milliste algandmetega funktsioon peab tööle hakkama.

Et programmi koodi jaotamine funktsioonidesse muudab programmilause kirjas olevat täitmise järjekorda (ei alustata faili esimesest reast ja ei liiguta faili lõpuni), siis selliste programmide uurimisel on kasulik järgida tegelikku täitmisvoogu (*ingl flow of execution*). Sellest omakorda tuleneb vajadus, et mitte öelda kohustus, anda funktsioonidele sisu kirjeldavad nimed. Luges funktsiooni väljakutset `LeiaKeskmine(jada)`, ei peagi alati funktsiooni ennast lähemalt uurima, vaid üldise pildi saamiseks piisab ka funktsiooni nime usaldamisest – listis `jada` olevatele arvudele leitakse keskmine (ehkki siinkohal oleks kasulik täpsustada, et see on näiteks aritmeetiline keskmine).

Funktsioonid Pythonis

Pythonis ei eristata keeleliselt protseduuri ja funktsiooni kirjeldust. Funktsioon tagastab traditsiooniliselt peale andmete töötlemist ühe väärtuse (selleks on vastav käsk `return`). Kui funktsioonist ei tagastata väärtust, siis ongi tegemist sisuliselt protseduuriga – ta täidab lihtsalt mingi osa tegevuslauseid.

Funktsiooni kirjeldamine

Funktsiooni definitsioon ehk kirjeldus algab keelesõnaga `def`:

```
def funktsiooni_nimi(parameetrite_loetelu):  
    """Nn docstring: funktsiooni lühikirjeldus inimkeeles, parameetrid ja  
    tagastatav väärtus"""  
    Funktsiooni_sisu
```

def – keele sõna, millega algab funktsiooni kirjeldus

funktsiooni_nimi – programmeerija valitud nimi, mille kaudu funktsiooni hiljem välja kutsutakse (ehk käivitatakse); seega peab nimi olema mõistlikult ja funktsiooni tegevusele vastavalt valitud. Nimes võib kasutada samu märke, mida muutujate nimedeski (samad reeglid kehtivadki üldiselt igasugustele programmeerija poolt kasutusele võetavatele nimedele ehk identifikaatoritele (*ingl identifier*)). Sama on ka piirangutega: ei tohi kasutada programmeerimiskeele sõnu (ehk nn reserveeritud sõnu), mõistlik pole kasutada keelega kaasas olevate (standard)funktsioonide nimesid. Viimasel juhul muutub vastav standardfunktsioon mittekasutatavaks.

parameetrite_loetelu – need on nime järel sulgudes: algandmed, mida funktsioon töötlema hakkab ja mis talle väljakutse käigus ette antakse (nt arv, mille ruutjuurt leida). Kui sisendandmed puuduvad, lisatakse tühjad sulud.

Eelnev kolmik moodustab funktsiooni **päise** (*header*), mis lõpeb semikooloniga. Järgneb **funktsiooni keha** (*body*), mis peab olema taandatud / trepitud.

„**kirjeldus** ...“ - string (nn *docstring*), mille abil selgitatakse funktsiooni olulised omadused ja tegevused ehk mille jaoks see funktsioon kasulik on, mida ta leiab, väga soovitatav lisada; nendest saab automaatselt dokumentatsiooni genereerida: näiteks käsuga `funktsiooni_nimi.help()` või `print(funktsiooni_nimi.__doc__`

funktsiooni_sisu – kood, mis tegelikult funktsiooni tegevuse määrab (nt programmi kood, mille abil arvust ruutjuur

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`.

Lausele `return` järgnevat koodi ei täideta kunagi, sest lisaks vastuse tagastamisele on see ka käsk funktsiooni töö lõpetamiseks. Funktsioonis võib olla ka mitu `return`-lauset, näiteks selleks, et `if`-lauses määrata erinevad tagastatavad tulemused.

Reeglina peavad funktsioonid olema enne kirjeldatud, kui neid kasutama hakata saab. Pythonis küll keeleliselt seda ei nõuta, kuid selline paigutus annab koodile parema ja loetavama struktuuri.

Funktsioonikutsel luuakse uus sümbolite (nt muutujate nimede) tabel, st tekivad antud **funktsiooni lokaalsed muutujad** (*ingl local variable*). Samasse tabelisse satuvad ka funktsiooni parameetrid. Kui funktsioon töö lõpetab, kaovad ka lokaalsed muutujad. Sama funktsiooni uuel väljakutsel 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 vaid selle sama funktsiooni seest.

Funktsiooni parameetrid ehk argumendid

Parameetrite põhiülesanne on lähteandmete toomine funktsiooni. Kõige tavalisem parameetrite edastamise viis on nõ **järjekorra alusel** (*ingl 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** (*ingl default argument*). Seda tehakse funktsiooni kirjelduses omistumärgi 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(arv1)      # arv1 tõstetakse ruutu  
    astenda(arv2,3)    # arv2 tõstetakse kuubi
```

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** (*ingl keyword argument*) 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.

Muutujate kehtivuspiirkond ehk skoop

Muutuja skoop (*ingl variable scope*) on osa programmist, kus muutuja deklaratsioon kehtib ehk kus muutuja on nähtav. Skoobi reeglid 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** tekivad peaprogrammis ja „elavad“ seni, kuni kogu programm töötab. Neid näeb suvalisest funktsioonist, mis selles programmifailis kirjeldatud ja programmi käigus käivitatud on.

Nime (muutuja nimne, funktsiooni nime) otsides käitub Python järgmisel, üsna tavapärasel viisil: 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 jõutakse enne lokaalne nimeni.

Muudetavate **globaalsete muutujate** tekitamiseks tuleb nad funktsioonis kirjeldada võtmesõnaga `global` enne vastava muutuja kasutamist. Mõistlik on kirjeldada funktsiooni alguses kõik globaalsed muutujad, et nad inimesele paremini leitavad oleksid.

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

Samas peab kirjeldatav muutuja olema peaprogrammis kasutuses, 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, kuid mitte muuta.

Kokkuvõttes on parem kui seda võimalust ei kasutata või kasutatakse üli kokkuhoidlikult.

Moodulid

Mooduleid kasutatakse programmikoodi organiseerimiseks. 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** (*ingl 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.

Moodul on funktsioonide organiseerimine loogilisel tasemel ja füüsilisel tasemel moodustab moodul ühe faili. Pythoni moodul kannab laiendit `.py`, nagu tavaline programmg. Moodul on ülesehitatud nagu tavaline Pythoni skript / programm. Ta sisaldab funktsioone ja võib lisaks sisaldada ka nõ 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 (*ingl 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, millisest moodulist millist funktsiooni kasutada. Erinevates moodulites võivad funktsioonide nimed korduda, kui programmis ei saa korraga kasutada mitut sama nimega moodulit.