Stiilijuhised Pythonis programmeerimiseks

Stiil hõlmab koodis kõike, mis ei oma tähendust arvuti jaoks, küll aga inimese jaoks, kes koodi loeb. Stiilivõtted teevad koodi loetavamaks. Mingil juhul ei tohi koodi vormistus, kasutatavad nimed ja kommentaarid olla vastuolus sellega, mida kood tegelikult teeb ja lugejat eksitada.

Kui koodiga töötab, või on võimalik, et tulevikus hakkab töötama, rahvusvaheline tiim, siis on soovitav nimedes ja kommentaarides kasutada inglise keelt.

Avaldised

Tehte või omistamismärgi ümber on avaldises soovitav kasutada tühikuid.

Näiteks `tulem = 2 + 3`. Kui tühikuid kasutada, siis kindlasti märgi mõlemal poolel. Näiteks `vanusepiir = 18` on hea stiil, aga `vanusepiir= 18` mitte.

Ilma tühikuteta stiili võib kasutada segamini tühikutega stiiliga tehteprioriteetide rõhutamiseks.

Näiteks `2*x + 3*y > 0`

Väga vale oleks vastupidine: `2 * x+3 * y>0`

Muutujate nimed

Pythonis on tavaks, et muutujate nimed algavad väiketähega (suurtähega algavad klasside nimed).

Muutujate nimed peavad olema sisulised ja informatiivsed. Näiteks `tulumaks`, mitte `x23`.

Nimed peaks võtma pigem probleemvaldkonnast, mitte programmeerimisloogikast. Näiteks `isikukood` on parem nimi kui `sisend`, kuigi programmiloogikas võib isikukood olla sisendiks.

Ühetähelised nimed on aktsepteeritavad, kui probleemvaldkonnas on kasutusel vastav matemaatiline notatsioon (näiteks füüsikas kiirus: `v`, aeg: `t`) või tegemist on väga üldise programmeerimisprobleemiga.

Mõned levinud ühetähelised nimed:

`n` – elementide koguarv

`i` – elemendi indeks

`s` – string

`d` – sõnastik (*dictionary*)

`f` – fail.

Kui nimi koosneb mitmest sõnast, siis on Pythonis tavaks eraldajana kasutada allkriipsu. Näiteks `firma_auto`. Alternatiivne *mixedCase* kuju alustab uut sõna suurtähega, näiteks `firmaAuto`. Valitud stiili tuleks kasutada läbivalt.

Kaks muutujat ei tohiks olla liiga sarnaste nimedega. Kui meil on näiteks kaks muutujat `maatriks` ja `maatrix`, siis on kerge eksida.

Sama muutujat ei tohi kasutada erinevatel eesmärkidel. Muutuja `firma_auto` ei tohiks hakata mingist hetkest auto asemel viitama firma töötajale.

Selgete nimedega abimuutujaid on hea sisse tuua keerukate avaldiste sisu selgitamiseks.

Näiteks

```
aastaintress = kuuintress**12
koguintress = aastaintress**laenuperiood
makstav_summa = koguintress * laenusumma
```

on selgem, kui

```
makstav_summa = (kuuintress**12)**laenuperiood * laenusumma
```

.

Tõeväärtusmuutuja peab olema sellise nimega, et oleks selge, mida selle tõene/väär väärtus tähendab ja et see oleks if-lauses ladusalt loetav. Näiteks `if isComplete` on palju selgem kui `if status`.

Tõeväärtusmuutuja ei tohiks olla eitus. Muutuja `notComplete` võib viia kummalise loogikaavaldiseni `not notComplete`. Samas `isComplete` ja `not isComplete` on mõlemad hästiloetavad.

Järjendite nimed võiksid olla kas mitmuses või lõppeda sufiksiga `list` või `dict`. Näiteks `car_list` või `cars`, mitte `car`, kui meil on tegemist autode hulgaga.

Funktsiooni poolt tagastatavale väärtusele vastava muutuja nimeks funktsiooni kehas võiks olla `result` (`tulem`). Sisuline nimi oleks siin funktsiooni nimi, aga see on juba defineeritud.

Kommentaarid

Kommentaare tuleks kasutada mõõdukalt. Liigsed ja mõttetud kommentaarid teevad koodi raskelt loetavaks. Kommentaare tuleb koodiga kooskõlas hoida ja liigne hulk kommentaare tähendab lisatööd koodi muutmisel. Kindlasti ei tohiks kommentaar koodiga vastuolus olla. Muutmata jäänud kommentaar on eksitav ja kahjulik.

Üherealised (#)

Kommentaar peaks olema täislause.

Kommentaar ei tohi korrata koodi teiste sõnadega.

Näiteks `i = i + 1 # Suurenda i-d ühe võrra` on sobimatu stiil.

Kommentaarid peaks olema koodist kõrgemal tasemel, andma lühikokkuvõte toimuvast ja selgitama, mis eesmärki kommentaarile järgnev kood täidab.

Näiteks:

```
# Joonista majakarp
turtle.forward(200)
turtle.left(90)
turtle.forward(300)
turtle.left(90)
turtle.forward(200)
# Joonista katus
turtle.left(45)
turtle.forward(212)
turtle.left(90)
turtle.forward(212)
```

Moodulite import

Moodulite importimine peab toimuma faili alguses.

Kõigi funktsioonide ja klasside importimine moodulist kujul `from <module> import *` ei ole üldjuhul soovitatav, kuna seejärel on ebaselge, millised nimed on kasutusel.

Treppimine

Treppimiseks kasutada alati Pythoni standardit ehk nelja tühkut ühe taseme kohta. IDLE vaikeseadistuses seda kasutataksegi ja Tab klahv lisab neli tühikut.

Treppimine, mida Pythoni süntaks nõuab, vastab üldisele heale programmeerimisstiilile.

Tingimuslaused

Tingimuslause peaks algama normaaljuhuga ja lõppema erijuhtudega.

Vaata, et võrdsuse alusel hargnemine tingimuses oleks õige. Võrdluste < ja <= äravahetamine on sagedane viga.

if-plokk peaks sisaldama mõtekaid lauseid. Kood stiilis:

```
if tingimus:
    pass
else:
    tee_midagi()
```

tuleks asendada koodiga:

```
if not tingimus:
    tee_midagi()
```

Lihtsusta keerukas tingimusavaldis abimuutujate ja abifunktsioonide abil.

Tsüklid (for, while)

Kui tsükkel põhineb mingi jada elementide järjest läbikäimisel või mingi muutuja lihtsal inkrementeerimisel, siis tuleks kasutada *f*or-tsüklit.

Ära modifitseeri for-tsüklis läbikäidavat jada!

```
for element in jada:
    if mingi tingimus:
        jada.append(uus_element)
    …
```

Selline kood on väga veaohtlik ja segane.

Kuigi üldise või matemaatilise valdkonna korral võib *i* for-tsükli indeksi nimena olla sobiv, tuleks parema ja sisulisema termini olemasolul kasutada seda terminit. Näiteks `for rida in sisendfail` on selgem kui `for i in sisendfail`.

Kui tsükli sees on üle kolme taseme tsükleid ja tingimuslauseid, siis tuleks kaaluda lihtsustamist abifunktsioonide sissetoomise abil.

Kui programmis on järjestikused dubleerivad tegevused, siis tuleks dubleerimisest vabaneda tsükli abil.

Funktsioonid

Programmi tükeldamine funktsioonideks on äärmiselt oluline keerukuse haldamise võte. Funktsioonid peaksid vastama selgelt defineeritud alamtegevustele. Funktsiooni kasutavas koodis on see alamtegevus asendatud funktsiooni väljakutsega, mis võimaldab funktsiooni kasutajal ignoreerida küsimust *kuidas* vastav alamtegevus on üles ehitatud ja keskenduda küsimusele *mida*

funktsioon teeb. Hästi defineeritud funktsiooni kasutamiseks peab piisama funktsiooni nimest, parameetrite loetelust (funktsiooni signatuur) ja dokumentatsioonistringist. Kokku võib seda nimetada **funktsiooni spetsifikatsiooniks**. Dokumentatsioonistring peaks kirjeldama funktsiooni lepingut: funktsiooni eeltingimusi ja järeltingimusi. **Eeltingimused** kirjeldavad vajalikku seisundit enne funktsiooni väljakutsumist, mida funktsiooni väljakutsuja peab tagama: parameetrite tüübid ja lubatud väärtused kui funktsioon kasutab globaalmuutujaid, siis ka eeldused nende kohta. **Järeltingimused** kirjeldavad funktsiooni väljundit ja/või muudatusi globaalmuutujates olekus ning muid funktsiooni täitmise tulemusel toimunud muudatusi (failidesse kirjutamine, joonistamine,...).

```
def joonista_trikoloor(riik, riikide_sonastik):
    """
    Joonistab kilpkonnagraafikaga riigi trikoloori. - Järeltingimus

    Parameetrid ja eeltingimused:
    riik on riigi nimetus stringina. Peab sisalduma võtmena riikide sõnastikus.
    riikide_sonastik on sõnastik kujul {riik: (varv1, varv2, varv3),...}
    """
```

Funktsiooni koodi (keha) vaatamiseks ei peaks kasutajal vajadust olema. Kindlasti tuleks defineerida funktsioon siis kui meil on võimalik sinna kokku panna dubleerivad alamtegevused programmis. Lisaks teeb programmi tükeldamine funktsioonideks lihtsamaks programmi testimise. Funktsioonil peab olema üks ja selge ülesanne.

Funktsiooni realisatsiooni üldideed, mis on programmeerimiskeelest sõltumatu ja kõrgemal tasemel nimetame **algoritmiks.** Algoritmi võib esitada koodis kommentaaridena, millele järgnevad vastavad koodi käsud (vt ka kommentaarid).

```
# Leia riikide_sõnatikust võtme riik abil värvide kolmik varvid
varvid = riikide_sonastik[riik]
# Iga värvi v jaoks  värvide hulgast:
for v in varvid :
    # Joonista värviga v täidetud ristkülik
    ...
    # Nihuta kilpkonn selle ristküliku vasakusse alanurka
    ...
```

Kui koodi plokk vajab kommentaari, siis tasub kaaluda selle ploki asendamist funktsiooni väljakutsega. Kindlasti tuleks seda kaaluda juhul kui selline funktsioon oleks üldine ja korduvkasutatav. Näiteks `joonista_ristkülik(kylg1,kylg2,v)` oleks korduvkasutatav alamfunktsioon.

Funktsioonide nimede täpsus ja selgus on üks tähtsaim programmi selguse ja loetavuse määraja. Funktsiooni nimi peab olema antud funktsiooni kasutaja, mitte funktsiooni realiseerija seisukohast, peab kirjeldama *mida* funktsioon teeb, mitte *kuidas* funktsioon on sisemiselt üles ehitatud. Näiteks `joonista_spiraal()` on hea funktsiooni nimi,

`tsükkel_pööramisega()` on väga halb funktsiooni nimi. Kui funktsiooni nimi on halb, siis on kogu funktsioon halb: kas on nimi ebatäpne ja vajab muutmist või on nimi täpne ja funktsioon on olemuslikult segase sisu ja eesmärgiga.

Kui funktsioon ei tagasta väärtust, vaid muudab mingit objekti, siis on funktsiooni nimi tüüpiliselt *verb-objekt* stiilis: joonista_spiraal(), loe_sagedused(), transponeeri_maatriks().

Kui funktsioon tagastab väärtuse, siis peab funktsiooni nimi kirjeldama tagastatavat väärtust: `sageduste_sõnastik()`, `pikkus()`, `max_element()`.

Kui funktsiooni on võimalik sõnastada üldisemalt, paindlikumalt ja korduvkasutatavamalt, ilma funktsiooni oluliselt keerukamaks tegemata, siis tuleks seda teha. Näiteks funktsioon, mis tagastab mingi väärtuse on paindlikum ja korduvkasutatavam kui funktsioon, mis selle väärtuse välja prindib. Printimise võib vajadusel alati funktsiooni väljakutsuvas koodis lisada, prinditud väärtust muuks otstarbeks lugeda on keeruline. Samuti on funktsiooni vaikeväärtusega parameeter paindlikum kui funktsiooni sisse kirjutatud vaikeväärtus.

Pythonis on tavaks alustada funktsioonide nimesid väikese algustähega. Sama kehtib ka funktsiooni parameetrite nimede kohta.

Olulisemate ja korduvkasutuseks mõeldud funktsioonide juures peaks olema ka dokumentatsioonistring, mis kirjeldab, mida funktsioon teeb ja millises olukorras on lubatud funktsiooni välja kutsuda.

Kui `global` ja `nonlocal` muutujate kasutamine jätab funktsiooni kasutaja jaoks segaseks, milliseid andmeid muudetakse ja mida funktsioon teeb, siis on tuleks neist hoiduda. Sama kehtib ka muutujate kasutamise kohta, mis ei ole funktsiooni lokaalmuutujad või parameetrid. Vajadusel tuleks sellised andmed parameetrina ette anda. Mooduli muutujate manipuleerimine ja kasutamine funktsioonide sees on sobilik juhul kui mooduli muutujad ja funktsioonid moodustavad läbimõeldud terviku ja nendevahelised sõltuvused on mooduli kasutajale selgelt dokumenteeritud. Näiteks mooduli `turtle.py` funktsioonid sõltuvad mooduli globaalsest olekust – kilpkonna värv, asukoht, suund jne – aga selline sõltuvus on selge ja vastavate muutujate kontrolliks ja muutmiseks on moodulis vastavad funktsioonid: `heading()`, `setheading()`, `fillcolor()` jne. Sarnane andmete ja nendega seotud funktsionaalsuse kombineerimine on objektorienteeritud programmeerimise aluseks, mida selles aines pikalt ei käsitleta.

Viiteid ja lisalugemist

[1] van Rossum, G. *PEP 8 -- Style Guide for Python Code,* [WWW] http://www.python.org/dev/peps/pep-0008/ (04.02.2014)

[2] McConnell, S. *Code Complete, Second Edition (2004)*. Microsoft Press, Redmond, WA, USA.