E-kursuse

"Programmeerimine lingvistidele"

materjalid

Aine maht 6 EAP

Neeme Kahusk (Tartu Ülikool), 2010

Programmeerimine lingvistidele

Sisukord

Sissejuhatus

Käivitamine käsurealt

Kus on Python?

Eeldusel, et Python on korralikult installeeritud, peaks ta käivituma käsurealt:

```
>python
```

Pythoni interaktiivse seansi alustamine ja lõpetamine

Käsurealt käivitamiseks antakse käsk:

```
>python
```

Windowsis Start -> Programs -> Python -> Python (command line)

Pythoni interpretaatorist väljumiseks kasutatakse faili lõpumärki (Ctrl-D). Windowsis Ctrl-Z ja Enter.

Käivitub interaktiivne Pythoni interpretaator. Ekraanil on näha midagi taolist nagu:

```
Python 2.5.2 (r252:60911, Mar 28 2010, 16:52:39) [GCC 4.3.1
20080507 (prerelease) [gcc-4_3-branch revision 135036]] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Igal real võib olla kas käsk või kommentaar. Kommentaar algab trelliga (#).

Aritmeetlised tehted

```
>>> 2+3 5
>>> # Võime kirjutada vahele mingi kommentaari
>>> 2*3 # kommentaar samal real
6
```

Kui palju on 1 + 2 x 2? Python arvestab tehete järjekorda. Vajaduse korral saab kasutada sulgusid:

```
>>>1 + 2 * 2
5
>>>(1 + 2) * 2
6
```

Kas Python jagab valesti?

```
>>>5 / 3
1
```

Ei. Kui tehtes on täisarvud, siis antakse ka täisarvuline vastus. Seda mitte ei ümardata, vaid "lõigatakse ära" kõik kohad, mis jäävad peale koma. Proovime nii, et vähemalt üks avaldise liige oleks antud suurema täpsusega:

```
>>>5.0 / 3
1.6666666666666667
```

Vahet ei ole, kas lisada koma-null jagatavale või jagajale:

```
>>>5 / 3.0
1.6666666666666667
```

Jagamisele järgneva liitmise puhul see enam ei mõju:

```
>>>5 / 3 + 5.0
6
```

Kui on tegemist kompleksoperaatoriga, kus operaatori sisu kirjutatakse järgmisele reale (järgmistele ridadele), siis tuleb kasutada taanet. Operaatori lõpus jääb üks rida tühjaks (interpretaator ei tea muidu, millal operaator lõpeb):

```
>>> for i in [0,1,2]:
...     print "Tere, maailm!"
...
Tere, maailm!
Tere, maailm!
Tere, maailm!
```

Nii lühikest käsku võib kirjutada ka ühele reale:

```
>>> for i in [0,1,2]: print "Tere, maailm!"
...
Tere, maailm!
Tere, maailm!
Tere, maailm!
```

...aga üldiselt ei peeta seda heaks stiiliks, kui liitkäsk on ühel real, siis on seda raskem pärast lugeda ja parandada.

Interpretaatoris kirjutatu läheb kaotsi. Kui on vaja programmi, mis kestaks kauem kui ühekordseks kasutamiseks, on mõistlik see kirjutada faili.

Algtekst failis

Kui meil on vaja pikemat programmi kui paar rida, kus arve liita-lahutada või sõnu hakkida, siis on meil tarvis programm kuidagi kirja panna, faili salvestada et teda järgmine kord kasutada.

Põhimõtteliselt on võimalik Pythoni programmiteksti kirjutada suvalise tekstiredaktoriga, kuid parematel on võimalus automaatselt teksti treppida, süntaksi esile tõsta ja muidki trikke teha.

Linuxi all on headeks vahenditeks Emacs ja Xemacs, ka Vi-l on olemas Pythoni redigeerimise keskkond. Väga paljud Linuxi all kasutatavad redaktorid toetavad Pythonit.

Ka Windowsis on palju vahendeid Pythoni tekstide redigeerimiseks.

Ülevaate Pythoni jaoks sobivatest redaktoritest saab aadressilt
http://wiki.python.org/moin/PythonEditors

Üldise tava kohaselt on Pythonis kirjutatud programmide laiend `.py`.

Pythonis kirjutatud programmi saab käivitada käsurealt. Proovime lihtsa programmiga, mis kirjutab kolm korda “Tere maailm!”. Selleks avame tekstitoimeti ja kirjutame sinna programmi teksti:

```
for i in [1,2,3]: print "Tere maailm!"
```

Salvestame faili nimega `teremaailm.py`. Selleks, et vastloodud programmi käivitada, kirjutame käsureale:

```
$>python teremaailm.py
```

Vastuseks peaks programm meid tervitama kolm korda.

Unixi all (kehtib ka Linuxis ja Cygwinis) piisab ka sellest, kui käivitada `teremaailm.py` üksinda. Selleks on vaja faili esimeseks reaks kirjutada:

```
#! /usr/bin/env python
```

ja muuta fail käivitatavaks (vähemalt kasutajale):

```
$>chmod u+x teremaailm.py
```

Nüüd võime faili käivitada otse käsurealt:

```
$>./teremaailm.py
```

Andmetüübid

Aritmeetiliste tehete juures nägime, kuidas tulemus olenes sellest, millist tüüpi andmetega tehet tehti. Sellist olukorda võib mujalgi ette tulla.

Olgu meil kaks avaldist. Esimene:

```
1 + 1 = 2
```

ja teine:

```
1 + 1 = 10
```

Kas teine avaldis on vale? Ei ole, kui meil on tegemist kahendsüsteemiga. Kui meil on kolmas avaldis:

```
I + I = II
```

siis võime seda vaadelda kui liitmistehet rooma numbritega või kui kahe `I`-tähe liitmist, nagu toimub tähtede liitmine neljandas avaldises:

```
'A' + 'B' + 'B' + 'A' = 'ABBA'
```

Pythonis on kasutusel järgmised andmetüübid:

- tavalised täisarvud (ingl. *plain integers*)
- pikad täisarvud (ingl. *long integers*)
- ujukumaarvud (ingl. *floating point numbers*)
- kaheksandsüsteemi täisarvud (ingl. *base-8 integer*)
- kuueteistkümnendsüsteemi täisarvud (ingl. *base-16 integer*)
- kompleksarvud (ingl. *complex numbers*)
- sõned (ingl. *strings*)
- loendid (ingl. *lists*)

- hulgad (ingl. *sets*)
- korteežid (ingl. *tuples*)
- kujutused (ingl. *dicts*)
- failid (ingl. *files*)

Käesoleva kursuse raames on meile olulised tekstid, loendid, tavalised täisarvud, vähem ka kujutused ja failid.

Nagu eespool öeldud, võivad ühed ja samad tehted olenevalt andmete tüübist anda erinevaid tulemusi. Iga andmetüübi juures on toodud ka tabel nendega teostavatest tehetest koos oodatavate tulemuste seletusega.

Arvud

Tehted arvudega:

```
>>>2+2
4
>>> ...2*2
4
>>>2**2
4
```

Pythonis eristatakse järgmisi arvandmete tüüpe:

- tavalised täisarvud (ingl. *plain integers*)
- pikad täisarvud (ingl. *long integers*)
- ujukumaarvud (ingl. *floating point numbers*)
- kompleksarvud (ingl. *complex numbers*)

Tõeväärtuse andmeid (ingl. *boolean*) võib käsitleda täisarvude alatüübina. Tõeväärtusi `True` (tõene) ja `False` (väär) võib esitada ka kahendsüsteemi arvudena, vastavalt `1` ja `0`.

Täisarvud on (vähemalt) 32-bitise esitustäpsusega (sõltub masinast), pikad täisarvud on piiramatu esitustäpsusega. Ujukomaarvude täpsus sõltub masinast, millel töötatakse.

Pikkadele täisarvudele lisatakse lõppu `L`. Ujukomaarvud sisaldavad kümnendkoha eraldajat (Pythonis on selleks punkt `.`) või eksponendimärki (`e` ja kohtade arv).

Kui on vaja näidata, et on tegemist kaheksandsüsteemi arvuga, lisatakse sellele ette `0`:

```
>>> a = 010
>>> a
8
```

Kui on vaja näidata, et on tegemist kuueteistkümendsüsteemi arvuga, lisatakse sellele ette `0x` või `0X`:

```
>>> a = 0Xa4
>>> a
164
>>> b = 0xBc
>>> b
188
```

Tekst

Teksti on võimalik esitada ülakomade või jutumärkidega. Kui on vaja kasutada teksti sees sama märki, millega see piiratud on, siis varjatakse see kurakaikaga (\):

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn't'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn't," she said.'
'"Isn't," she said.'
>>>jama = '"Isn't," she said.'
>>>print jama
"Isn't," she said.
```

Teksti võib jagada mitmele reale. Kui on vaja programmi algtekstis (loetavuse huvides) tekitada mitu rida, mis tegelikult trükitakse üheks, siis kasutame kurakaika märki:

```
>>>tere = "tere tere\
...vana kere"
>>>print tere
tere terevana kere
```

Kui käsk pole lõpetatud, siis näitab interpretaator järgmisel real kolme punkti. Kui on vaja teksti lisada reavahetusi, siis märgitakse need kasutades reavahetuse tähist, mis on UNIXites (sh ka Linuxis) `\n`, Windowsi keskkonnas `\r\n`:

```
>>>tere = "tere tere\nvana kere"
>>>print tere
tere tere
vana kere
```

Kui `\n` järele jätta tühik, siis algab järgmine rida tühikuga:

```
>>>tere = "tere tere\n vana kere"
>>>print tere
tere tere
 vana kere
```

Etteantud vorminduse säilitab täht-tähelt — koos reavahetuste ja tühikutega — kolmekordsete ühe- või kahekordsete jutumärkide kasutamine:

```
>>>tere = """ tere tere
... vana kere"""
>>>print tere
 tere tere
  vana kere
```

Hulgad

Pythonis on eraldi andmetüüp hulkade jaoks. Hulk on järjestamata elementide kogum, milles ei ole kordusi.

```
>>> korv = ['õun', 'apelsin', 'õun', 'pirn', 'apelsin', 'banaan']
>>> puuvili = set(korv) # Loome ilma kordusteta hulga
>>> puuvili
set(['apelsin', 'pirn', 'õun', 'banaan'])
>>> 'apelsin' in puuvili # Leiame sisalduvuse
True
>>> 'kaalikas' in puuvili
False
```

Hulkadega on võimalik teha tehteid tehteid (summa, ühisosa, ühend, vahe):

```
>>> # hulgaoperatsioonid
... >>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a # a erinevad tähed
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b # tähed, mis on a-s, kuid mitte b-s
set(['r', 'd', 'b'])
>>> a | b # tähed a-s või b-s
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b # tähed a-s ja b-s
set(['a', 'c'])
>>> a ^ b # tähed a-s või b-s, kuid mitte mõlemas
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Kujutused

Erinevalt numbriliste indeksitega loendidest, hulkadest, korteežidest ja tekstist võib kujutuse võtmeks olla suvaline muutumatu andmetüüp. Tekst ja arv võib olla igal juhul kujutuse võtmeks. Korteež võib olla kujutuse võtmeks ainult siis kui ta sisaldab ainult teksti, arve või korteeže. Loendid ei saa olla kujutuse võtmeteks, kuna neid võib käigu pealt muuta.

Kujutust võib käsitleda järjestamata võtme: väärtuse paaride hulgana, eeldades, et võtmed on unikaalsed. Loogeliste sulgude paar `{}` loob tühja kujutuse.

Kujutust on võimalik täita sisuga, eraldades võtme ja väärtuse omavahel kooloninga ning iga sellise paari teisest komaga. Nii kirjutatakse kujutused ka väljundisse:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
```

```
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
>>> 'guido' in tel
True
```

Kui võtmeteks on lihtne (ilma tühikuteta) tekst, on mõnikord hõlpsam määrata paarid võtmesõnaliste argumentidena:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Kujutust lehitsedes (ingl. *looping*) võib võtit ta talle vastavat väärtust koraga kätte saada kasutades `iteritems()` meetodit:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
... print k, v
... gallahad the pure robin the brave
```

Tõeväärtus

Tõeväärtuse andmeid (ingl. *boolean*) võib käsitleda täisarvude alatüübina. Tõeväärtusi `True` (tõene) ja `False` (väär) võib esitada ka kahendsüsteemi arvudena, vastavalt siis `1` ja `0`.

Teistesse andmetüüpidesse kuuluvate muutujate väärtused on samuti käsitletavad tõeväärtusena: nii on näiteks lisaks nullile väär (`False`) ka tühi loend. Lähemalt tõeväärtuste kontrolli käsitlevas osas.

Loendid

Pythonis on mitmeid andmetüüpe, mis võimaldavad väärtusi grupeerida. Loend esitatakse komaga eraldatud väärtustena, mis on ümbritsetud nurksulgudega. Kõik loendi liikmed ei pea olema samast tüübist:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Nii nagu teksti indeksid, algavad ka loendi indeksid nullist. Negatiivse indeksi korral alustatakse lugemist tagant ettepoole, viimane element on indeksiga -1. Loendeid on võimalik nii lõigata kui jätkata:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
```

```
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Erinevalt tekstist, mis on muutumatu objekt, on loendi elemente võimalik muuta kohapeal:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Loendi lõikudele on võimalik omistada väärtusi, nii võib isegi loendi suurust muuta:

```
>>> # Asendame mõned elemendid:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Kustutame:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Lisame juurde:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a # Lisame iseenda koopia algusesse:
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

Sisseehitatud funktsioon `len()` on kohaldatav ka loendidele:

```
>>> len(a)
8
```

Loendid võivad üksteises sisalduda:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Korteežid

Korteež koosneb komaga eraldlatud väärtustest, nagu näiteks:

```
>>> t = 12345, 54321, 'Tere!'
>>> t[0]
12345
>>> t
(12345, 54321, 'Tere!')
```

Korteežid võivad üksteises sisalduda:

```
>>>u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'Tere!'), (1, 2, 3, 4, 5))
```

Väljundis on korteež alati sulgudes. Luues ei pea korteežile alati sulge ümber panema, kuigi avaldiste puhul on see möödapääasmatu.

Korteeže võib kasutada mitmeti, näiteks koordinaatide paaride hoidmiseks (x, y) või andmebaasi kirjete hoidmiseks. Korteežid on nagu sõnedki muutumatud: ei ole võimalik omistada korteeži liikmele omaette väärtust. Siiski on võimalik luua korteeže, mille liikmed on muudetavad (näiteks korteež võib sisaldada loendit).

Tühja korteeži on võimalik konstrueerida lihtsalt sulgude abil:

```
>>> tyhi = ()
```

Ühe liikmega korteeži on võimalik konstrueerida väärtusele koma lisades:

```
>>>yksik = 'Tere', # lõpus on koma
>>>len(tyhi)
0
>>>len(yksik)
1
>>>yksik
('Tere',)
```

Funktsioonid

Programmeerimise juures tuleb sageli ette, et mingit koodi on vaja korrata. Kuigi redaktoris programmi teksti kirjutades pole eriti keeruline "kopipeisti" teha, pole see siiski kuigi mõistlik. Seda koodi peab saama keegi ka lugeda, kui mitte muu, siis vähemalt koodikirjutaja ise. Ja kui homme võib tänane idee ja lähenemine meeles olla, siis järgmisel nädalal võib see olla juba ununenud, rääkimata järgmisest kuust.

Üks võimalus koodi korrata on defineerida funktsioon.

Funktsiooni defineerimine

Operaatori `def` abil defineeritakse funktsioon. See peab sisaldama funktsiooni nime ja parameetreid. Funktsioon võib sisaldada dokumentantsiooniteksti. Kuigi dokumentatsiooniteksti ei ole kohustuslik, on soovitav seda siiski kasutada, sest

see hõlbustab ka automaatset dokumenteerimist. Näiteks on võimalik interaktiivse sessiooni puhul moodulis leiduvate funktsioonide kohta abi küsida võtmesõnaga `help`.

```
>>> def korruta(x,y):
... "Korrutab argumendid x ja y"
... return x*y
...
>>>print korruta(2,3)
6
>>> help(korruta)
Help on function korruta in module __main__:

korruta(x, y)
 Korrutab argumendid x ja y
```

Võtmesõna return tagastab (väljastab) funktsiooni väärtuse. Seda väärtust on võimalik kasutada mujal, näiteks teise funktsiooni või operaatori argumendina:

```
>>>print korruta(4,korruta(2,3))
24
>>>
```

Kuna mitme funktsiooni kirjutamine üksteise "sissse" võib koodi segaseks ajada, siis on kasulik omistada funktsioonist saadud väärtus mingile vahemuutujale:

```
>>>print korruta(4,korruta(2,3))
24
>>>
```

Funktsiooni on võimalik kirjutada ka nii, et ta ei väljasta midagi ja trükib tulemuse välja. Siis näeb küll tulemust kohe, kuid seda ei saa kasutada järmises tehtes:

```
>>>def korruta_prindi(x,y):
... print x*y
...
...
>>>korruta_prindi(2,4)
8
>>>korruta(2,korruta_prindi(2,4))
8
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
   File "<stdin>", line 2, in korruta
     TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

Seades printimisega korrutamise väliseks funktsiooniks, saame mõistliku tulemuse (aga ainult väljatrükina):

```
>>>korruta_prindi(2,korruta(2,3))
12
```

Muidugi ei ole mõistlik defineerida funktsioonina tehet, mis on niigi lihtne ja mis funktsioonina välja kirjutatuna võtab rohkem aega ja vaeva kui lihtsa tehtena. Seepärast teeme midagi praktilisemat: kirjutame funktsioonid, mis teisendaks tollid sentimeetriteks ja vastupidi.

Tollide ja sentimeetrite kohta on meil teada, et üks toll on 2,54 cm. Selleks, et saada mõõt tollides mõõduks sentimeetrites, tuleb korrutada etteantud mõõt (see on siis meie funktsioonis muutuja) 2,54-ga. Defineerime funktsiooni:

```
>>>def toll_cmiks(x):
... "Teisendab mõõdu tollides mõõduks sentimeetrites."
... return x*2.54
...
>>>toll_cmiks(19)
48.26
>>>print toll_cmiks.__doc__
Teisendab mõõdu tollides mõõduks sentimeetrites.
```

Pythonis on võimalus lisada funktsioonile dokumentatsioon. Lisaks selgemini loetavale ja kergemini jälgitavale koodile võimaldab see ka interaktiivses sessioonis kasutatava funktsiooni kohta abi küsida.

Kui väikesel ja kergesti mõistetaval funktsioonil ei ole dokumendi osa kriitiline, siis suurema funktsiooni puhul oleks soovitav kirjeldada, mida ta teeb, milliseid argumente võtab ja mida väljastab.

Pythonis lisatakse dokumentatsioonitekst kohe pärast `def` käsku.

Funktsiooni väärtus ja kõrvaltoime

Funktsioon võib tagastada väärtuse, ent tal võib olla ka kõrvaltoime. Funktsiooni väärtust saab kasutada teiste funktsioonide argumendina. Kõrvaltoimet niimoodi kasutada ei saa. Tüüpiliseks kõrvaltoime näiteks on käsk `print`.

Funktsiooni väärtuse tagastab käsk `return`.

Olgu meil defineeritud funktsioonid:

```
>>>def liida_viis(a):
    b = a + 5
    return b

>>>def liida_p(a):
    b = a + 5
    print b
```

Nad näivad esmapilgul toimivat ühesugusel viisil:

```
>>>liida_viis(0)
5
>>>liida_p(0)
5
>>>liida_viis(3)
8
>>>liida_p(3)
8
```

Kui me rakendame funktsiooni korduvalt, siis esimesel juhul me saame mõistliku tulemuse, teisel juhul aga mitte:

```
>>>liida_viis(liida_viis(0))
10
>>>liida_p(liida_p(0))
5

Traceback (most recent call last):
File "<pyshell#11>", line 1, in <module>
liida_p(liida_p(0))
File "<pyshell#10>", line 2, in liida_p
b = a + 5
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

See on sellepärast nii, et esimene funktsioon (`liida_viis`) tagastab väärtuse, kuid teine funktsioon (`liida_p`) ei tagasta mitte midagi, ainult prindib välja. Seda on võimalik kontrollida käsuga `type`:

```
>>> x = liida_viis(3)
>>> type(x)
<type 'int'>
>>> z = liida_p(3)
8
>>> type(z)
<type 'NoneType'>
```

Kui me kasutame funktsiooni `liida_p` argumendis funktsiooni `liida_viis`, siis tuleb trikk välja. Vastupidi aga mitte. Proovi!

Seda võib võrrelda olukorraga, kus meil on keldrikorter, mille aknast me näeme ainult möödakäijate jalgu, heal juhul ka näpuotsi. Kui möödakäijad on nagu funktsioonid ja see, mis neil näpus, väärtused, siis on olukord järgmine:

- Kui keegi pillab midagi maha, siis me seda näeme: see on nagu funktsioon, mis prindib midagi välja
- Kui keegi hoiab midagi näpu vahel, siis seda me ka näeme: see on nagu funktsioon, mis tagastab väärtuse (kuid ei anna seda teisele edasi).
- Kui keegi ulatab midagi teisele, siis me seda ei näe (nagu funktsioon, mis tagastab väärtuse, kuid selle saab teine funktsioon) enne kui teine selle maha pillab (välja prindib) või niisama näpu otsas lehvitab (tagastab).
- Kui meil on selline tüüp, kes pillab midagi maha (`liida_p`), siis teine tüüp ei saa seda kätte ja jääb tühjade kätega (ja ilmselt hakkab nurisema)

Rekursiivsed funktsioonid

Hargnemine

Tingimus `if`

Mis oleks programmeerimiskeel ilma tingimusteta? Pythonis on põhiliseks tingimus-operaatoriks `if`

```
>>> x = int(raw_input("Kui palju on 2+2? "))
>>> if x == 4:
...     print "Õige!"
...elif x < 4:
...     print "Liiga vähe pakkusid."
...elif x > 4:
...     print "Liiga palju pakkusid."
...
```

Erinevate valikute vahel lülitab võtmesõna `elif`. Selle võib vajadusel ka ära jätta, samas võib teda olla ka kuitahes palju.

Tõeväärtuse kontroll

Selleks, et kontrollida tõeväärtust, ei pea olema avaldist, piisab ka objektist. Järgmised objekti väärtused annavad vastuseks "väär" (`False`):

- None
- False
- null, ükskõik millise arvutüübi puhul
- tühi järgnevus, st. "",[],()
- tühi kujutis, st. {}

Ülejaanud juhtumitel (enamasti) on objekti tõeväärtuseks "tõene" (`True`).

Tsüklid

Tsükkel `for`

Operaator for täidab temas sisalduva(d) korralduse(d) iga argumendis leiduva elemendi kohta. Argumendiks võib olla suvaline järgnevus (loend, korteež, sõne):

```
>>> for i in ['apelsin', 'kollakasroheline','suur reheahi']:
... print i, len(i)
... apelsin 7 kollakasroheline 16 suur reheahi 12
```

Tsükkel `while`

Operaator while täidab sisaldist kuni tingimus vastab tõele:

```
>>> a = ['apelsin', 'kollakasroheline','suur reheahi']
>>> while a:
...     print a.pop()
...
suur reheahi
kollakasroheline
apelsin
```

Eesti keeles võiks seda kahrerealist koodijuppi ümber kirjutada järgmiselt: "Nopi loendi liikmeid (tagantpoolt) seni kuni loendi jätkub."

Kõigepealt omistatakse muutujale a loendi väärtus. Loendil on kolm liiget: 'apelsin', 'kollakasroheline' ja 'suur reheahi'. Operaator while töötab seni, kuni tingimus on täidetud. Pikemalt võiks seda kirjutada ka nii:

```
>>> while len(a) > 0:
```

See oleks siis: "Kuni loendi `a` pikkus on suurem kui null."

Kuna null pikkusega loend vastab `False` tõeväärtusele (lähemalt tõeväärtustest), siis piisab ka sellest, kui tingimuseks seada loendi `a` olemasolu.

Tsükli töö katkestamine ja jätkamine

Operaator `break` katkestab lähima `for` või `while` tsükli töö.

Operaator `continue` jätkab järgmise tsükli käiguga.

Operaator `else` võib sisalduda ka tsüklis. Sel juhul täidetakse `else` sisaldis juhtumil, kui `for` loend on läbi saanud või `while` tingimus ei ole täidetud, kuid mitte siis, kui on jõutud `break` operaatorini.

Failide lugemine ja kirjutamine

Failiobjekti tagastab funktsioon `open()` (kasutatav ka kujul `file()`). Nendel funktsioonidel on võimalikud kolm argumenti: esimene on tekst, milleks on avatava faili nimi, teine tekst, mis näitab, mis moel faili avatakse, kolmas on täisarv, mis näitab, kui palju faili loetakse. Teine ja kolmas on fakultatiivsed argumendid.

```
>>> f=open('katsetus.txt', 'w')
>>> print f
<open file 'katsetus.txt', mode 'w' at 80a0960>
```

Esimene argument on sõne, mis sisaldab faili nime. Teine argument on samuti sõne, mis sisaldab tähti, millega on kodeeritud viis, kuidas failiga edasi tegutsetakse. Mood on 'r' siis kui fail avatakse ainult lugemiseks; 'w' kui fail avatakse kirjutamiseks (olemasolev fail, millel on sama nimi, kirjutatakse üle). Mood 'a' avab faili lisamiseks: andmed kirjutatakse faili lõppu. Mood 'r+' avab faili nii lugemiseks kui kirjutamiseks. Moodi argument on fakultatiivne, kui seda ei määrata, siis avatakse fail vaikimisi lugemiseks, nagu 'r' puhul.

Windowsi ja Macintoshi masinate puhul avab 'b' faili binaarmoodis, nii on võimalikud ka moodid 'rb', 'wb', 'r+b'. Windowsis tehakse vahet teksti ja binaarfailide vahel. Realõpu märke muudetakse automaatselt, kui andmeid loetakse või kirjutatakse. Tavalise tekstifaili puhul pole sellest midagi, aga EXE või JPEG failide puhul tuleb hoolikalt jälgida 'b' kasutamist, muidu võivad sellistes failides andmed kaotsi minna.

Failiobjektide meetodid

Järgnevate näidete puhul eeldatakse, et failiobjekt `f` on juba loodud.

Selleks, et lugeda faili `f` sisu, kasutatakse failiobjekti meetodit `read()`. Võib sisaldada fakultatiivset täisarvulist argumenti, mis näitab, kui palju failist loetakse. Meetod loeb suuruse jagu faili ja tagastab selle sringina. Kui suurust pole määratud või on see negatiivne, siis loetakse ja tagastatakse kogu fail. Kui suurus on ette antud, siis loetakse niipalju faili baitides. Kui jõutakse faili lõppu, siis tagastab `f.read()` tühja sõne (“”):

```
>>> f.read()
'See on kogu fail.\n'
>>> f.read()
''
```

Faili `f` meetod `readline()` loeb failist `f` ühe rea. Reavahetuse märk (`\n`) jääb rea lõppu. Ära jäetakse ta ainult siis, kui tegemist on faili viimase reaga ja selle lõpus ei ole reavahetusmärki:

```
>>> f.readline()
'See on faili esimene rida.\n'
>>> f.readline()
'Faili teine rida\n'
>>> f.readline()
''
```

Faili `f` meetod `readlines()` tagastab loendi, mis sisaldab kõiki ridu failis. Kui on antud ka suvandparameeter sizehint, siis loetakse nii palju faili (baitides) ja veel niipalju, et rida saaks lõpetatud, siis tagastatakse need read. Seda kasutatakse selleks, et sirvida suuri faile ridade kaupa, ilma et peaks kogu faili korraga mällu lugema. Tagastatakse ainult tervikread:

```
>>> f.readlines()
['See on faili esimene rida.\n', 'Faili teine rida\n']
```

Teine võimalus on tekitada tsükkel üle objekti. See ei koorma mälu, on kiire ja lihtne:

```
>>> for line in f:
... print line
...
See on faili esimene rida.

Faili teine rida
```

Teisena toodud näide on lihtsam, kuid ei võimalda nii üksikasjalikku kontrolli. Kuna erinevad lähenemised kasutavad erinevaid viise rea puhverdamiseks, ei tohiks neid ühe ja sama faili lugemisel segamini kasutada.

Faili `f` meetod `write(<string>)` kirjutab sõne faili, tagastab `None`

```
>>> f.write('See on katse.\n')
```

Selleks, et kirjutada faili ka teisi andmetüüpe, on vaja need kõigepealt konverteerida sõneks:

```
>>> value = ('vastus on', 42)
>>> s = str(value)
>>> f.write(s)
```

Faili `f` meetod `tell()` tagastab täisarvu, mis tähistab failiobjekti jooksvat positsiooni failis, mõõdetuna baitides faili algusest. Selleks, et muuta failiobjekti positsiooni, kasutatakse meetodit `seek(<offset>, <millest>)`. Positsioon arvutatakse lisades offseti referentspunktile. Referentspunkti valib 'millest' argument. Kui 'millest' on 0, siis loetakse faili algusest; 1 kasutab jooksvat positsiooni ja 2 kasutab faili lõppu referentspunktina. Vaikimisi on 'millest' 0, seega loetakse faili algusest:

```
>>> f = open('katsetus.txt', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5) # mine 6. baidi juurde failis
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # mine 3. baidi juurde lõpust arvates
>>> f.read(1)
'd'
```

Peale faili kasutamist on soovitav kasutada meetodit `close()`, et faili sulgeda ja vabastada süsteemi ressursid, mis on seotud avatud failiga. Peale faili sulgemist ei saa seda avada enne, kui kasutada uuesti meetodit `open()`:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
 ValueError: I/O operation on closed file
```

Regulaaravaldised

Erivõtted

MULTILINE režiim

`re.M` või `re.MULTILINE`

> Moodulis `re` on võimalik määrata konstante, mille abil on võimalik juhtida regulaaravaldiste käitumist. `MULTILINE` konstant, kui tema väärtus on on tõene, muudab sõne alguse ja lõpu otsimist, andes vastuseks ka vastavalt pärast ja enne reavahetust leiduvad märgid.

DOTALL lipp

`re.S` või `re.DOTALL`

> Moodulis `re` on võimalik määrata konstante, mille abil on võimalik juhtida regulaaravaldiste käitumist. `DOTALL` konstant, kui tema väärtus on tõene, muudab `.` (punkt) erimärgi käitumist, seades talle vastavusse ka reavahetuse.

Ahne ja kasin avaldis

Regulaaravaldis võib olla "ahne" või "kasin". Vaikimisi on regulaaravaldised "ahned", avaldisele vastab võimalikult pikk sõne. Sageli pole see tulemus soovitud. Nii näiteks püüdes leida html tekstist pealkirja märgendust `<h1>` ja kasutades selleks avaldist `<.*>` saame vastuseks `<h1>pealkiri</h1>`, mitte lithsalt `<h1>`.

Kasinaks muudab regulaaravaldise `"?"` lisamine. Kasina avaldise puhul otsitakse vähimat sobivat vastet. Eelmise näite puhul tuleks kasutada avaldist `<.*?>`, mis viib soovitud tulemuseni.

Operatsioonid regulaaravaldistega

Pythoni regulaaravaldised sarnanevad Perli omadele. Nii mustrid kui sõned võivad olla nii 8-bitised kui unikoodi omad.

Regulaaravaldistes kasutatakse kurakaigast (\) tähistamaks erikorraldusi või erimärke. See käib vastu Pythoni üldisele ideele kasutada sõnedes ühesugust märgendust. Nii tuleb näiteks kurakaika tähistamiseks kirjutada regulaaravaldises `\\`, kuna kurakaika regulaaravaldis on `\` ja iga kurakaigas on `\` tavalises Pythoni sõnes.

Õnneks on Pythonis võimalus kasutada toorsõne märgendust. Kui sõne on tähistatud `r` lipuga, siis ei käsitleta kurakaikad erireeglite kohaselt. Nii on `r"\n"` kahetäheline sõne, mis sisaldab kurakaigast ja väikest n tähte, kuid `"\n"` reavahetus. Tavaliselt estitatakse regulaaravalduste mustrid just r-lippudega sõnedena.

Regulaaravaldiste süntaks

Regulaaravaldis võib sisaldada nii tavalisi märke kui erimärke. Kõige lihtsamad regulaaravalised koosnevad ainult tavalistest märkidest, nagu `"A"`, `"a"`, `"B"`, `"0"`. Need vastavad ainult iseendale.

Erimärgid

Mõned märgid on erilised, nagu näiteks "|", "(" või "*". Need tähistavad kas mitut tavalist märki või muudavad regulaaravaldise tähendust.

Pythoni regulaaravaldistes kasutatavad erimärgid on järgmised:

`.` (punkt)

> Vaikimisi tähistab iga märki peale reavahetuse. Kui DOTALL lipp[2] on püsti, siis tähistab iga märki, kaasa arvatud reavahetust.

`^` (katus)

Tähistab sõne algust. MULTILINE režiimis[1] tähistab ka iga märki, mis järgneb vahetult reavahetusele.

`$` (dollarimärk)

Tähistab sõne lõppu või märki, mis eelneb vahetult reavahetusele. MULTILINE režiimis[1] tähistab ka reavahetusele eelnevat märki. Avaldisele `böö` vastavad nii `böö` kui `böö minghäm`, aga avaldisele `böö$` vastab ainult `böö`.

`*` (tärn)

Kordab vahetult eelnevat avaldist null või enam korda, nii palju kui võimalik. Avaldisele `ab*` vastavad nii `a`, `ab`, `abbb` kui ka `abbbbbbbbbbbbb`.

`+` (pluss)

Kordab vahetult eelnevat avaldist vähemalt ühe korra, nii palju kui võimalik. Avaldisele `ab+` vastavad nii `ab`, `abbb` kui ka `abbbbbbbbbbbbb`, kuid mitte `a`.

`?` (küsimärk)

Kordab vahetult eelnevat avaldist null või üks kord. Avaldisele `ab?` vastavad `a` ja `ab`.

`*?, +?, ??`

Avaldised `*`, `+`, ja `?` on kõik "ahned"[3]: neile vastab nii suur hulk teksti kui võimalik. Alati ei ole see sobiv. Olgu meil vaja näiteks leida html-märgendust. Kasutades avaldist `<.*>` saame vastuseks `<h1>pealkiri</h1>`, mitte lithsalt `<h1>`. `?` lisamine muudab avaldise minimaaset vastet otsivaks. Eelmise näite puhul tuleks kasutada avaldist `<.*?>`, mis viib soovitud tulemuseni.

`{m}`

Eelnev avaldis esineb täpselt `m` korda. Nii näiteks avaldis `a{6}` leiab kuus `a`-d, aga mitte viit `a`-d.

`{m,n}`

Eelnev avaldis esineb `m` kuni `n` korda, niipalju kui võimalik. Näiteks avaldisele `a{3,5}` vastab kolm kuni viis `a`-d. Kui `m` ära jätta, siis muutub alumine raja nulliks, kui `n` ära jätta, siis muutub ülemine raja lõpmata suureks. Näiteks avaldis `a{4,}b` leiab `aaaab` ja `aaaaaaaaab` või kuitahes paljude `a`-dega analoogilist teksti, kuid mitte `aaab`.

Ettevaatust!

Koma ei tohi ära jätta, muidu on tegemist eelmise avaldisega!

[1] vt MULTILINE režiim

[2] vt DOTALL lipp

`{m,n}?`

Eelnev avaldis esineb `m` kuni `n` korda, nii vähe kui võimalik. See on eelmise avaldise kasin variant[3]. Nii näiteks vastab kuuemärgilise teksti `'aaaaaa'` puhul variandis `a{3,5}` viis `a`-d, aga `a{3,5}?` puhul ainult kolm.

`\`

Maskeerib erimärgid (võimaldades otsida/asendada regulaaravaldise kontrollmärke, nagu `*`, `?`, `+` ja muud) või tähistab eritähendusega märgi algust.

`[]`

Tähistab märkide valikut. Need võivad olla ükshaaval üles loetud, või märgitud kui vahemik. Viimasel juhul on märgid eraldatud `--`ga. Erimärgid on klambrites tavaliste märkide seisuses. Näiteks `[aui]` vastab `a`-le, `u`-le või `i`-le, `[0-9]` vastab ükskõik millisele numbrile, `[a-z]` vastab ükskõik millisele väiketähele[4]. Ka täheklassid on klambrites kasutatavad. Kui on vaja klambritesse panna `]` või `-` märki, siis tuleb need panna kas esimeseks klambrite sees olevaks märgiks või kasutada nende ees kurakaigast. Näiteks `[]]` või `[.,\-]`.

On võimalik tähistada ka valikut, mis leiab kõik märgid, välja arvatud need, mis on klambrite vahel. Selleks pannakse klambrite vahele esimesena katus (`^`). Katus, mis ei ole esimses positsioonis, tähistab lihtsalt katust. Kehtivad ka vahemikud. Näiteks `[^1-5]` leiab kõik märgid, mis ei ole 1, 2, 3, 4, ega 5.

`|`

`A|B`, kus `A` ja `B` võivad olla suvalised regulaaravaldised. Vastab kas `A`-le või `B`-le. Selliseid avaldisi võib olla mitu üksteise järel. Võib kasutada ka grupi sees. Vastavust kontrollitakse vasakult paremale. See tähendab seda, et kui üks avaldis sobib, siis teist ei hakata enam kontrollima, isegi siis kui see vastaks pikemale sõnejadale. Selline omadus teeb `|`-st kasina operaatori.

Selleks, et leida tekstis esinavat `|` märki, kasutatagu kurakaikaga maskeerimist (nagu `\|`) või märgi klambritesse sulgemist (nagu `[|]`).

Objektorienteeritud programmeerimine

Mõnikord on mõttekas ka funktsiooni sees defineerida funktsiooni. Sellist, mille kohta on teada, et neid pole mujal vaja kui ainult selle funktsiooni allülesannete täitmiseks. Siis on meil raamfunktsioon, millel on mingid argumendid, ja tegeliku töö teevad ära raamfunktsiooni alamfunktsioonid. Võib öelda, et sellisest programmeerimisstiilist on alguse saanud objekt-orienteeritud programmeerimine.

[3] vt Ahne ja kasin avaldis

[4] Eeldusel, et kasutusel on `C` lokaale.

Objektorienteeritud programmeerimise puhul võib objekti käsitleda kui andme- ja funktsioonikogumit. Kõigepealt defineeritakse mudel, milles on kohad andmete jaoks ja funktsioonid, mida nendega tehakse. OO lähenemise korral nimetatakse selliseid andme- ja funktsioonikogumeid klassideks, andmeid atribuutideks ja funktsioone meetoditeks. Kasutusel on ka mõiste omadus (ingl. *property*) , mis on sisuliselt sama, mis atribuut, millel kasutatakse spetsiaalseid `get()` ja `set()` meetodeid.

Võtame konkreetse näite. Olgu meil vaja programmi, mis aitaks arvestust pidada mitmete jalgrataste mitmesuguste omaduste üle.

Sissejuhatus

Klassi kirjeldus

Klassiks nimetatakse seda "raamfuktsiooni", milles sisalduvad kohad andmete ja alamfunktsioonide jaoks. Klassi võib käsitleda ka kui prototüüpi või mudelit, millele vastavad konkreetsed objetkid konkreetsete väärtustega.

Pythonis on klassi defineerimiseks võtmesõna `class`. Konstruktorina kasutatakse funktsiooni `__init__()`. Selle funktsiooni abil defineeritakse vastava klassi objektidele *atribuudid*.

Olgu meil näiteks tarvis kirjeldada erinevaid jalgrattaid. Olgu meil antud ülesanne, kus on vaja arvutada, mitu meetrit jalgratas edasi liigub, kui meil on teada, mitu vändapööret sõitja teeb. Lähteandmetena on iga jalgratta kohta ka teada, mitu hammast on väntadega hammasrattal ja mitu hammast on tagumisel hammasrattal, mis ajab ratast ringi. (Lihtsuse mõttes eeldame, et tegemist on ilma käikudeta jalgratastega.)

Et lahendada seda ülesannet objektorienteeritud stiilis, defineerime kõigepealt jalgratta klassi. Et süntaks oleks korrektne, ütleme, et ta ei pea alguses midagi tegema. (Selleks kasutame võtmesõna pass.)

```
>>> class jalgratas():
...     pass
```

Kontrollime, kas töötab:

```
>>> a = Jalgratas()
>>> print a
<__main__.Jalgratas instance at 0xb7d772cc>
```

Klass Jalgratas töötab, me võime luua selle klassi eksemplare.

Meil ei ole palju praktilist kasu jalgrattast, millega midagi teha ei saa, mida isegi kirjeldada ei saa. Palju kasu pole sellestki, kui defineerime oma jalgratta klassi interaktiivses sessioonis. Sestap avame uue faili ja kirjutame oma jalgratta definitsiooni sinna.

Alustame uuesti otsast peale. Kõigepealt loome klassi jalgratas, koos klassiga ka kirjelduse - dokumentatsiooni osa.

Nüüd näeb meie fail välja selline:

```
class Jalgratas:
    '''
    Jalgratta klass.
    '''
```

Klassi eksemplari loomisel käivitatakse kõigepealt funktsioon `__init__()`. Kuna meie klass on loodud "puhta lehena", siis peame defineerima kõigepealt selle funktsiooni. Kirjeldame seal atribuute, mida tahame kasutada. Atribuutideks on omanik, see hakkab hoidma omaniku nime; esimene, sinna tuleb esimese hammasratta hammaste arv; tagumine (tagumise hammasratta hammaste arv), rattad, (rataste läbimõõt tollides, ehk kummi mõõt).

Kui me tahame, et kõiki neid atribuute oleks võimalik sättida ka klassi eksemplari loomisel, siis peame nad kirjutama ka `__init__()` argumentideks.

Nüüd näeb meie fail välja selline:

```
class Jalgratas:
    '''
    Jalgratta klass.
     '''
     def __init__(self,omanik="jalgratta omanik",
             esimene=0, tagumine=0,
                     rattad=0):
        self.omanik = omanik
        self.esimene = esimene
        self.tagumine = tagumine
        self.rattad = rattad
        self.kumm = 1
```

Vahepeal võib tekkida isu proovida, kas loodud klass ka funktsioneerib. Salvestame faili nimega `kulgurid.py` - võibolla on meil vaja defineerida veel mõnda liiklusvahendit.

Nüüd on võimalik seda faili juba proovida. Kuigi meil pole tegemist käivitatava failiga, saame seda välja kutsuda kui moodulit. Selleks käivitame interpretaatori ja kasutame selles käsku `import`:

```
>>> import kulgurid
>>> a = kulgurid.Jalgratas()
>>> print a.omanik
jalgratta omanik
>>> print a.esimene
0
```

Näeme, et klassi eksemplari loomine õnnestub ja atribuutidel on esialgsed väärtused olemas.

Nüüd võime ka imporditud mooduli kohta abi küsida:

```
>>> help(kulgurid.Jalgratas)
Help on module kulgurid01:

NAME
    kulgurid01

    FILE
        /home/nemee/Pythoniasjad/kulgurid01.py

    CLASSES
        Jalgratas
```

```
        class Jalgratas
        |  Jalgratta klass.
        |
        |  Methods defined here:
        |
        |  __init__(self, omanik='jalgratta omanik',
esimene=0, tagumine=0, rattad=0)
```

See tuletab meile meelde, et on paras aeg täiendada jalgratta klassi dokumentatsiooni. Ilus on kirjeldada ära klassi atribuudid, näiteks nii:

```
Atribuudid:

omanik - omaniku nimi (sõne)
esimene - esimese (väntadega) hammasratta hammaste arv (int)
tagumine - tagumise hammasratta hammaste arv (int)
rattad - kummi siseläbimõõt tollides (int)
kumm - kummi paksus (tollides), vaikimisi 1
```

Järgmisena loome objektile rakendatava *funktsiooni* ehk *meetodi*. Meil oli tarvis välja arvutada teekond, mille jalgratas teatava hulga vändapööretega läbib. Võtame siis vaikimisi vändapööreteks ühe, hiljem saame vajaliku numbri anda loodavale funktsioonile argumendiks ja sellega siis lõpptulemuse läbi korrutada.

defineerime funktsiooni `teekond()`:

```
def teekond(self,p=1):
```

Funktsiooni `teekond()` argumentideks on `self` - see tähistab objekti ennast - ja `p`, mille puhul me leppisime kokku, et see tähistab vändapöörete arvu ja seda kasutame pärast. Argument `self` võimaldab meil kasutada funktsiooni `__init__()` abil defineeritud atribuute.

Meil on vaja teada esimese hammasrata hammaste arvu - selle annab meile atribuut `self.esimene` ja tagumise hammasratta hammaste arvu - selle annab meile atribuut `self.tagumine`. Nende abil arvutame ülekande, tähistame selle muutujaga `ylekanne`. Ülekanne on lihtsalt esimese ja tagumise hammasratta suuruste suhe (meil siis hammastes), saame, kui jagame atribuudid esimene ja tagumine. Funktsioon näeb meil nüüd välja järgmine:

```
def teekond(self,p=1):
    ylekanne = self.esimene / self.tagumine
```

Tagumine ratas liigub koos tagumise hammasrattaga, edasi on vaja arvutada, kui pika maa ratas läbib ühe pöördega. Ühe pöördega läbib ratas teekonna, mis on võrdne ratta ümbermõõduga: läbimõõt korda pii. Läbimõõdu saame nii, kui võtame pöia mõõdu ja lisame sellele kahekordse kummi paksuse (mis on sama palju kui laius, enamvähem). Pöia mõõt on meil atribuudis rattad , kummi laius(=kõrgus) atribuudis `kumm`. Läbitud vahemaa kokku on `ylekanne * (rattad + 2 * kumm) * pii`. Me ei pea võtma uut muutuja nime, omistame lihtsalt muutujale `ylekanne` uue väärtuse. Nüüd on meie funktsioon kasvanud jälle rea võrra:

```
    def teekond(self,p=1):
        ylekanne = self.esimene / self.tagumine
        ylekanne = ylekanne * (self.rattad + 2*self.kumm) * 3.14
```

Kuna rataste läbimõõt ja kummi paksus on antud tollides, siis arvutame tulemuse meetriteks ümber, arvestusega `1" = 2,54cm ja 1cm = 0,01m`. Seega `teekond = ylekanne * 2.54 * 0.01`. Muutuja `teekond` tagastatakse käsuga `return`:

```
    def teekond(self,p=1):
        ylekanne = self.esimene / self.tagumine
        ylekanne = ylekanne * (self.rattad + 2*self.kumm) * 3.14
        teekond = ylekanne * 2.54 * 0.01
        return teekond
```

Kirjutame funktsiooni kohta jutu dokumentatsiooniridadele ja saame lõpliku klassi:

```
    class Jalgratas:
        '''
        Jalgratta klass.

        Atribuudid:

        omanik - omaniku nimi (sõne)
        esimene - esimese (väntadega) hammasratta hammaste arv (int)
        tagumine - tagumise hammasratta hammaste arv (int)
        rattad - kummi siseläbimõõt tollides (int)
        kumm - kummi paksus (tollides), vaikimisi 1

        Meetodid:

        teekond(p=1)
        tagastab vändapööretega p läbitud teekonna (meetrites)
        '''
        def __init__(self,omanik="",
                     esimene=0, tagumine=0,
                     rattad=0):
            self.omanik = omanik
            self.esimene = esimene
            self.tagumine = tagumine
            self.rattad = rattad
            self.kumm = 1

        def teekond(self,p=1):
            '''
            Arvutab vändapööretega läbitud teekonna.

            Sisendiks on klassieksemplar (self) ja
            vändtud pöördeid (p), mis on vaikimisi 1.
```

```
        Sel ajal kui vändahammasratas teeb 1 pöörde,
        teeb tagumine hammasratas esimene/tagumine pööret.

        See on ylekanne.

        Tagumine ratas liigub koos tagumise hammasrattaga.

        Ühe pöördega läbib ratas teekonna, mis on võrdne
        ratta ümbermõõduga: läbimõõt korda pii.

        Läbitud vahemaa kokku on ylekanne * (rattad + 2 * kumm) * pii.

        Kuna rataste läbimõõt ja kummi paksus on antud tollides,
        siis arvutame tulemuse meetriteks ümber, arvestusega
        1" = 2,54cm ja 1cm = 0,01m. Seega

        teekond = ylekanne * 2.54 * 0.01

        Väljastatakse teekond.

        '''
        ylekanne = self.esimene / self.tagumine
        ylekanne = ylekanne * (self.rattad + 2*self.kumm) * 3.14
        teekond = ylekanne * 2.54 * 0.01
        return teekond
```

Nüüd on võimalik küsida ka täielikumat abi, kasutades päringut `help(kulgurid.Jalgratas)`.

Klassi eksemplarid

Klassi on võimalik kasutada nüüd millekski kasulikuks. Kõigepealt loome klassi eksemplarid - kirjeldame ära konkreetsed jalgrattad, mille kohta meil on vaja ülesandeid lahendada.

Teeme seda jälle interaktiivses kasutusmoodis:

```
>>> import kulgurid
>>> a = kulgurid.Jalgratas()
>>> b = kulgurid.Jalgratas()
>>> c = kulgurid.Jalgratas()
>>> a.omanik = 'Mikk'
>>> a.esimene = 46
>>> a.tagumine = 14
>>> a.rattad = 27
>>> b.omanik = 'Mann'
>>> b.esimene = 42
>>> b.tagumine = 16
>>> b.rattad = 28
>>> c.omanik = 'Juku'
>>> c.esimene = 38
>>> c.tagumine = 22
>>> c.rattad = 26
```

Defineerime nüüd funktsiooni, mis näitab, kui palju keegi edasi vändates liigub:

Funktsioon `liigub()` ei tagasta midagi, ainult prindib välja, kes kui palju väntas ja kui palju edasi liikus.

Olgu meil funktsiooni parameetriteks:

1. jalgrattaobjekt, millega tehteid teeme
2. mitu vändapööret teeme.

Olgu meil esimene muutuja `kes` ja teine `ajab_ringi`. Muutuja kes väärtuseks tohib olla ainult klassi Jalgratas eksemplar.

Funktsiooni `liigub()` sisu saame väljendada ühe `print` operaatoriga. Kirjutame sinna välja trükitava lause koos lünkadega, mille täidame erinevate muutujate väärtustega, millest osad arvutame kohapeal:

```
print "%s väntas %d korda ja liikus edasi %.2f meetrit."
```

Esimest lünka tähistav `%s` täidetakse jalgratta omaniku nimega. Jalgratta omaniku ja kõik muud defineeritud tunnused saame jalgratta objektiga kaasa. Esimese lünga täidab `kes.omanik`.

Teine lünk täidekse täisarvuga, milleks on mitu korda vändati. Täisarvu kohta hoiab `%d`. Muutuja on `ajab_ringi`.

Kolmas lünk täidetakse reaalarvuga, mis tähistab läbitud maad. See saadakse jalgratta objekti `kes` meetodi `teekond()` abil, mille argumendiks on väntamiskordade arv `ajab_ringi`:

```
kes.teekond(ajab_ringi)
```

Meil ei ole vaja kasutada kogu täpsust, mida teekonna meetod pakub. Sellepärast rakendame sellele ümardamisfunktsiooni round, andes täpsuse sajandikega (kaks kohta peale koma):

```
round(kes.teekond(ajab_ringi),2)
```

Kui me jätaksime lünka tähistama ainult `%f`, siis saaksime vastuseks teksti, kus on küll välja arvutatud täpsusega kaks kohta peale koma, kuid sellele järgneks veel hulk nulle. Selleks, et lüngas oleks ka märgitud vajadus väljastada kaks kohta pärast koma, tähistame selle `%.2f`.

Nüüd on meil funktsioon töövalmis, kuid ta kirjutab igal juhul, et `kes.omanik` väntas `ajab_ringi` korda, ka siis kui `ajab_ringi = 1`. Selleks märgime veel ühe lünga, osutades, et see tuleb täita muutujaga `k`. Muutuja `k` seame sõltuvusse muutuja `ajab_ringi` väärtusest:

```
if ajab_ringi == 1:
    k = "kord"
else:
    k = "korda"
```

Koos dokumendikirjega saame funktsiooni `liigub()` definitsiooniks:

```
def liigub(kes, ajab_ringi=1):
    '''
    Arvutab, kes kui palju vändates edasi liigub.
```

```
kes olgu klassi kulgurid.Jalgratas() eksemplar,

ajab_ringi int. Vaikimisi väärtus on sellel 1.
'''
if ajab_ringi == 1:
    k = "kord"
else:
    k = "korda"
print "%s väntas %d %s ja liikus edasi %.2f meetrit." % (kes.omanik,
                                                         ajab_ringi,
                                                         k,
                                                         round(
                                                         kes.teekond(
                                                         ajab_ringi),2
                                                         ))
```

Atribuudid ja meetodid

Millal kasutada objektorienteeritud stiili?

Ühest küljest võib öelda, et tegemist on maitse asjaga. Hea programmeerija võib saavutada hea tulemuse ka klasse ja objekte kasutamata, kuid näiteks eelmises osas vaadeldud ülesande puhul oleks vaja kirjutada palju tingimuslauseid, mis teevad keeruliseks nii koodi kirjutamise kui hilisema parandamise ja muutmise. Just selliste ülesannete puhul, kus on vaja paljude programmeerijate koostööd, soovitatakse objektorienteeritud lähenemist.

Moodulid

Kui Pythoni interpretaator sulgeda, siis teda uuesti avades võite tõdeda, et eelmine kord defineeritud funktsioonid ja muutujatele omistatud väärtused on kadunud. Pikema programmi puhul on kasulik tarvitada mõnda tekstiredaktorit, milles programm valmis kirjutada. Suure projekti korral on mõttekas kasutada mitut faili, mida on kergem majandada. Samuti on kasulik kirjutada eraldi faili need omakirjutatud funktsioonid, mida mitmes programmis võib vaja minna.

Moodulite kasutamine

Moodulite kasutamisel tuleb nad kõigepealt laadida.

Selleks on käsk `import`, mida võib kasutada mitmel moel. Olgu meil vaja importida moodulit sys:

```
>>>import sys
```

Sellise importimise korral tuleb moodulis leiduva kasutamiseks lisada ette mooduli nimi, millele järgneb punkt:

```
>>>import sys
>>>print sys.argv[1]
```

Standardmoodulid

Vead ja erindid

Pythonis on (vähemalt) kahte liiki eristatavaid vigu: süntaksivead ja erandid.

Süntaksivead

Pythoni õppija kohtab tõenäoliselt kõige sagedamini süntaksivigu:

```
>>> while True print 'Tere maailm!'
  File "<stdin>", line 1, in ?
    while True print 'Tere, maailm!'
                     ^
SyntaxError: invalid syntax
```

Parser kordab rida, mis talle ei meeldinud ja näitab 'nooleotsaga' kohta, kus avastati viga. Viga on põhjustatud (või vähemalt avastatud) selles osas, mis eelneb nooleotsaga märgitud kohale. Antud näites võtmesõna `print` juuures, kuna puudub sellele eelnev koolon (":"). Näidatakse ka failinime ja rea numbrit, see aitab vigast kohta kiiresti leida.

Ka siis, kui käsk vastab süntaksile, võib selle käivitamisel tekkida viga. Käivitamisel tekkivaid vigu nimetatakse erinditeks. Erindi tekkimine ei pruugi programmi tööd katkestada - programmeerijal on võimalik tekkiv signaal "kinni püüda" ja anda käsk midagi muud teha:

```
>>> 100/0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Kasutajale antakse tagasisidet sellest, millises programmi osas viga juhtus:

```
File "<stdin>", line 1, in ?
```

ja mis liiki see oli:

```
ZeroDivisionError: integer division or modulo by zero
```

Erinditöötlus

Need tekkinud erindid on võimalik "kinni püüda" ja kasutada neid programmi suunamiseks "haruteele":

```
>>> def jaga(jagatav, jagaja):
...     "Jagab jagatava jagajaga"
...     try:
...         return jagatav/jagaja
...     except ZeroDivisionError:
...         print "Nulliga ei saa jagada!"
...         return None
...
>>> jaga(6,2)
```

```
3
>>> jaga(6,0)
Nulliga ei saa jagada!
```

Ülaltoodud programm püüab kinni ainult nulliga jagamise vea, kui üritada jagajaks või jagatavaks sokutada sõnet, siis lõpeatab funktsioon töö veateatega:

```
>>> jaga(6,'kassikangas')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 4, in jaga
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Selle vastu aitab see, kui defineerida ka tüübi vea püüdmise erind:

```
>>> def jaga(jagatav, jagaja):
...     "Jagab jagatava jagajaga"
...     try:
...         return jagatav/jagaja
...     except ZeroDivisionError:
...         print "Nulliga ei saa jagada!"
...         return None
...     except TypeError:
...         print "Jagada saab ainult arve!"
...         return None
```

Erindite defineerimisel tuleb lähtuda tervest mõistusest ja otstarbekusest - eelkõige kinni püüda need vead, mis muidu oleksid raskesti avastatavad, või need, mis on kerged tekkima (näiteks kasutaja sisestab vale tüüpi andmed)

Testimine

Doctest

Keeletöötlusvahendid

`eurown` moodul

xml

Mis on XML?

`xml.etree.ElementTree`

NLTK

Stiil ja soovitused

Programmi kood pole mitte ainult Pythonile interpreteerimiseks, vaid ka inimestele lugemiseks. Kui keegi teine peaks teie loodud programmi hiljem parandama või isegi lugema, et sellest aru saada, on hea pidada kinni mõnendest soovitustest, mis teevad programmi lugemise kergemaks. Äärmuslikud stiilist

kinnipidamise pooldajad väidavad koguni, et kood peaks olema mõeldud eelkõige teistele inimestele lugemiseks, et see midagi kasulikku ka masinas teeb, on puhtalt kõrvalmõju.

Ühtsest stiilist on kasu ka siis, kui loete ise pärast enda kirjutatud programmi.

Need soovitused ei ole väga ranged, erinevad programmeerimisfirmad näiteks võivad anda erinevaid soovitusi, need võivad sõltuda nii varasematest traditsioonidest kui teistest vahenditest, mida kasutatakse. Mingid üldisemad soovitused võib siiski välja tuua.

Üldine

Taanded

Kõige parem on kasutada taandeks alati nelja tühikut.

Dokumenteerimine ja testimine

Kasuta funktsiooni, klassi või mooduli sees doctest mooduli pakutavaid võimalusi.

Avaldised

Üldiselt on soovitav jätta tehtele tühikud sisse:

```
>>> print 5 + 6 +23
```

kuid vahel on loetavuse huvides parem seda mitte teha:

```
>>> print 3 * (5+6)
```

Funktsiooni defineerimine ja kasutamine

Sulud vahetult funktsiooni nime järel, sulgude sees on parem tühikuid mitte kasutada. Funktsiooni väljakutsumisel võiks ka argumentides avaldiste puhul tühikuid mitte kasutada:

```
>>> def liida(x,y):
...     return x + y
...
>>> print liida(12,13+5)
30
>>>
```

Funktsiooni nimi võiks olla väikeste tähtedega, pikema nime osad eraldatud allkriipsuga `_`:

```
>>> def paneme_viis_juurde(x):
...     return x + 5
```

Klasside defineeerimine

Kuigi praegu veel on võmalik kasutada ka vana tüüpi klasse:

```
>>> class Jalgratas:
...     pass
```

on parem kasutada uut tüüpi klasse, mis on objekti alamklassid:

```
>>> class Jalgratas(object):
...     pass
```

Klasside nimed võiksid alata suurte tähtedega, kui on vaja pikemat nime, siis võiks see olla nn kaamelikirjas, ehk siis klassi nime moodustavad sõnad algavad samuti suure tähega:

```
>>> class KaameliKirjasKlass(object):
...     pass
```

Muutujate nimed

- Muutujate nimed võiksid olla tähenduslikud ja seotud väärtusega, mida nad kannavad:

  ```
  >>> nimi = 'Mari'
  ```

- Kui kasutatakse pikemaid muutja nimesid, siis võiksid nad olla nn kaamelikirjas, ehk siis muutuja nime moodustavad sõnad algavad suure tähega

  ```
  >>> eesNimi = 'Mari'
  >>> perekonnaNimi = 'Maasikas'
  ```

- Ära kasuta muutja nimeks võtmesõnu ka siis, kui see on tehniliselt võimalik
- Mõnikord on otstarbekas kasutada lühikesi muutjanimesid (eriti näiteks `lambda` funktsioonis):

  ```
  >>> hinded = [1, 2, 3, 4, 5]
  >>> headHinded = filter(lambda x: x > 3 , hinded)
  ```

Sõnede tähistamine

Sõnesid võib Pythonis märgendada mitut moodi: ühe või kolmekordsete jutumärkide või apostroofidega. Soovitav on kasutada läbi koodi ühesugust tähistust, vähemalt pidada kinni mingist süsteemist.

Mõnel moodulil on omad nõudmised sõnede märkimisele. Näiteks moodul `Qt` määrab, et kahekordsete jutmärkide vahel olevad sõned on need, millele võib määrata tõlkeid.

Ülesanded

Valik kursuse "Programmeerimine lingvistidele" ülesandeid.

1. Pythoni interpretaator

- Leia oma arvutist Python.
- Kui võimalik, siis installeeri ka oma arvutisse Python.
- Käivita Pythoni interpretaator ja proovi vähemalt neid tehteid ja näiteid, mis abimaterjalides toodud.
- Kirjuta vastuseks, kuidas õnnestus ja milline Pythoni versioon on kasutada.

2. Programmifail

    - Kirjuta programmitekst faili (võib kasutada ka näites toodut). Proovi seda käivitada. Saada vastuseks see fail, milles oleks ka kommentaariread, kuidas õnnestus.

3. Lähme külla kalale

    - Kirjuta programm, mis küsiks kasutaja käest sõna ja kui see lõppeb tähega "a", siis vastab tekstiga "Lähme külla <see sõna> ja "le" lõpp. Näiteks nii, et kui antud sõna on "maa", siis oleks vastus "lähme külla maale." Kui sõna ei lõppe a-ga, siis olgu vastus "Ma ei tunne sellist sõna".

4. Kuupäevade leidmine tekstist

    - Kirjuta Pythoni programm, mis leiaks etteantud tekstifailist laused, kus sisaldub kuupäev. Failis on iga lause omaette real.
    - Lisa programmile nii palju dokumentatsiooni, et sellest selguks, missuguseid kuupäevi programm tunneb ja missuguseid mitte.

5. Kirjuta Pythoni programm, mis näitaks iga sisestatud sõna kohta tema vokaalide ja konsonantide arvu ning tähemärkide arvu kokku. Kasuta objektorienteeritud lähenemist.

6. Jedi nime leidmine

    - Kirjuta Pythonis programm, mis leiaks Jedi nime vastavalt kasutaja poolt sisestatud andmetele.
    - Jedi eesnimi: Võta perekonnanime kolm esimest tähte ja liida eesnime kaks esimest tähte.
    - Jedi perekonnanimi: võta ema neiupõlvenime kolm esimest tähte ja lisa sünnilinna kolm esimest tähte.
    - Lisatingimused:
        - kasuta ühte funktsiooni andmete küsimiseks ja teist funktsiooni tulemuste arvutamiseks/esitamiseks
        - kontrolli suur- ja väiketähtede kasutamist
        - Kontrolli sisestatava sõne pikkust