. . . TAUTOJE MOKSLO YRA TIEK, KIEK JO YRA PARAŠYTA GIMTĄJA KALBA.

E. KRIŠČIŪNAS

VERTĖ PYTHON ENTUZIASTAI

PYTHON'O IR JO LOKALIZACIJOS VADOVĖLIS

XXXX LEIDYKLA

publikavo xxxx leidykla

www.vgtu.lt

First printing, October 2014

Turinys

Dedikuojama visiems esamiems ir būsimiems

Python'o bei Perūnio mylėtojams Autoriai

.

Įvadas

Python'as yra lengvai išmokstama, galinga programavimo kalba. Ji turi efektyvias aukšto lygio duomenų struktūras ir paprastą, bet efektyvų objektinio programavimo modelį. Dėl elegantiškos Python'o sintaksės, dinaminio tipizavimo ir interpretatoriaus, Python'as yra ideali kalba scenarijų rašymui ir greitam aplikacijų kūrimui įvairiose srityse įvairioms platformoms.

Python'o interpretatorių ir didelę standartinę biblioteką visoms pagrindinėms platformoms, bei jų išeities tekstus, galima parsisiųsti iš Python'o svetainės http://www.python.org/. Atsisiųstą kopiją vėliau galima laisvai platinti. Šiame vadovėlyje taip pat rasite informacijos apie kitus nemokamus Python'o modulius, programas, įrankius, bei papildomą dokumentaciją.

Python'o interpretatorius yra lengvai praplečiamas naujomis funkcijomis ir duomenų tipais naudojant C arba C++ (arba kitas kalbas, kurias galima kviesti iš C). Python'as taip pat gali būti naudojamas kaip praplėtimo kalba.

Šis vadovėlis neformaliai supažindina skaitytoją su pagrindinėmis Python'o kalbos koncepcijomis ir savybėmis. Jums tikrai bus paprasčiau, jei po ranka turėsite Python'o interpretatorių, nors visi pavyzdžiai yra išsamūs, todėl vadovėlį galima tiesiog skaityti.

Jei jums reikia standartinių objektų ir modulių aprašymo, žiūrėkite Python Library Reference_ dokumentą. Python Reference Manual_ formaliai aprašo kalbą. Jei norite rašyti plėtinius C ar C++ skaitykite Extending and Embedding the Python Interpreter_ ir Python/C API Reference_. Taip pat galima įsigyti knygų išsamiai aprašančių Python'ą.

Vertėjo pastaba: Aukščiau išvardintų šaltinių pavadinimai neišversti į lietuvių kalbą, nes patys šaltiniai neturi vertimo į lietuvių kalbą (2010-06-13).

Vadovėlyje nerasite išsamių kiekvienos savybės ar net gi dažnai naudojamų savybių aprašymų. Čia aprašytos Python'o savybės, kurias verta žinoti, vadovėlis padės suprasti kalbos pobūdį ir stilių. Perskaitę jį jūs galėsite skaityti ir rašyti Python'o modulius ir programas, būsite pasiruošę išmokti daugiau apie Python'o bibliotekos modulius aprašytus *Python Library Reference* dokumente.

Rekomenduojame peržvelgti skyrių apie :ref:`Vertimą <vertimas>`
ir :ref:`Terminų Žodyną <glossary>`.

Vertimas

Tikslas

Tikslas yra labai paprastas: išversti kažkiek informacijos apie Python'ą į lietuvių kalbą. Tikrai nesiekiama išversti visos Python'o dokumentacijos (bet jei taip ir atsitiks, tarkim per 50 metų, nematau nieko tame blogo).

Tikslinė auditorija

Kai buvau mažas ir kvailas geros literatūros apie programavimą lietuvių kalba rasti buvo labai sunku. Nei anglų nei rusų kalbos nemokėjau tiek, kad galėčiau skaityti ir tuo labiau suprasti. Man išties pasisekė, kad tuo metu jau egzistavo Jaunųjų Programuotojų Mokykla_, taigi jie man davė pirmą taip reikalingą spyrį, kad pradėčiau gaudytis kas kur kaip ir su kuo valgoma. Vėliau susidarė draugų ratas ir atsirado literatūros tiek anglų tiek rusų kalba (rusiškos deja taip ir nesupratau iš esmės).

Taigi pirmiausiai šis vertimas skirtas žmonėms, kurie dar nemoka anglų kalbos (galbūt tik pradeda mokytis) ir jau tikrai nebemoka rusiškai (tokia tad ta Nepriklausomybės karta). Nebijokite, jei visko nesuprasite iš pradžių (šis vadovėlis išties yra labai techniškas). Jei kas neaišku visada galite paklausti žinančių žmonių. Jų galite rasti Python el. konferencijoje_.

O vėliau jums tikriausiai vistiek teks išmokti anglų kalbą --- iš kitos pusės, tai nebus taip jau ir sudėtinga, kai būsite pagauti azarto ir reikės išsiaiškinti kaip reikia padaryti vieną ar kitą dalyką.

Beje, jei nepatenkate į aukščiau aprašytą tikslinę auditoriją, tai dar nereiškia, kad šis vadovėlis jums netinka.

Padėkos žodis

Dėkoju prie vertimo vienaip ar kitaip prisidėjusiems žmonėms: Adomui Paltanavičiui, Armandui Jarušauskui, Mantui Zimnickui, Maksim Norkin, Vytautui Šalteniui, Martynui Jociui, Martynui Sklizmantui, Albertui

Agėjavui, Mariui Gedminui, Jurgiui Pralgauskiui, Agniui Vasiliauskui, Audriui Kažukauskui, Bogdan M. Maryniuk, Vakarui Liutinkevičiui ir kitiems asmenims, kurie vienaip ar kitaip prisidėjo prie vertimo.

Jeigu norite prisidėti prie vertimo visada tai galite padaryti. Kaip? Perskaitykite kitame skyriuje.

Kaip prisidėti?

Prisidėti galima įvairiais būdais: naujais vertimais, pataisymais ir t.t. Pataisymus galite tiesiog atsiųsti man, bet jei norite tai galite daryti ir kaip tikri programuotojai.

Dokumentacija yra verčiama naudojantis bitbucket.org sistemoje sukurta Mercurial repozitorija: http://bitbucket.org/Dalius/python-tutorial-lt/

Susikurkite paskyrą http://bitbucket.org arba prisijunkite ten naudodamiesi OpenId (jūs žinoma rinksitės šį variantą!). Tada jums reikės šakoti (kaip teisingai verčiasi fork'inti?) python-tutorial-lt repozitoriją ir jeigu norėsite, kad jūsų vertimas būtų įtrauktas į pagrindinę repozitoriją tiesiog mane informuokite (galite tai padaryti paštu dalius@sandbox.lt).

Tiesa sakant, jūs neprivalote naudotis bitbucket.org, bet tai gali būti greitesnis ir paprastesnis būdas.

Šiuo metu angliški failai imami iš pagrindinės Python'o repozitorijos ir tiesiog verčiami: http://svn.python.org/projects/python/trunk/Doc/. Jei jūs norite versti kažką kito negu tai, kas dabar yra repozitorijoje, paanalizuokite Python'o dokumentaciją. Tiesa sakant, griežtų reikalavimų nėra, kad reikia versti būtent iš Python'o dokumentacijos.

Jei neaišku kaip versti žodį pasitikrinkite viename iš šių šaltinių: :ref:`glossary` arba http://www.likit.lt/en-lt/angl.html.

Kaip testuoti savo vertimą?

Aš naudoju tokį būdą (Ubuntu Linux sistemoje):

```
cd projects
hg clone http://bitbucket.org/ianb/virtualenv/
python virtualenv/virtualenv.py pytut
cd pytut
./bin/easy_install Sphinx
hg clone http://bitbucket.org/Dalius/python-tutorial-lt/
./bin/sphinx-build -b html python-tutorial-lt/ pytut
```

Jei norite sugeneruoti pdf'ą taip pat reikės įdiegti rst2pdf biblioteką:

```
./bin/easy_install rst2pdf
```

Prieš tai jums gali tekti įsirašyti kelis papildomus paketus:

```
sudo aptitude install build-essential
sudo aptitude install python-dev
```

Kitas variantas – naudoti buildout (Ubuntu Linux sistemoje):

```
cd projects
hg clone http://bitbucket.org/vakaras/python-tutorial-lt
make bootstrap
make buildout
```

Kai darbo aplinka paruošta, galima susigeneruoti HTML (PDF) ir iš karto jį atverti:

```
make show-html                      # Sugeneruoja HTML ir atveria
                                    # naršyklėje.
make show-pdf                       # Sugeneruoja PDF ir jį parodo.

make html                           # Tiesiog sugeneruoja HTML.
make pdf                            # Tiesiog sugeneruoja PDF.
```

Taip pat galima dirbti ir be virtualenv ar buildout aplinkos. Įdiekite reikalingas bibliotekas tiesiai į sistemą:

```
sudo apt-get install python-sphinx rst2pdf
```

Naudokitės paruoštu make failu:

```
make && firefox ./html/index.html
```

Šio būdo trūkumas yra toks, kad python-sphinx biblioteka yra pakankamai sena (be lietuviškų vertimų).

Vertėjai

Čia surašyti asmenys, kurie prisidėjo prie kiekvieno skyriaus vertimo:

- glossary.rst – Dalius Dobravolskas
- tutorial\index.rst – Dalius Dobravolskas
- tutorial\appetite.rst – Dalius Dobravolskas
- tutorial\interpreter.rst – Armandas Jarušauskas
- tutorial\introduction.rst – Adomas Paltanavičius
- tutorial\controlflow.rst – Dalius Dobravolskas

- tutorial\datastructures.rst – Adomas Paltanavičius, Dalius Dobravolskas
- tutorial\modules.rst – sirex, Dalius Dobravolskas
- tutorial\inputoutput.rst – Dalius Dobravolskas
- tutorial\errors.rst – Dalius Dobravolskas
- tutorial\classes.rst – Dalius Dobravolskas
- tutorial\stdlib.rst – Dalius Dobravolskas
- tutorial\stdlib2.rst – Dalius Dobravolskas
- tutorial\whatnow.rst – Dalius Dobravolskas
- tutorial\floatingpoint.rst – Dalius Dobravolskas
- python lokalizacija - Algirdas Maknickas Antano

Python lokalizacija

Apetitui sužadinti

Jeigu dirbate kompiuteriu anksčiau ar vėliau pastebėsite, kad yra užduočių, kurias norėtumėte automatizuoti. Pavyzdžiui, jums gali prireikti atlikti teksto pakeitimo operaciją dideliam tekstinių failų kiekiui arba pervadinti ir surikiuoti didelį nuotraukų kiekį ypatinga tvarka. Galbūt jūs norėtumėte parašyti mažą duombazę, specializuotą grafinę programą arba paprastą žaidimą.

Jei esate profesionalus programuotojas, tikriausiai jums reikia dirbti su keliomis C/C++/Java bibliotekomis, bet įprastas darbo ciklas rašai/kompiliuoji/testuoti/perkompiliuoji jums yra per lėtas. Galbūt jūs rašote testavimo įrankį tokiai bibliotekai ir testinio kodo rašymas yra varginanti užduotis. Galbūt jūs parašėte programą, kuri galėtų naudoti praplėtimo kalbą ir jūs nenorite sukurti visos naujos kalbos jūsų aplikacijai.

Python'as yra kalba kaip tik jums.

Žinoma, galite rašyti Unix'o arba Windows'ų skriptus realizuoti kai kurioms iš šių užduočių, nes skriptai tinka failų operacijoms ir tekstinių duomenų keitimui, bet nėra tinkami grafinėms aplikacijoms ar žaidimams. Jūs galite rašyti C/C++/Java programą, bet jums reikės tikrai daug laiko norint parašyti pačią pirmą programos versiją. Naudoti Python'ą yra paprasčiau, jis veikia Windows, Max OS X ir Unix-tipo operacinėse sistemose ir su juo darbą atliksite daug greičiau.

Python'ą paprasta naudoti, bet tuo pačiu tai yra pilnavertė programavimo kalba, kur kas geriau struktūrizuota ir lengviau palaikoma rašant didelės apimties programas, nei skriptai. Iš kitos pusės, Python'as gali pasiūlyti daugiau klaidų tikrinimo negu C ir būdama *labai-aukšto-lygio kalba* Python'as turi standartinių aukšto lygio duomenų tipų kaip kad lanksčių masyvų ir žodynų. Dėl bendresnių duomenų tipų Python'as gali būti pritaikytas platesniam problemų spektrui nei Awk ar netgi Perl ir vėlgi dauguma dalykų padaromi taip pat lengvai Python'u kaip ir paminėtomis kalbomis.

Python'as leidžia jūsų programas išskaidyti į modulius, kurie gali būti panaudoti kitose Python'o programose. Python'e yra daug standartinių modulių, kuriuos jūs galite naudoti kaip pagrindą savo programoms arba kaip pavyzdį pradedant mokytis programuoti Python'u.

Kai kurie iš šių modulių leidžia atlikti failų operacijas, sisteminius kvietimus, lizdų (angl. socket) operacijas arba netgi grafines naudotojo aplinkas naudojant bibliotekas kaip Tk.

Python'as yra interpretuojama kalba, o tai reiškia kad jūs galite sutaupyti daug laiko, nes programos kūrimo metu nereikia kompiliuoti ir saistyti. Interpretatorius gali būti naudojamas interaktyviai, todėl jūs galite lengvai eksperimentuoti su įvairiomis kalbos galimybėmis, rašyti programas pasibandymui arba testuoti funkcijas programuodami iš apačios viršun. Tai taip pat tiesiog patogus skaičiuotuvas.

Python'as leidžia rašyti programas kompaktiškai ir skaitomai. Programos parašytos Python'u dažniausiasi yra trumpesnės negu ekvivalenčios programos parašytos C, C++ arba Java dėl kelių priežasčių:

- aukšto lygio duomenų tipai leidžia išreikšti sudėtingas operacijas paprastais sakiniais;
- sakinių grupavimas atliekamas stumdant tekstą užuot naudojus pradžios ir pabaigos skliaustelius;
- kintamųjų ar argumentų nereikia deklaruoti.

Python'as yra *išplečiamas*: jeigu jūs mokate programuoti C tada labai paprasta pridėti naujas funkcijas ar modulius į interpretatorių. Tai gali praversti norint atlikti operacijas maksimaliu greičiu arba susaistyti Python'o programas su bibliotekomis, kurias turime tik sukompiliuotame formate (pvz.: tiekėjui specifines grafines bibioletekas). Jeigu jums reikia, jūs galite susieti Python'o interpretatorių su programa parašyta C ir naudoti jį kaip praplėtimų arba komandinę kalbą tai programai.

Tarp kitko, programa yra pavadinta pagal BBC laidą „Monty Python's Flying Circus“ ir neturi nieko bendro su reptilijomis. Nuorodos į Monty Python ištraukas dokumentacijoje yra ne tik leidžiamos, bet ir skatinamos!

Dabar, kai jūs jau esate susižavėję Python'u, jūs norite jį panagrinėti detaliau. Kadangi geriausias būdas išmokti kalbą yra ją naudoti, vadovėlis kviečia jus žaisti su Python'o interpretatoriumi beskaitant.

Kitoje dalyje paaiškinta, kaip naudotis interpretatoriumi. Tai nėra labai įdomi informacija, bet būtina norint išbandyti vėliau pateiktus pavyzdžius.

Likęs vadovėlis supažindina su įvairiomis Python kalbos savybėmis ir sistema per pavyzdžius. Pradedama nuo paprastų reiškinių, sakinių ir duomenų struktūrų, vėliau paaiškinamos funkcijos ir moduliai, ir galiausia paliečiami pažangesnės koncepsijos kaip kad išimtys ir naudotojo apibrėžtos klasės.

Naudojimasis Python'o interpretatoriumi

Interpretatoriaus paleidimas

Python'o interpretatorius dažniausiai įdiegiamas į :file:`/usr/local/bin/python`; pridėję :file:`/usr/local/bin` į savo Unix apvalkalo (angl. *shell*) paieškos kelią, galėsite Python'ą paleisti įrašę komandą:

```
python
```

Kadangi direktorija, kurioje „gyvena“ interpretatorius, yra pasirenkama diegimo metu, aukščiau minėtas kelias nebūtinai yra teisingas; pasitikslinkite detales su savo administratoriumi (pvz.: :file:`/usr/local/python` yra populiari alternatyva).

Windows sistemose, Python'as įdiegiamas kataloge :file:`C:\Python26`, tačiau šis nustatymas gali būti pakeistas įdiegimo metu. Norėdami pridėti šią direktoriją į paieškos kelią, galite įrašyti tokią komandą į DOS terminalą:

```
set path=%path%;C:\python26
```

Įrašę failo pabaigos simbolį (:kbd:`Control-D` Unix sistemose, :kbd:`Control-Z` Windows) Python'o komandinėje eilutėje išjungsite interpretatorių. Jei tai nesuveikia, galite išjungti interpretatorių surinkę : `import sys; sys.exit()`.

Teksto rinkimas interpretatoriuje nėra labai rafinuotas. Unix sistemose, Python'ą įdiegęs žmogus galėjo pridėti GNU readline bibliotekos palaikymą, taip įgalindamas naudotojus naudoti interaktyvų redagavimą, bei komandų istoriją. Turbūt greičiausias būdas patikrinti, ar komandinės eilutės redagavimas yra palaikomas -- surinkti Control-P interpretatoriuje. Jei išgirdote pyptelėjimą --redagavimas įjungtas. Jei pyptelėjimo nesigirdi arba išvedama `^P`, redagavimo palaikymas neįjungtas; galėsite tik trinti simbolius, toje pačioje eilutėje, naudodami Backspace klavišą.

Interpretatoriaus darbas šiek tiek primena Unix apvalkalą: kai interpretatorius iškviečiamas prijungus standartinę įvestį prie TTY įrenginio, jis nuskaito ir vykdo komandas interaktyviai; kai iškviečiamas su failo pavadinimo parametru arba failu pateikiamu per standartinę įvestį, interpretatorius nuskaito ir įvykdo *skriptą* esantį faile.

Antras būdas paleisti interpretatorių yra `python -c komanda [argumentai] ...`. Šiuo būdu paleistas interpretatorius įvykdys sakinius pateiktus *komanda* parametre, analogiškai apvalkalo :option:`-c` parinkčiai. Kadangi Python'o sakiniuose dažnai būna tarpų ar kitų simbolių, kuriuos apvalkalas laiko specialiais, patartina visą *komandos* turinį apgaubti apostrofais.

Kai kurie Python'o moduliai yra naudingi kaip skriptai. Jie gali būti iškviesti su `python -m modulis [argumentai] ...`. Ši eilutė įvykdys failo *modulis* turinį taip, lyg būtumėte parašę to failo pilną vardą komandinėje eilutėje.

Atkreipkite dėmesį, kad `python failas` ir `python <failas` skiriasi. Pastaruoju atveju, įvesties reikalavimas, pavyzdžiui kviečiant input() ar raw_input(), yra užpildomas iš *failo*. Kadangi failas yra nuskaitomas dar prieš pradedant vykdyti skriptą, failo pabaiga bus rasta iškart. Pirmu atveju (kurio jūs dažniausiai ir norite), duomenys bus imami iš įrenginio, kuris yra sujungtas su standartine interpretatoriaus įvestimi.

Kartais, kai vykdomas skriptas, gali būti naudinga persijungti į interaktyvų režimą skripto vykdymo pabaigoje. Tai galima padaryti su :option:`-i` parinktimi prieš skriptą. (Šis būdas neveikia, jei skriptas yra nuskaitomas iš standartinės įvesties dėl tos pačios priežasties, kuri aprašyta praėjusioje pastraipoje.)

Argumentų perdavimas

Kai pateikti, skripto pavadinimas ir kiti argumentai yra perduodami skriptui, naudojant `sys.argv` kintamąjį, eilučių sąrašo (angl. *list of strings*) pavidalu. Sąrašo ilgis ne mažesnis už vienetą; kai skriptas ir argumentai neperduodami, `sys.argv[0]` bus tuščia eilutė. Kai skripto vardas yra pateiktas kaip `-` (kas reiškia standartinę įvestį), `sys.argv[0]` įgauna `'-'` reikšmę. Kai naudojama :option:`-c` *komanda*, `sys.argv[0]` reikšmė bus `'-c'`. Jei naudojamas :option:`-m` *modulis*, `sys.argv[0]` reikšmė bus pilnas modulio vardas. Parinktys rastos po :option:`-c` *komanda* ar :option:`-m` *modulis* nėra apdorojamos interpretatoriaus, jos paliekamos `sys.argv` sąraše iš kur, reikalui esant, jas gali pasiekti komanda ar modulis.

Interaktyvus režimas

Kai komandos yra nuskaitomos iš TTY įrenginio, sakome, kad interpretatorius yra interaktyviajame režime. Šiame režime *pirminis raginimas* (dažniausiai trys „daugiau“ ženklai >>>) reiškia, kad interpretatorius yra pasiruošęs nuskaityti kitą komandą; antrinis raginimas (trys taškai `...`) reiškia, kad laukiamas prieš tai įvestos komandos pratęsimas. Prieš atspausdindamas pirmąjį raginimą, interpretato-

rius parodo pasisveikinimo žinutę, kurioje nurodoma interpretatoriaus versija bei autorinių teisių pranešimas:

```
python
Python 2.6 (#1, Feb 28 2007, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Tęsiamosios eilutės yra naudojamos įvedinėjant komandas užimančias daugiau nei vieną eilutę. Kaip pavyzdį galime naudoti šį :keyword:`jei` sąlygos sakinį [1]:

[1] pasaulis_yra_plokscias = 1
jei pasaulis_yra_plokscias:
... rodo "Atsargiai, nenukriskite!"

```
>>> pasaulis_yra_plokscias = 1
>>> if pasaulis_yra_plokscias:
...     print "Atsargiai, nenukriskite!"
...
Atsargiai, nenukriskite!
```

Interpretatorius ir jo aplinka

Klaidų valdymas

Kai įvyksta klaida, interpretatorius atspausdina klaidos pranešimą ir dėklo pėdsaką (angl. *stack trace*). Jei klaida įvyksta interaktyviajame režime, interpretatorius tiesiog grįžta į pirminį raginimą; jei įvestis buvo nuskaityta iš failo, interpretatorius atspausdins dėklo pėdsaką ir baigs darbą grąžindamas atitinkamą (nelygų nuliui) būsenos kodą. Išimtys, suvaldytos naudojant :keyword:`except` sakinį :keyword:`try` sakinyje, šiame kontekste nėra laikomos klaidomis. Kai kurios klaidos yra besąlygiškai lemtingos ir priverčia interpretatorių baigti darbą su klaidos būsena nelygia nuliui; prie tokių klaidų priskiriami vidiniai neatitikimai bei kai kurie atminties trūkumo atvejai. Visi klaidų pranešimai yra surašomi į standartinį klaidų srautą; įprastas tekstas, grąžinamas vykdomų komandų, yra rašomas į standartinę išvestį.

Įrašę pertraukties simbolį (paprastai Control-C arba DEL) į pirminį arba antrinį raginimą, nutrauksite įvestį ir interpretatorius grįš į pirminį raginimą. [#]_ Jei pertraukties simbolis įrašomas kol vykdoma komanda, interpretatorius sukelia išimtį, kuri gali būti suvaldyta :keyword:`try` teiginiu.

Python'o vykdomieji skriptai

BSD tipo Unix sistemose, Python'o skriptai gali būti padaryti vykdomaisiais, taip pat, kaip apvalkalo skriptai. Tam failo pradžioje reikia pridėti tokią eilutę:

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

Kad ši eilutė veiktų, interpretatorius turi būti naudotojo kelyje (:envvar:`PATH`), o failui turi būti suteikta vykdymo būsena. Pirmi du simboliai faile privalo būti `#!`. Kai kuriose platformose pirmoji eilutė turi baigtis Unix tipo eilutės pabaigos simboliu (`'\n'`), o ne Windows (`'\r\n'`). Atkreipkite dėmesį, kad grotelės `#` Python'e yra naudojamos kaip komentaro pradžios simbolis.

Skriptui vykdymo būseną galite suteikti pasinaudoję :program:`chmod` komanda:

```
$ chmod +x skriptas.py
```

Windows sistemose nėra „vykdomosios būsenos“ žymėjimo. Python'o diegimo programa automatiškai susieja `.py` rinkmenas su `python.exe`, todėl spragtelėjus du kartus ant Python'o failo, jis bus įvykdytas kaip skriptas. Failas taip pat gali baigtis `.pyw` plėtiniu. Tokiu atveju terminalo langas nebus rodomas, kaip įprasta.

Išeities teksto koduotė

ASCII nėra vienintelis kodavimas, kuris gali būti naudojamas Python'o išeities tekstuose. Geriausias būdas nurodyti savo koduotę yra įdėti dar vieną specialų komentarą iš karto po `#!` eilute [2]:

[2] perūnis koduotė visada utf-8

 # -- coding: <koduotė> --

Su šia deklaracija, visi simboliai faile bus traktuojami, kaip turintys *koduotė* kodavimą, be to galėsite rašyti Unikodo simbolius tiesiogiai pasirinktoje koduotėje. Galimų koduočių sąrašą galite rasti Python'o bibliotekų rodyklėje, :mod:`codecs` skiltyje.

Pavyzdys: norėdami rašyti Unikodo simbolius įskaitant ir Euro valiutos simbolį, galite naudoti ISO-8859-15 kodavimą, kur Euro simbolio eilės numeris yra 164. Šis skriptas atspausdins reikšmę 8364 (Kodas atitinkantis Euro simbolį Unikode) ir baigs darbą:

```
# -*- coding: iso-8859-15 -*-

valiuta = u"€"
print ord(valiuta)
```

Jei jūsų teksto redaktorius turi galimybę išsaugoti failą kaip `UTF-8` su UTF-8 *baitų eiliškumo žyme* (angl. *byte order mark, BOM*), tuomet galite naudoti šį būdą vietoj koduotės deklaracijos. IDLE redaktorius turi tokią galimybę, jei pasirinktas `Options/General/Default Source Encoding/UTF-8` nustatymas. Atkreipkite dėmesį, kad senesnės Python'o laidos (2.2 ir ankstesnės) neatpažįsta šio parašo. Taip pat operacinės sistemos

neatpažįsta BOM skriptuose, kurie prasideda eilute su `#!` (naudojama tik Unix sistemose).

Naudojant UTF-8 (su koduotės deklaracija ar BOM parašu) komentaruose ir simbolių eilutėse vienu metu gali būti naudojami daugumos pasaulio kalbų simboliai. Ne ASCII simboliai negali būti naudojami kintamųjų, funkcijų, klasių ir t.t. *varduose.* Galimybė naudoti unikodo simbolius yra palaikoma nuo Python'o 3 versijos. Kad visi simboliai būtų atvaizduoti tvarkingai, jūsų redaktorius turi atpažinti UTF-8 rinkmenas ir naudoti šriftą, palaikantį visus faile esančius simbolius.

Interaktyvaus režimo paleidimo failas

Naudojant Python'ą interaktyviai, gali būti naudinga įvykdyti kokias nors komandas kaskart paleidžiant interpretatorių. Tai atlikti galite pakeitę aplinkos kintamojo :envvar:`PYTHONSTARTUP` reikšmę į failo, su jūsų paleidimo komandomis, vardą. Šis būdas yra panašus į Unix apvalkalo :file:`.profile` galimybę.

Ši rinkmena yra nuskaitoma tik interaktyvių sesijų metu, bet ne tada, kai Python'as nuskaito komandas iš skripto ar kai :file:`/dev/tty` yra pateiktas kaip konkretus įvesties šaltinis (nors šiuo atveju interpretatorius elgsis taip pat kaip ir interaktyviajame režime). Paleidimo failo komandos yra vykdomos toje pačioje vardų srityje, kurioje vykdomos interaktyvios komandos, todėl failo sukurti ar importuoti objektai gali būti naudojami iškart interaktyviojoje sesijoje. Šiame faile taip pat galite keisti ir raginimus `sys.ps1` bei `sys.ps2`.

Jei norite nuskaityti papildomus paleidimo failus, galite tai padaryti globaliame paleidimo faile. Pavyzdžiui: `if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`. Jei norite naudoti paleidimo failą skripte, turite tai aprašyti [3] :

```
import os
rinkmena = os.environ.get('PYTHONSTARTUP')
if rinkmena and os.path.isfile(rinkmena):
    execfile(rinkmena)
```

[3] naudoja os
rinkmena =
os.environ.get('PYTHONSTARTUP')
jei rinkmena ir os.path.isfile(rinkmena):
execfile(rinkmena)

Pastabos darbui su lokalizacija

Įvadas į Python'ą

Pavyzdžiuose, pateikiamuose toliau, įvedamas komandas bei jų išvestį galima atskirti pagal tai, ar eilutė prasideda raginimu (>>> ir . . .). Norėdami pakartoti pavyzdį savarankiškai, perrašykite viską, kas eina pavyzdyje po raginimo. Eilutės, neprasidedančios raginimu, yra interpretatoriaus išvestis. Atkreipkite dėmesį, jog norėdami užbaigti komandą, užimančią daugiau nei vieną eilutę (tuo metu bus rodomas antrasis raginimo variantas . . .), turėsite įvesti tuščią eilutę.

Dauguma pavyzdžių, net ir tie, kurie prasideda raginimu, turi paaiškinamuosius komentarus. Python'e komentarai pradedami simboliu # ir tęsiasi iki eilutės pabaigos. Komentarai gali prasidėti eilutės pradžioje, taip pat po tarpų arba po kodo, tačiau ne simbolių eilutės viduje. Simbolių eilutėje # tėra paprastas simbolis, neatliekantis jokios papildomos funkcijos. Kadangi komentarai paaiškina kodą, tačiau Python'as jų niekaip neinterpretuoja, bandydami pateiktus kodo fragmentus interaktyviame apvalkale komentarus galite tiesiog praleisti.

Pora pavyzdžių:

```
# čia pirmasis komentaras
SPAM = 1                 # o čia -- antrasis komentaras
                         # ...ir trečiasis!
STRING = "# Tai nėra komentaras."
```

Python'as kaip skaičiuoklis

Pradėkime išbandydami porą paprastų Python'o komandų. Pasileiskite interpretatorių ir luktelėkite, kol pasirodys raginimas >>>. (Neturėtų užtrukti.)

Skaičiai

Interpretatorius veikia kaip paprastas skaičiuoklis -- jūs įvedate reiškinį, o interpretatorius užrašo rezultatą. Reiškinių sintaksė paprasta: operatoriai +, -, * bei / veikia kaip ir daugumoje kitų kalbų (tarkime, Paskalyje arba C). Grupavimui galima naudoti skliaustelius. Pavyzdžiui:

```
>>> 2+2
4
>>> # Tai komentaras
... 2+2
4
>>> 2+2  # o čia -- komentaras toje pačioje eilutėje, kaip ir kodas
4
>>> (50-5*6)/4
5
>>> # Sveikų skaičių dalyba apskaičiuoja sveiką dalį:
... 7/3
2
>>> 7/-3
-3
```

Lygybės ženklas (`'='`) naudojamas priskirti reikšmę kintamajam. Atlikus priskyrimą nėra parodomas joks rezultatas [4]:

[4] plotis = 20
aukštis = 5*9
plotis * aukštis

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Reikšmę galima priskirti iš karto keliems kintamiesiems:

```
>>> x = y = z = 0  # Priskiriame nulį x, y ir z
>>> x
0
>>> y
0
>>> z
0
```

Prieš naudojant, kintamieji turi būti apibrėžti (jiems priskirta reikšmė), kitaip įvyks klaida:

```
>>> # pabandome gauti neapibrėžto kintamojo reikšmę
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Taip pat galima dirbti su slankaus kablelio skaičiais. Jeigu naudosime vieną iš anksčiau aptartų operatorių su argumentais, kurių vienas -- sveikas skaičius, o kitas -- slankaus kablelio skaičius, tuomet rezultatas bus slankaus kablelio:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Be to, galima naudoti ir kompleksinius skaičius. Menamoji dalis aprašoma naudojant galūnę `j` arba `J`. Kompleksiniai skaičiai su nenuline realiąja dalimi užrašomi kaip `(realioji+menamojiJ)`, arba pasinaudojama `complex(realioji, menamoji)` funkcija.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Kompleksiniai skaičiai visuomet aprašomi dviem slankaus kablelio skaičiais, realiąja ir menamąja dalimi. Norėdami išskirti šias dalis iš kompleksinio skaičiaus *z*, naudokite `z.real` (realiąjai daliai) ir `z.imag` (menamąjai daliai).

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Sveikų ir slankaus kablelio skaičių konvertavimo funkcijos (`float`, `int` ir `long`) neveikia kompleksiniams skaičiams -- nes nėra vienaprasmio būdo tai atlikti. Naudodami `abs(z)` gausite kompleksinio skaičiaus modulį (slankaus kablelio skaičiaus pavidalu). Jau minėtas `z.real` leis gauti realiąją kompleksinio skaičiaus dalį. [5]

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
```

[5] Bandymas versti į float viengubo tikslmo realių skaičių tipą duoda klaidą. sqrt - šaknies traukimo funkcija. abs - absolučios vertės skiačiavimo funkcija.

```
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

Interaktyvioje veiksenoje, paskutinė atspausdinta reikšmė yra priskiriama kintamajam _. Tai reiškia, kad jums naudojant Python'ą kaip skaičiuoklį, kai kuriuos skaičiavimus galima atlikti daug paprasčiau. [6]

[6] round - apvalinimo orimu tikslumu funkcija

```
>>> mokestis = 12.5 / 100
>>> kaina = 100.50
>>> kaina * mokestis
12.5625
>>> kaina + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Su šiuo kintamuoju turėtų būti elgiamasi taip, lyg jis būtų skirtas tik skaitymui. Nepriskirkite jam reikšmės, nes taip sukursite nesusijusį kintamąjį, kurio vardas neleistų prieiti prie tikrojo magiško Python'o kintamojo.

Eilutės

Python'as dirba ne tik su skaičiais, bet ir su simbolių eilutėmis. Jas galima užrašyti dviem būdais --- apsupti viengubomis arba dvigubomis kabutėmis:

```
>>> '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.'
```

Simbolių eilutės gali aprėpti porą eilučių. Naudojant kairinį brūkšnį eilutės gale galima nurodyti, kad toliau einanti eilutė yra šios tęsinys [7]:

[7] hello priskiriama simbolių eilutė: "Tai labai ilga simbolių seka per kelias eilutes kaip tai būyų C kalboje. Pažymėtina, kad tarpas perkeltos eilutės pradžioje svarbus."
rodo hello

```
>>> hello = "This is a rather long string containing\n\
... several lines of text just as you would do in C.\n\
...     Note that whitespace at the beginning of the line is\
... significant."
>>> print hello
```

Turėkite omenyje, kad kairinis brūkšnys sujungia gretimas eilutes į vieną, todėl naujas eilutes reikia sudaryti naudojant `\n`. Ankstesnis pavyzdys ekrane atspausdintų:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

Kita vertus, jeigu eilutę pažymime kaip „neapdorojamą“ (angl. *raw*), tuomet `\n` nepradeda naujos eilutės. Neapdorojamose eilutėse kairiniai brūkšniai taip pat praranda savo reikšmę (t.y. lieka kairiniais brūkšniais), tačiau skirstymas į eilutes nėra prarandamas -- lieka taip, kaip užrašyta [8]:

[8] rodo hello

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print hello
```

atspausdintų:

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

Dar galima eilutes apgaubti poromis trigubų kabučių: `"""` arba `'''`. Eilučių pabaigų nereikia užrašyti specialiai, bus paliekamas toks skaidymas į eilutes, kaip užrašyme.

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

ekrane išvestų:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Interpretatorius atspausdina operacijų su eilutėmis rezultatus visiškai taip pat, kaip eilutės yra įvedamos: apgaubia kabutėmis iš šonų, o

viduje eilutės esančios kabutės užrašomos pridedant kairinį brūkšnį. Jeigu eilutės viduje yra vienguba kabutė, eilutė spausdinama apgaubta dvigubomis kabutėmis. Kitais atvejais apgaubiama viengubomis kabutėmis. (Komanda :keyword:`print`, kurią aptarsime kiek vėliau, gali būti naudojama eilutėms atspausdinti be apgaubiančių kabučių.)

Eilutės gali būti sujungtos (pridėtos viena prie kitos) naudojant + operatorių bei pakartotos keletą kartų su $*$ operatoriumi:[9]

[9] word - kitamasis pavadinimu žodis

```
>>> word = 'Pagalb' + 'a'
>>> word
'Pagalba'
>>> '<' + word*5 + '>'
'<PagalbaPagalbaPagalbaPagalbaPagalba>'
```

Dvi eilutės, užrašytos viena po kitos, yra automatiškai sujungiamos. Taigi pirmąją eilutę pavyzdyje galėjome užrašyti tiesiog kaip `word = 'Pagalb' 'a'`; turėkite omenyje, kad šitaip galima daryti tik su rankomis užrašytomis eilutėmis, o ne su operacijų rezultatais:[10]

[10] string vertimas - simbolių eilutė

```
>>> 'str' 'ing'             #  <-  Taisyklinga
'string'
>>> 'str'.strip() + 'ing'   #  <-  Taisyklinga
'string'
>>> 'str'.strip() 'ing'     #  <-  Netaisyklinga
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                  ^
SyntaxError: invalid syntax
```

Eilutės gali būti indeksuojamos; kaip ir C kalboje, pirmasis eilutės simbolis atitinka indeksą 0. Python'o kalboje nėra atskiro tipo simboliams aprašyti; simbolis --tai eilutė, kurios dydis -- vienas simbolis. Eilučių dalys gali būti nurodomos naudojant :term:`atkarpos notaciją <atkarpa>` (angl. *slice notation*): du indeksai atskirti dvitaškiu.

```
>>> word
'Pagalba'
>>> word[4]
'l'
>>> word[0:2]
'Pa'
>>> word[2:4]
'ga'
```

Atkarpos indeksai gali būti nenurodyti; jeigu praleistas pirmasis indeksas, vietoje jo naudojamas nulis, o praleistas antrasis indeksas tapatus eilutės ilgio nurodymui.

```
>>> word[:2]    # Du pirmi simboliai
'Pa'
>>> word[2:]    # Viskas, kas eina po dviejų pirmų simbolių
'galba'
```

Priešingai negu C kalboje, Python'o eilutės negali būti keičiamos. Bandant pakeisti eilutės simbolį nurodant indeksą įvyksta klaida:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Tačiau naujų eilučių sukūrimas sudedant turimas yra paprastas ir efektyvus:

```
>>> 't' + word[1:]
'tagalba'
>>> 'Kav' + word[1]
'Kava'
```

Naudinga atkarpų operacijų savybė: `s[:i] + s[i:]` visuomet lygu `s`.

```
>>> word[:2] + word[2:]
'Pagalba'
>>> word[:3] + word[3:]
'Pagalba'
```

Išeinantys iš ribų indeksai yra tvarkingai apdorojami. Per didelis indeksas pakeičiamas eilutės ilgiu. Jeigu aktarpos pradžios indeksas didesnis už pabaigos indeksą, gausime tuščią eilutę.

```
>>> word[1:100]
'Pagalba'
>>> word[10:]
''
>>> word[2:1]
''
```

Kaip indeksus galima naudoti ir neigiamus skaičius, tokiu atveju skaičiuojama nuo dešinės. Pavyzdžiui:

```
>>> word[-1]     # Paskutinis simbolis
```

```
'a'
>>> word[-2]     # Priešpaskutinis simbolis
'b'
>>> word[-2:]    # Du paskutiniai simboliai
'ba'
>>> word[:-2]    # Viskas iki dviejų paskutinių simbolių
'Pagal'
```

Pastebėkite, kad -0 iš tiesų yra tas pats kaip ir 0, todėl šiuo atveju nuo dešinės nėra skaičiuojama.

```
>>> word[-0]     # (nes -0 lygu 0)
'P'
```

Išeinantys iš ribų neigiami indeksai yra apkarpomi, tačiau tai galioja tik atkarpoms:

```
>>> word[-100:]
'Pagalba'
>>> word[-10]    # klaida
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Bus lengviau atsiminti, kaip veikia atkarpos, jeigu galvosite apie indeksus kaip apie rodykles *tarp* simbolių, o eilutės kraštą prieš pirmą simbolį kaip turintį nulinį indeksą. Tuomet eilutės, sudarytos iš *n* simbolių, dešinys kraštas turės indeksą *n*.

```
+---+---+---+---+---+---+---+
| P | a | g | a | l | b | a |
+---+---+---+---+---+---+---+
0   1   2   3   4   5   6   7

  -7 -6 -5 -4 -3 -2 -1
```

Pirma skaičių eilutė parodo indeksų 0..7 vietas simbolių eilutėje. Antroji eilutė -- atitinkamai atvaizduoja neigiamus indeksus. Tada atkarpa nuo *i* iki *j* susideda iš visų simbolių, esančių tarp *i* ir *j*.

Neneigiamiems indeksams, atkarpos ilgis yra indeksų skirtumas (jeigu abu indeksai yra eilutės ribose). Tarkime, atkarpos `word[1:3]` ilgis yra 2.

Standartinė funkcija :func:`len` grąžina eilutės ilgį.

>>> s = 'supercalifragilisticexpialidocious' >>> len(s) 34

Unikodo eilutės

Pradedant Python'o versija 2.0 programuotojams pateikiamas naujas duomenų tipas, skirtas tekstinių duomenų saugojimui: Unikodo objektai. Jis gali būti naudojamas saugoti bei operuoti Unikodu paremtais duomenimis (daugiau informacijos rasite http://lt.wikipedia.org/wiki/Unikodas bei http://www.unicode.org). Šis duomenų tipas yra suderinamas su paprastomis simbolių eilutėmis: prireikus pakeičiamas automatiškai.

Didelis Unikodo pranašumas yra tai, kad šis standartas aprašo visus ženklus: tiek naudojamus dabartiniuose, tiek naudotus senoviniuose tekstuose. Anksčiau, prieš sukuriant Unikodą, buvo naudojamos 256-ių rašto ženklų kodavimo lentelės. Kiekvienas tekstas buvo susietas su kuria nors iš jų. Tokia sistema sukeldavo labai daug maišaties, ypač ten, kur tai buvo susiję su programinės įrangos daugiakalbyste. Unikodas šias problemas išsprendžia pristatydamas vientisą kodų lentelę, kurią galima naudoti visoms rašto sistemoms

Unikodo eilučių sukūrimas Python'e yra ne ką sudėtingesnis negu paprastų eilučių:

```
>>> u'Labas, pasauli!'
u'Labas, pasauli!'
```

Mažoji `'u'` prieš kabutę nurodo, kad aprašome Unikodo eilutę. Jeigu norite eilutėje naudoti specialius simbolius, galite tai padaryti naudodami specialią Python'o sintaksę.

```
>>> u'Sveikas,\u0020pasauli!'
u'Sveikas, pasauli!'
```

Čia užrašyta seka `\u0020` reiškia Unikodo simbolio, kurio kodas 0x0020 (o tai yra tarpo simbolis), įterpimą eilutėje.

Visi kiti simboliai interpretuojami pagal tai, kokį Unikodo kodą jie atitinka. Jeigu jūs užrašote eilutes naudodami Latin-1 koduotę (turinčią daugumai vakarų Europos kalbų skirtų simbolių), tikrai pamatysite tai, kad pirmi 256 Unikodo rašmenys sutampa su Latin-1 kodų lentele.

Ypatingiems poreikiams galima naudoti neapdorojamas (angl. *raw*) eilutes --visai taip pat, kaip ir paprastų eilučių atveju. Norėdami tokias eilutes įvesti, prieš atidarančią kabutę parašykite 'ur'. Tuomet `\uXXXX` užrašymas bus apdorojamas tik tuomet, kai kairinių brūkšnių skaičius prieš 'u' yra nelyginis.

```
>>> ur'Sveikas,\u0020pasauli!'
u'Sveikas, pasauli!'
>>> ur'Sveikas,\\u0020pasauli!'
u'Sveikas,\\\\u0020pasauli!'
```

Šis režimas naudingas tada, kai reikia įvesti daug kairinių brūkšnių, pavyzdžiui reguliariuosius reiškinius (angl. *regular expression*).

Be šių užrašymo būdų, Python'as pateikia ir daugiau būdų Unikodo eilutėms sukurti tuomet, kai žinoma teksto koduotė.

Standartinė funkcija :func:`unicode` moka iškoduoti ir užkoduoti daug tekstų koduočių. Keletas žinomesnių yra *Latin-1*, *ASCII*, *UTF-8*, ir *UTF-16*. Dvi paskutinės koduotės yra vadinamos *kintamo ilgio koduotėmis*, kadangi naudojant šias koduotes vienas Unikodo simbolis gali būti paverčiamas vienu arba daugiau baitų. Pagal nutylėjimą dažniausiai naudojama ASCII koduotė, kuri turi 127 simbolius, atitinkančius pirmus 127 Unikodo simbolius. Ši koduotė atmes visus kitus simbolius, pranešdama apie klaidą. Spausdinant Unikodo eilutę ekrane arba faile, naudojama funkcija :func:`str`, kuri konvertuoja eilutę naudodama numatytąją teksto koduotę.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not in range(128)
```

Galite paversti Unikodo eilutę į 8 bitų simbolių eilutę nurodydami norimą teksto koduotę su :func:`encode` metodu, kuriam būtinas vienas parametras --koduotės pavadinimas. Teksto koduočių vardus rekomenduojama užrašyti mažosiomis raidėmis.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Jeigu jūs turite duomenis žinomoje teksto koduotėje ir norite iš jų gauti Unikodo eilutę, galite naudoti :func:`unicode` funkciją, kartu nurodydami koduotės pavadinimą.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

Sąrašai

Python'as pateikia daug *sudėtinių* duomenų tipų, naudojamų apjungti kitas reikšmes. Lanksčiausias iš tokių tipų yra *sąrašas*, kuris aprašomas laužtiniais skliausteliais apgaubiant kelis iš eilės einančius, vienas nuo kito kableliais atskirtus, elementus. Sąrašo elementai gali būti skirtingų tipų.

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

Kaip ir simbolių eilučių indeksai, sąrašų indeksai prasideda nuo 0, sąrašai gali būti atkirpti, sujungti ir taip toliau:

```
>>> 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!']
```

Priešingai negu simbolių eilutės, kurios yra nekintamos, sąrašo elementus galima pakeisti:

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

Atkarpoms (angl. *slice*) taip pat galima priskirti reikšmes, net jeigu tai pakeičia sąrašo dydį arba jį ištuština:

```
>>> # Pakeiskime porą elementų:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Panaikinkime keletą:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Įterpkime:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Įterpkime sąrašo kopiją į paties pradžią
>>> a[:0] = a
```

```
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Ištuštinkime sąrašą: pakeiskime visus elementus tuščiu sąrašu
>>> a[:] = []
>>> a
[]
```

Standartinė funkcija :func:`len` tinka ir sąrašams:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Galima sąrašus sudėti į sąrašus (sukurti sąrašus, kurių elementai yra kiti sąrašai), pavyzdžiui:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Žiūrėkite skyrių 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Turėkite omenyje, kad paskutiniame pavyzdyje `p[1]` ir `q` iš tiesų nurodo tą patį objektą! *Objektų semantiką* aptarsime vėliau.

Pirmieji žingsniai link programavimo

Be abejonės, Python'as gali būti naudojamas daug sudėtingesniems darbams, negu apskaičiuoti du plius du. Pavyzdžiui, galime apskaičiuoti pradinius *Fibonačio* sekos narius:[11]

```
>>> # Fibonačio seka:
... # dviejų narių suma apibrėžia tolesnį narį
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
```

[11]
```
a, b = 0, 1
kol b < 10:
    rodo b
    a, b = b, a+b
```

```
1
1
2
3
5
8
```

Šiame pavyzdyje buvo panaudota keletas naujų dalykų.

- Pirmoje eilutėje naudojamas *priskyrimas keliems kintamiesiems*: kintamieji `a` ir `b` reikšmes (atitinkamai 0 ir 1) įgauna tuo pat metu. Panašus priskyrimas naudojamas ir paskutinėje eilutėje. Čia galima pamatyti, kad visos dešinės priskyrimo pusės reiškiniai apskaičiuojami anksčiau nei atliekamas bet koks priskyrimas. Dešinės pusės apskaičiavimas atliekamas iš kairės į dešinę.
- Ciklo konstrukcija :keyword:`while` vykdoma tol, kol sąlyga yra teisinga (šiuo atveju: `b < 10`). Python'e, panašiai kaip ir C, bet kuris nelygus nuliui skaičius laikomas logine konstanta *teisinga*. Analogiškai, nulis yra laikomas logine konstanta *klaidinga*. Sąlyga taip pat gali būti eilutės arba sąrašo reikšmė. Apibendrinant: bet kokia seka. Tuščia seka yra laikoma *klaidinga*; netuščia (turinti bent vieną elementą) laikoma *teisinga*. Pavyzdyje naudojama sąlyga yra paprastas palyginimas. Standartiniai palyginimo operatoriai užrašomi kaip ir C kalboje: < (mažiau negu), > (daugiau negu), == (lygu), <= (mažiau arba lygu), >= (daugiau arba lygu) bei `!=` (nelygu).
- Vidinis ciklo kodas yra *pastumtas* -- tai Python'o būdas sugrupuoti kodo sakinius. Python'as nesuteikia (kol kas) gudrių eilutės redagavimo galimybių, taigi tarpus arba tabuliacijos ženklus reikia eilutės pradžioje įterpti rankomis. Praktikoje, visgi, dauguma tekstų redaktorių, naudojamų rašyti Python'o kodą, pateikia galimybę automatiškai lygiuoti kodą. Kai sudėtinis kodo sakinys įvedamas interaktyvioje veiksenoje, po jo turi būti įrašyta tuščia eilutė tam, kad Python'as suprastų, jog jūs užrašėte paskutinę eilutę. Taip pat turėkite omenyje, kad visos to paties kodo bloko eilutės privalo būti pastumtos vienodu skaičiumi tarpo ženklų.
- :keyword:`print` komanda atspausdina duoto reiškinio reikšmę. Tai skiriasi nuo paprasčiausio reiškinio užrašymo prie Python'o raginimo tuo, kad :keyword:`print` gali atspausdinti daugiau negu vieną reikšmę. Taip pat ji spausdina eilutes be kabučių ženklų, o pateikus daugiau negu vieną reiškinį, spausdinant tarp jų įterpiami tarpo simboliai:[12]

```
>>> i = 256*256
```

[12]
```
i = 256*256
rodo 'i reikšmė yra ', i
```

```
>>> print 'The value of i is', i
The value of i is 65536
```

Gale parašytas kablelis nurodo atspausdinus reikšmes nepradėti naujos eilutės:[13]

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Tačiau interpretatorius įterps naujos eilutės simbolį prieš pateikdamas raginimą, jei ankstesnė eilutė nebuvo užbaigta.

[13]
```
a, b = 0, 1
kol b < 10:
    rodo b,
    a, b = b, a+b
```

Daugiau srauto valdymo įrankių

Apart tik ką pristatyto reiškinio :keyword:`while`, Python'as pažįsta daugiau srauto valdymo įrankių, žinomų iš kitų kalbų, tačiau yra keletas smulkių skirtumų.

`if` sakinys

Tikriausiai labiausiai žinomas sakinio tipas yra :keyword:`if` sakinys. Pavyzdžiui:[14]

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
More
```

Gali būti nulis arba daugiau :keyword:`elif` dalių, ir :keyword:`else` dalis yra neprivaloma. Bazinis žodis ':keyword:`elif`' yra 'else if' sutrumpinimas ir yra naudingas norint išvengti perkrauto teksto stumdymo. Bazinių žodžių :keyword:`if` ... :keyword:`elif` ... :keyword:`elif` ... seka yra `switch` arba `case` sakinių randamų kitose kalbose atitikmuo.

`for` sakinys

:keyword:`for` sakinys Python'e šiek tiek skiriasi nuo to prie ko jūs galbūt esate pripratę C arba Pascal. Užuot iteravus per aritmetinę skaičių progresiją (kaip Paskalyje) arba leidžiant naudotojui apsibrėžti

[14]
raw_input - įvedimo iš konsolės funkcija.

```
x = int(raw_input("Prašom įvesti sveiką skaičių: "))
jei x < 0:
   x = 0
   rodo 'Neigiamas verčiamas nuliu'
ojei x == 0:
   rodo 'Nulis'
ojei x == 1:
   rodo 'Vienas'
kitaip:
   rodo 'Daug'
```

vienu metu iteracijos žingsnį ir baigimo sąlygą (kaip C), Python'o :keyword:`for` sakinys iteruoja per bet kokios sekos narius (sąrašą arba eilutę), tokia tvarka, kokia jie yra sekoje. Pavyzdžiui:[15]

[15]
```
a = ['katė', 'langas', 'saugumolygis']
kožnam x iš a:
    rodo x, len(x)
```

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Modifikuoti seką, kuri yra iteruojama, nėra saugu (tai gali įvykti tik su kintamais sekų tipais, kaip kad sąrašai). Jeigu jums reikia modifikuoti sąrašą, per kurį jūs iteruojate (pavyzdžiui, norite dvigubinti pasirinktus narius), jūs privalote iteruoti per kopiją. Atkarpos naudojimas tam ypač patogus:[16]

[16]
```
kožnam x iš a[:]:
    jei len(x) > 6: a.insert(0, x)
```

```
>>> for x in a[:]: # padarome viso sąrašo kopiją naudodami atkarpos notaciją
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

`range` funkcija

Jeigu jums reikia iteruoti per skaičių seką, jums pravers įtaisytoji funkcija :func:`range` . Ji sugeneruoja sąrašą, kuriame laikoma aritmetinė progresija:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Duotasis galutinis taškas niekada nėra sąrašo dalis: `range(10)` sugeneruoja 10 reikšmių sąrašą -- legalias indekso reikšmes dešimties narių sekai. Taip pat galima nurodyti kitokią pirmąją reikšmę arba nurodyti kitokį žingsnį (netgi neigiamą):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Norėdami iteruoti per sekos indeksus, jūs galite sujungti :func:`range` ir :func:`len` šitaip:[17]

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Tačiau dažniausiai patogiau naudoti :func:`enumerate` funkciją, žiūrime :ref:`tut-loopidioms`.

`break` ir `continue` sakiniai, bei `else` reiškinys cikluose

:keyword:`break` sakinys, kaip ir C, išeina iš mažiausio uždarančio :keyword:`for` arba :keyword:`while` ciklo.

:keyword:`continue` sakinys, taip pat pasiskolintas iš C, pereina prie kitos ciklo iteracijos.

Ciklai taip pat gali turėti `else` dalį --- ji yra įvykdoma, kai ciklas pereina visus sąrašo elementus (naudojant :keyword:`for`) arba kai sąlyga nebetenkinama (naudojant :keyword:`while`), bet ne tada, kai sąrašas baigiamas naudojant :keyword:`break`. Tai pademonstruota ciklo, kuris ieško pirminių skaičių, pavyzdžiu:[18]

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # ciklas baigėsi neradęs daliklių
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

[17]
```
a = ['Marija', 'turi', 'mažą', 'ėriuką']
kožnam i iš range(len(a)):
    rodo i, a[i]
```

[18]
```
kožnam n iš range(2, 10):
    kožnam x iš range(2, n):
        jei n % x == 0:
            rodo n, 'lygus', x, '*', n/x
            break
        else:
            # ciklas baigėsi neradęs daliklių
            rodo n, 'yra pirminis skaičius'
```

`pass` sakiniai

:keyword:`pass` sakinys nedaro nieko. Jis gali būti naudojamas, kai sakinio reikalauja sintaksė, bet iš programos nereikia jokio veiksmo. Pavyzdžiui:[19]

```
>>> while True:
...     pass  # Laukiame kol naudotojas nutrauks programą naudodamasis klaviatūra (Ctrl+C)
...
```

19
```
kol True:
    pass
```

Tai dažniausiai naudojama norint sukurti minimalią klasę:[20]

```
>>> class MyEmptyClass:
...     pass
...
```

20
```
klasė TuščiaKlasė:
    pass
```

Kita vieta, kur :keyword:`pass` gali būti panaudotas, tai funkcijos arba sąlyginio kodo bloko vieta, kai jūs dirbate prie naujo kodo, kas leidžia mąstyti abstraktesniame lygyje. :keyword:`pass` yra tyliai ignoruojamas:[21]

```
>>> def initlog(*args):
...     pass   # Nepamiršk parašyti funkcijos!
...
```

21
```
apif initlog(*args):
    pass
```

Funkcijų apibrėžimas

Mes galime sukurti funkciją, kuri surašo Fibonači skaičių seką iki tam tikros ribos:[22]

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

22
```
apif fib(n):
    """Spausdina Fibonacci seriją iki
n."""
    a, b = 0, 1
    kol b < n:
        rodo b,
        a, b = b, a+b
fib(2000)
```

:keyword:`def` pradeda funkcijos *apibrėžtį*. Po jo turi sekti funkcijos vardas ir apskliaustas formalių parametrų sąrašas. Sakiniai, kurie sudaro funkcijos kūną, prasideda kitoje eilutėje, ir privalo būti pastumti.

Pirmasis funkcijos sakinys gali būti eilutė --- ši eilutė yra funkcijos dokumentacijos eilutė arba :dfn:`docstring`. (Daugiau apie dokumentacijos eilutes galima rasti :ref:`tut-docstrings` skyriuje.) Egzistuoja

priemonės, kurios, naudodamos dokumentacijos eilutes, gali automatiškai sukurti dokumentaciją arba leidžia naudotojui interaktyviai naršyti po kodą. Dokumentacijos eilučių rašymas yra gera praktika, todėl įpraskite jas rašyti.

Funkcijos *vykdymas* prideda naują simbolių lentelę, kuri naudojama funkcijos lokaliems kintamiesiems. Arba tiksliau, visi kintamųjų priskyrimai funkcijoje prideda reikšmes į lokalią simbolių lentelę. Taigi kintamųjų paieška pirmiausia atliekama lokalioje simbolių lentelėje, tada uždarančios funkcijos lokalioje simbolių lentelėje, po to globalioje simbolių lentelėje ir galiausiai įtaisytoje vardų lentelėje. Taigi globaliems kintamiesiems negalima tiesiogiai priskirti reikšmės funkcijoje (nebent jie būtų paminėti :keyword:`global` sakinyje), nors juos galima skaityti.

Funkcijos apibrėžtis sukuria naują funkcijos vardą dabartinėje simbolių lentelėje. Funkcijos vardo reikšmė turi tipą, kurį atpažįsta interpretatorius. Ši reikšmė gali būti priskirta kitam vardui, kuris vėliau taip pat gali būti naudojamas kaip funkcija. Tai naudojama kaip pervadinimo mechanizmas:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Jeigu jūs atėjote iš kitų kalbų, jūs galite papriekaištauti, kad `fib` yra ne funkcija, o procedūra, nes ji negrąžina reikšmių. Tiesa sakant, netgi funkcijos, kurios nenaudoja :keyword:`return` sakinio grąžina reikšmę, tačiau pakankamai nuobodžią. Ši reikšmė yra `None` (tai yra įtaisytasis vardas). `None` reikšmė paprastai nespausdinama interpretatoriaus, jei tai yra vienintelė reikšmė, kurią reikia atspausdinti. Bet jeigu tikrai norite ją pamatyti, tai galite padaryti naudodami :keyword:`print`:[23]

```
>>> fib(0)
>>> print fib(0)
None
```

Parašyti funkciją, kuri sugrąžina Fibonačių sekos skaičių sąrašą (užuot jį spausdinus) yra labai paprasta:[24]

```
>>> def fib2(n): # grąžina n Fibonacci skaičių seką
...     """Grąžina Fibonacci skaičių seriją iki n."""
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # žiūr. žemiau
```

[23]
```
fib(0)
rodo fib(0)
```

[24]
```
apif fib2(n):
    """Grąžina Fibonacci skaičių seriją
iki n."""
    resultas = []
    a, b = 0, 1
    kol b < n:
       resultas.append(b)
       a, b = b, a+b
    grąžina resultas
f100 = fib2(100)
f100
```

```
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # kviečiama funkcija
>>> f100                # spausdinamas rezultatas
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Šis pavyzdys kaip įprasta taip pat demonstruoja kelias naujas Python'o savybes:

- :keyword:`return` sakinys iš funkcijos grįžta su reikšme. :keyword:`return` be argumentų reiškinio sugrąžina `None`. Jeigu funkcija baigiama nesutikus :keyword:`return` sakinio, tai ji taip pat grąžina `None`.
- Sakinys `result.append(b)` iškviečia sąrašo objekto `result` *metodą*. Metodas yra funkcija, kuri „priklauso“ objektui ir vadinasi `obj.methodname`, kur `obj` yra koks nors objektas (tai gali būti ir reiškinys), ir `methodname` yra metodo, kurį apibrėžia objekto tipas, vardas. Skirtingi tipai apibrėžia skirtingus metodus. Skirtingų tipų metodai gali vadintis taip pat nesukeldami problemų. Norint apibrėžti savo objektų tipus ir metodus reikia naudoti *klases*, kurios aprašytos vėliau šitame vadovėlyje. Metodas :meth:`append` (naudojamas pavyzdyje) yra apibrėžtas sąrašo objektams --- jis prideda naują elementą į sąrašo pabaigą. Tai yra tolygu `result = result + [b]`, bet veikia daug efektyviau.

Daugiau apie funkcijų apibrėžimą

Taip pat galima apibrėžti funkcijas su kintamu argumentų skaičiumi. Galimos trys formos, kurios gali būti kombinuojamos.

Numatytos argumentų reikšmės

Naudingiausia forma yra numatytų reikšmių naudojimas vienam ar daugiau argumentų. Taip sukuriama funkcija, kuri gali būti iškviesta nurodant mažiau argumentų negu ji priima. Pavyzdžiui:[25]

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no', 'nop', 'nope'): return False
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

[25]

```
apif klausk_gerai(įvestis, kiekkartoti=4,
frazė='Taip ar ne, prašau!'):
    kol True:
    gerai = raw_input(įvestis)
    jei gerai iš ('t', 'T', 'taip'):
        grąžina True
    jei gerai iš ('n', 'ne', 'nenene'):
        grąžina False
    kiekkartoti = kiekkartoti - 1
    jei kiekkartoti < 0:
        išveda IOError, 'jums kartoti
nebegalima'
    rodo frazė
```

Šią funkciją galima iškviesti taip: `ask_ok('Do you really want to quit?')`[26] arba štai šitaip: `ask_ok('OK to overwrite the file?', 2)`.[27]

Šis pavyzdys taip pat demonstruoja :keyword:`in` bazinį žodį. Šis raktažodis tikrina ar reikšmė yra sekoje.

Numatytosios reikšmės yra įvertinamos funkcijos apibrėžimo momentu *apibrėžties* srityje, taigi:[28]

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

atspausdins `5`.

Svarbus įspėjimas: numatytoji reikšmė įvertinama tik vieną kartą. Tai yra svarbu kai naudojamas kintamas objektas kaip sąrašas, žodynas ar daugumos klasių egzemplioriai. Pavyzdžiui, žemiau esanti funkcija surenka argumentus, kurie perduodami jai kiekvienu kvietimu:[29]

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Šis kodas atspausdins

```
[1]
[1, 2]
[1, 2, 3]
```

Jeigu jūs nenorite, kad numatytoji reikšmė būtų naudojant kiekvienam kvietimui, jūs galite perrašyti funkciją taip:[30]

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

[26]
```
tekstas = 'Jūs tikrai norite bagti?'
klausk_gerai(tekstas)
```

[27]
```
tekstas = 'Taip noredami keisti failo turinį?'
klausk_gerai(tekstas, 2)
```

[28]
```
i = 5
apif f(arg=i):
    rodo arg
i = 6
f()
```

[29]
```
apif f(a, L=[]):
    L.append(a)
    grąžina L
rodo f(1)
rodo f(2)
rodo f(3)
```

[30]
```
apif f(a, L=None):
    jei L yra None:
        L = []
        L.append(a)
    grąžina L
```

Vardiniai argumentai

Funckijos taip pat gali būti kviečiamos naudojant vardinius argumentus naudojant tokią formą `vardas = reikšmė`. Pavyzdžiui, ši funkcija:[31]

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

[31]
```
apif papūga(įtampa, būsena='a stiff',
veiksmas='voom', tipas='Norwegian
Blue'):
    rodo "-- This parrot wouldn't",
veiksmas,
    rodo "if you put", įtampa, "volts
through it."
    rodo "-- Lovely plumage, the", tipas
    rodo "-- It's", state, "!"
```

gali būti iškviesta tokiai būdais:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

bet sekantys kvietimas yra nevalidus:

```
parrot()                     # trūksta privalomo argumento
parrot(voltage=5.0, 'dead')  # nevardinis argumentas seka vardinį argumentą
parrot(110, voltage=220)     # sudvigubintas argumentas
parrot(actor='John Cleese')  # nežinomas vardinis argumentas
```

Bendrai argumentų sąrašas turi naudoti tik pozicinius argumentus po kurių seka bet kokie vardiniai argumentai, kur vardiniai argumentai turi būti parinkti iš formalių parametrų vardų. Yra visiškai nesvarbu ar formalūs parametrai turi numatytąją reikšmę ar ne. Nė vienas argumentas negali gauti reikšmę daugiau negu vieną kartą --- formalūs parametrų vardai atitinkantys pozicinius argumentus negali būti naudojami kaip vardiniai argumentai tame pačiame kvietime. Čia yra pavyzdys, kuris neveikia dėl šių apribojimų:[32]

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

[32]
```
apif funkcija(a):
    pass
funkcija(0, a=0)
```

Kai paskutinis parametras turi formą `**name`, jis gauna :term:`žodyną <žodynas>`, kuriame yra visi vardiniai argumentai išskyrus tuos, kurie yra aprašyti kaip formalūs parametrai. Ši forma gali būti naudojama su `*name` parametrų forma (aprašyta kitam poskyryje) kuri gauna

kortežą, kuriame sudėti poziciniai argumentai neapibrėžti formaliame parametrų sąraše. `*name` turi būti apibrėžtas prieš `**name`. Pavyzdžiui, jei mes apibrėšime funkciją taip:[33]

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print "-" * 40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ":", keywords[kw]
```

[33]
```
apif sūrioparduotuvė(rūšis, *argumentai, **raktažodžiai):
    rodo "-- Ar jūs turite nors truputį", rūšis, "?"
    rodo "-- Atsiprašau, bet mes nebeturime", kind
    kožnam arg iš argumentai:
        rodo arg
    rodo "-" * 40
    raktai = raktažodžiai.keys()
    raktai.sort()
    kožnam kw iš raktai:
        rodo kw, ":", raktažodžiai[kw]
```

Ji gali būti iškviečiama taip:[34]

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

[34]
```
sūrioparduotuvė("Limburgerio", "Labai apgailestaujame, ponas.",
 "Labai, labai apgailestaujame, ponas.",
 sūrininkas='Michaelis Palinas',
 klientas="Jonas Klišė",
 scena="Sūrio Parduotuvės Scena")
```

ir žinoma ji atspausdins:[35]

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

[35]
```
-- Ar jūs turite nors truputį Limburgerio?
-- Atsiprašau, bet mes nebeturime Limburgerio
Labai apgailestaujame, ponas.
Labai, labai apgailestaujame, ponas.
----------------------------------------
klientas : Jonas Klišė
sūrininkas : Michaelis Palinas
scena : Sūrio Parduotuvės Schema
```

Pastebėsime, kad argumentų vardų sąrašo metodas :meth:`sort` yra iškviečiamas prieš spausdinant žodyno `keywords` reikšmes. Jei to nepadarytūmėm, tai tvarka, kuria būti atspausdinami argumentai, būtų neapibrėžta.

Laisvas argumentų sąrašas

Galiausiai, rečiausiai naudojama galimybė yra nurodyti, kad funkcija gali būti iškviesta su bet kokiu argumentų skaičiumi. Šie argumentai bus sudėti į kortežą (žr. :ref:`tut-tuples`). Prieš kintamą argumentų skaičių, galima nurodyti nulį ar daugiau normalių argumentų:[36]

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

[36]
```
apif spausdinti_visus_kartu(failas, skirtukas, *args):
    failas.write(skirtukas.join(args))
```

Argumentų sąrašo išpakavimas

Atvirkštinė situacija įvyksta, kai argumentai jau yra sąraše arba kortеže, bet turi būti išpakuoti funkcijos kvietimui, kuri reikalauja pozicinių argumentų. Pavyzdžiui įtaisytoji funkcija :func:`range` tikisi atskirų *pradžios* ir *pabaigos* argumentų. Jeigu jų neturime atskirai, funkcijos kvietime panaudokite ∗-operatorių tam, kad argumentai būtų išpakuoti iš argumentų sąrašo:

```
>>> range(3, 6)             # normalus kvietimas su atskirais argumentais
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            # kvietimas su argumentais išpakuotais iš sąrašo
[3, 4, 5]
```

Tokiu pat būdu, žodynai gali ištraukti vardinius argumentus naudodami ∗∗-operatorių:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

[37]

[37]
```
apif papūga(įtampa, būsena='a stiff',
veiksmas='voom'):
    rodo "-- This parrot wouldn't",
veiksmas,
    rodo "if you put", įtampa, "volts
through it.",
    rodo "E's", būsena, "!"
d = "įtampa": "four million", "būse-
na": "bleedin' demised", "veiksmas":
"VOOM"
parrot(**d)
```

Lambda formos

Pagal populiarų prašymą naują savybė randama kitose funkcinėse programavimo kalbose kaip kad Lisp buvo pridėta ir į Python'ą. Naudojant :keyword:`lambda` bazinį žodį, galima kurti mažas anonimines funkcijas. Čia yra funkcija, kuri sugrąžina dviejų argumentų sumą: `lambda a, b: a+b`. Lambda formos gali būti naudojamas kiekvieną kartą kai reikia funkcijos objekto. Jos sintaksiškai yra apribotos iki vieno reiškinio. Semantiškai, jos tėra sintaksinis cukrus normalių funkcijų apibrėžtims. Kaip yra įdėtinės funkcijos, lambda formos gali pasiekti kintamuosius iš išorinės srities:[38]

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
```

[38]
```
apif kuria_prieaugį(n):
    return lambda x: x + n
f = apif kuria_prieaugį(42)
```

```
42
>>> f(1)
43
```

Dokumentacijos eilutės

Egzistuoja susitarimai dokumentacijos eilučių turiniui ir formatavimui.

Pirmoji eilutė visada turi būti trumpas, aiškus objekto paskirties apibendrinimas. Dėl trumpumo objekto vardas ar tipas neturi būti minimas, nes jis ir taip yra matomas (išskyrus, jei vardas yra veiksmažodis apibūdinantis funkcijos operaciją). Ši eilutė turi prasidėti iš didžiosios raidės ir pasibaigti tašku.

Jeigu dokumentacijos eilutėje yra daugiau eilučių, antroji eilutė turi būti tuščia, atskirianti apibendrinimą nuo aprašymo. Sekančios eilutės turi būti vienas ar daugiau paragrafų aprašančių objekto kvietimo būdus, šalutinius efektus ir t.t.

Python'o interpetatorius nepašalina pastūmimų iš daugiaeilutės eilutės, taigi priemonės, kurios tvarko dokumentaciją turi padaryti tai pačios, jei to nori. Tai atliekama pagal tokį susitarimą. Pirma netuščia eilutė po pirmos eilutės nustato per kiek yra pastumtas tekstas visai dokumentacijai. Mes negalime naudoti pirmosios eilutės, nes ji paprastai būną toje pačioje eilutėje kaip atidarančios kabutės, todėl jos lygiavimas nėra akivaizdus. Tada šis tarpų „ekvivalentas“ nukerpamas nuo visų eilučių starto. Tekste neturėtų būti eilučių, kurios pastumtos mažiau, bet jei taip atsitinka visi tarpai turi būti nukerpami.

Čia yra daugiaeilutės dokumentacijos eilutės pavyzdys:[39] [40]

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

[39]
```
apif mano_funkcija():
    """Nedaro nieko, bet dokumentuoja
tai.
    Visiškai nieko, tai yra ničnieko."""
    pass
```

[40] rodo mano_funkcija.__doc__
Nedaro nieko, bet dokumentuoja tai.
Visiškai nieko, tai yra ničnieko

Intermezzo: programavimo stilius

Dabar kai jūs esate pasiruošę rašyti ilgesnes ir sudėtingesnes Python'o programas, tai yra puikus metas pakalbėti apie *programavimo stilių*. Dauguma kalbų gali būti rašomas (arba tiksliau sakant *forma-*

tuojamas) skirtingais stiliais --- kai kurios yra skaitomesnės negu kitos. Rašyti kitiems skaitomą kodą yra gera idėja, todėl gražaus programavimo stiliaus naudojimas labai padeda.

Python'e dauguma projektų naudoja stilių apibrėžtą :pep:`8` dokumente --- kas skatina labai skaitomą ir akiai malonų programavimo stilių. Kiekvienas Python'o programuotojas privalo jį kada nors perskaityti. Štai čia aprašomi patys svarbiausi punktai:

- Naudokite 4-tarpų pastūmimą, ir nenaudokite tabuliacijos.

  4 tarpai yra kompromisas tarp mažo pastūmimo (leidžia naudoti gilesnį įdėjimo gylį) ir didelio pastūmimo (paprasčiau skaityti). Tabuliaciją naudoti teisingai yra sudėtinga ir todėl geriau jos nenaudoti iš viso.

- Nerašykite eilučių, kurios yra ilgesnės negu 79 simboliai.

  Tai padeda naudotojams, kurie naudoja mažus ekranus ir leidžia stebėti du kodo puslapius turint didelį ekraną.

- Naudokite tuščias eilutes atskirdami funkcijas ir klases, bei didesnius kodo blokus funkcijos viduje.

- Kai įmanoma komentarus rašykite atskiroje eilutėje.

- Naudokite dokumentacijos eilutes.

- Naudokite tarpus aplink operatorius ir po kablelių, bet ne skliaustelių konstrukcijose: `a = f(1, 2) + g(3, 4)`.

- Vardus klasėms ir funkcijoms duokite nuosekliai. Oficialiai susitarta, kad klasėms vardai turi būti duodami naudojant `KupranugarioLygius`, o funkcijoms ir metodams `mažosios_raidės_su_pabraukimais`. Visada naudokite `self` vardą pirmam metodo argumentui (žr. :ref:`tut-firstclasses` apie klases ir metodus).

- Nenaudokite keistų koduočių savo kode, jeigu jūsų kodas yra skirtas tarptautinei aplinkai. Paprasta ASCII koduose yra geriausia dauguma atvejų.

Duomenų struktūros

Šis skyrius pateiks daugiau informacijos apie duomenų struktūras, su kuriomis buvo glaustai susipažinta ankstesniuose skyreliuose. Taip pat čia rasite ir naujų dalykų.

Daugiau apie sąrašus

Sąrašo duomenų tipas pateikia naudingų metodų. Štai jie:

Pavyzdys, kuriame panaudojam dauguma sąrašo metodų:[41]

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

[41]
```
a = [66.25, 333, 333, 1, 1234.5]
# count() - suskaičiuoja kiek narių
rodo a.count(333), a.count(66.25),
a.count('x')
# insert() įterpia po
a.insert(2, -1)
# append() - prideda prie sąrašo
pabaigos a.append(333)
# index() - randa indeksą
a.index(333)
# remove() - ištrina narį
a.remove(333)
# reverse() - apverčia sąrašą
a.reverse()
# sort() - rūšiuoja sąrašą
a.sort()
```

Sąrašo naudojimas dėklams

Sąrašo metodai leidžia sąrašą naudoti kaip dėklą (angl. stack), kur paskutinis pridėtas elementas yra pirmiausia išimamas („paskutinis įdetas, pirmas išimtas“). Norėdami pridėti elementą į deklą, naudokite

:meth:`append`. Norėdami išimti elementą iš dėklo viršaus, naudokite :meth:`pop` nenurodydami indekso. Pavyzdžiui:[42]

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

[42]
```
vienspokito = [3, 4, 5]
# append() - prideda į pabaigą
vienspokito.append(6)
vienspokito.append(7)
# pop() - išima galutinai
vienspokito.pop()
```

Sąrašo naudojimas eilėms

Jūs taip pat patogiai galite panaudoti sąrašą kaip eilę, kur pirmas pridėtas elementas yra pirmas išimamas („pirmas-į, pirmas-iš“). Elemento pridėjimui naudokite :meth:`append`, o elemento išėmimui iš eilės priekio naudokite :meth:`pop` su indeksu `0`. Pavyzdžiui:[43]

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Terry atvyksta
>>> queue.append("Graham")          # Graham'as atvyksta
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

[43]
```
eilė = ["Eric", "John", "Michael"]
# Terry atvyksta
eilė.append("Terry")
# Graham'as atvyksta
eilė.append("Graham")
# pop(0) - išima nulinį galutinai
eilė.pop(0)
'Eric'
# pop(0) - išima nulinį galutinai
eilė.pop(0)
'John'
```

Funkcinio programavimo įrankiai

Egzistuoja trys įtaisytosios funkcijos, kurios yra labai naudingos su sąrašais: :func:`filter`, :func:`map`, ir :func:`reduce`.

`filter(function, sequence)` sugrąžina seką sudarytą iš elementų, kuriems tenkina funkcijos `function(item)` sąlygą. Jei *sequence* yra :class:`string` arba :class:`tuple` tipo, rezultatas bus to paties tipo, bet kokiu kitu atveju rezultatas visada yra :class:`list`. Pavyzdžiui, kelis pirminius skaičius galime suskaičiuoti taip:[44]

[44]
```
apif f(x):
   grąžina x % 2 != 0 ir x % 3 != 0
# skaičiuoja f x'ams nuo 2 iki 24
filter(f, range(2, 25))
```

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

`map(function, sequence)` iškviečia `function(item)` kiekvienam sekos elementui ir sugrąžina sugrąžintų reikšmių sąrašą. Pavyzdžiui, jei norime suskaičiuoti kelis kubus:[45]

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Galima perduoti daugiau negu vieną seką --- funkcija tada privalo turėti tiek argumentų kiek yra perduodama sekų. Tada funkcijai perduodami argumentai iš kiekvienos sekos, arba `None`, jei kuri nors seka trumpesnė. Pavyzdžiui:[46]

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` sugrąžina vieną reikšmę, kuri sukonstruojama naudojant dvejetainę funkciją *function* pirmiems dviems elementams, tada rezultatui ir sekančiam elementui ir t.t. Pavyzdžiui, jei norime suskaičiuoti sumą nuo 1 iki 10:[47]

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Jei sąraše yra tik vienas elementas, jo reikšmė yra sugrąžinama. Jei seka tuščia sukeliama išimtis.

Galima perduoti trečia arugmentą nurodantį pradinę reikšmę. Tokiu atveju pradinė reikšmė sugrąžinama tuščiai sekai, o funkcija iš pradžių pritaikoma pradinei reikšmei ir pirmas sekos elementui, tada rezultatai ir kitam elementui ir t.t. Pavyzdžiui:[48]

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
```

[45]
```
def kubu(x):
    return x*x*x
# map() - kiekvienam iteruojamam kuria sąrašą
map(kubu, range(1, 11))
```

[46]
```
sąr = range(8)
apif suma(x, y):
    grąžina x+y
map(suma, sąr, sąr)
```

[47]
```
apif suma(x,y):
    grąžina x+y
reduce(suma, range(1, 11))
```

[48]
```
apif suma(sąr):
    apif plius(x,y):
        grąžina x+y
    grąžina reduce(suma, sąr, 0)
suma(range(1, 11))
55
suma([])
0
```

```
55
>>> sum([])
0
```

Nenaudokite šio pavyzdžio apibrėžties :func:`sum`: kadangi skaičių sumavimas yra tokia dažna užduotis, kad įtaisytoji funkcija `sum(sequence)` jau egzistuoja ir ji būtent taip ir dirba.

Sąrašo užklausos

Sąrašo užklausos leidžia paprastai sukurti sąrašus nenaudojant funkcijų :func:`map`, :func:`filter` ir/ar :keyword:`lambda` funkcijų. Gauta sąrašo apibrėžtis dažniausiai linkusi būti aiškesnė negu naudojant paminėtas konstrukcijas. Kiekviena sąrašo užklausa sudaroma iš reiškinio po kurio seka :keyword:`for` sakinys, tada gali sekti nulis arba daugiau :keyword:`for` arba :keyword:`if` sakinių. Taip gausime sąrašą, kuris sudaromas įvertinant reiškinį :keyword:`for` ir :keyword:`if` sakinių kontekste, kurie seka po reiškinio. Jeigu reiškinio rezultatas yra kortežas, tai rezultatas privalo būti apskliaustas:

```
vaisiai = ['  bananai', '  gervuogės ', 'aistravaisiai  ']
[v.strip() for v in vaisiai]
['bananai', 'gervuogės', 'aistravaisiai']
vek = [2, 4, 6]
[3*x kožnam x iš vek]
[6, 12, 18]
[3*x kožnam x iš vek jei x > 3]
[12, 18]
[3*x kožnam x iš vek jei x < 2]
[]
[[x,x**2] kožnam x iš vek]
[[2, 4], [4, 16], [6, 36]]
[x, x**2 kožnam x iš vek]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vek]
               ^
SyntaxError: invalid syntax
[(x, x**2) kožnam x iš vek]
[(2, 4), (4, 16), (6, 36)]
vek1 = [2, 4, 6]
vek2 = [4, 3, -9]
[x*y kožnam x iš vek1 kožnam y iš vek2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
[x+y kožnam x iš vek1 kožnam y iš vek2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

```
[vek1[i]*vek2[i] kožnam i iš range(len(vec1))]
[8, 12, -54]

>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]  # klaida -- kortežui reikalingi skliausteliai
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Sąrašo užklausos yra daug lankstesnės negu :func:`map` ir gali būti pritaikytos sudėtingies reiškiniams ir vidinėms funkcijoms:[49]

[49] round(a,b) - apvalina a iki b ženklų
str - verčia simbolių eilute

```
>>> [str(round(355/113.0, i)) kožnam i iš range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']

>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Įdėtinės sąrašo užklausos

Jeigu jūs mėgstate sudėtingus dalykus, sąrašo užklausos gali būti įdėtinės. Tai yra galingas įrankis, bet kaip visi galingi įrankiai, jos turi būti naudojamos atsargiai (arba išvis nenaudojamos).

Tarkime turime 3x3 matricą, kuri yra sudaryta iš sąrašo, kuriame laikomi trys sąrašai:

```
>>> mat = [
...        [1, 2, 3],
...        [4, 5, 6],
...        [7, 8, 9],
...       ]
```

Taigi, jeigu norėtumėte sukeisti eilutes ir stulpelius, jūs galite panaudoti sąrašo užklausas:

```
>>> rodo [[eilutė[i] kožnam eilutė iš mat] for i in [0, 1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]

>>> print [[row[i] for row in mat] for i in [0, 1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Įdėtinės sąrašo užklausos turi būti įvertintos ypatingai:

> Kad išvengtumėte abejonių, kai sąrašo užklausos yra įdedamos viena į kitą, skaitykite iš dešinės į kairę.

Skaitomesnė šio kodo versija parodo veikimą aiškiau:[50]

[50]
kožnam i iš [0, 1, 2]:
  kožnam eilutė iš mat:
    rodo eilutė[i],
  rodo

```
for i in [0, 1, 2]:
    for row in mat:
        print row[i],
    print
```

Realiame pasaulyje, jūs turėtumėte naudoti įtaisytąsias funkcijas vietoj sudėtingų sakinių. Funkciją :func:`zip` šiuo atveju atliks būtent tai, ko jums reikia:

```
>>> zip(*mat)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Žr. :ref:`tut-unpacking-arguments`, ką reiškia žvaigždutė šioje eilutėje.

`del` sakinys

Norėdami išimti iš sąrašo elementą pagal indeksą, o ne pagal reikšmę, naudokite :keyword:`del`[51] sakinį. Jis skiriasi nuo :meth:`pop` metodo, kuris grąžina reikšmę. :keyword:`del` sakinys gali būti naudojamas išimti iš sąrašo atkarpas arba norint išvalyti visą sąrašą (seniau mes tai atlikome priskirdami tuščią sąrašą atkarpai). Pavyzdžiui:

[51] del - delete (trinti) trumpinys

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
```

```
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

:keyword:`del` gali būti naudojamas kintamųjų pašalinimui:

```
>>> del a
```

Po šios operacijos bandymas pasiekti `a` yra klaida (nebent kita reikšmė yra priskiriama šiam kintamajam). Daugiau :keyword:`del` panaudojimo būdų sutiksime vėliau.

Kortežai ir sekos

Mes pastebėjome, kad sąrašai ir eilutės turi daug bendrų savybių, kaip kad indeksavimas ir kirpimo operacijos. Jie yra *sekos* duomenų tipų pavyzdžiai. Kadangi Python'as yra besivystanti kalba, kiti sekos tipai gali būti pridėti ateityje. Python'e yra ir kitas standartinis sekos tipas: *kortežas* (angl. tuple).

Kortežas yra kelių reikšmių seka atskirta kableliais. Pavyzdžiui:

```
>>> t = 12345, 54321, 'labas!'
>>> t[0]
12345
>>> t
(12345, 54321, 'labas!')
>>> # Kortežai gali būti įdėti vienas į kitą:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'labas!'), (1, 2, 3, 4, 5))
```

Kaip jūs matote, kortežų rezultatas visada yra apskliaustas, tam kad idėtieji kortežai būtų interpretuojami teisingai. Jų įvedimas galimas tiek be tiek su skliausteliais, nors dažniausiai skliausteliai yra reikalingi (ypač jei kortežas yra didesnio reiškinio dalis).

Kortežai turi daug panaudojimo būdų. Pavyzdžiui: (x, y) koordinačių pora, darbuotojo įrašas duombazėje ir t.t. Kortežai, kaip eilutės, yra nekintami, t.y. neįmanoma priskirti reikšmės kuriam nors kortežo nariui (tačiau tai galite padaryti naudodami kirpimus ir sujungimus). Taip pat galima sukurti kortežą, kurio nariai yra kintami objektai (kaip kad sąrašai).

Išskirtinė problema iškyla norint sukurti kortežus, kurie sudaryti iš 0 ar 1 nario. Norint tai padaryti reikia žinoti tam tikrus sintaksės niuansus. Tuščias kortežas sukonstruojamas naudojant tuščius skliaustelius. Kortežas sudarytas iš vieno nario yra sukonstruojamas padedant kablelį po reikšmės (apskliausti vieną reikšmę neužteks). Negražu, bet efektyvu. Pavyzdžiui:[52]

```
>>> empty = ()
>>> singleton = 'hello',    # <-- atkreipkite dėmesį į kablelį
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

[52]
```
tuščia = ()
frazė = 'sveiki',
len(tuščia)# len - ilgio trumpinys
0
len(frazė)
1
frazė
('sveiki',)
```

Sakinys `t = 12345, 54321, 'hello!'` demonstruoja *kortežo pakavimą*: `12345`, `54321` ir `'hello!'` yra kartu supakuojamas į kortežą. Atvirkštinė operacija taip pat yra galima:

```
>>> x, y, z = t
```

Tai vadinama *sekos išpakavimu*. Sekos išpakavimas reikalauja tiek kintamųjų kiek reikšmių yra sekoje. Atkreipsime dėmesį, kad priskyrimas keliams kintamiesiems yra tik kortežo pakavimo ir sekos išpakavimo kombinacija!

Čia tėra tik šiek tiek asimetrijos: kelių reikšmių pakavimas visada sukuria kortežą, o išpakavimas veikia su bet kokia seka.

Aibės

Python'e taip yra duomenų tipas *aibėms*. Aibė yra nesurikiuotų elementų rinkinys, kuriame nėra pasikartojančių elementų. Įprastai aibės naudojamos buvimo aibėje tikrinimui ir dvigubų narių pašalinimui. Aibės objektai taip pat palaiko matematines operacijas kaip sąjunga, sankirta, skirtumas ar simetriškas skirtumas.

Čia demonstruojamos aibių galimybės:

```
>>> krepšys = ['obuolys', 'apelsinas', 'obuolys', 'kriaušė', 'apelsinas', 'bananas']
>>> vaisiai = set(krepšys)              # sukuriame aibę be dublikatų
>>> vaisiai
set(['apelsinas', 'kriaušė', 'obuolys', 'bananas'])
>>> 'apelsinas' iš vaisiai                # greitas priklausymo aibei tikrinimas
True
>>> 'krabgrasas' iš vaisiai
False
```

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # sukuriame aibę be dublikatų
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # greitas priklausymo aibei tikrinimas
True
>>> 'crabgrass' in fruit
False

>>> # Demonstruojame aibių operacijas unikalioms raidėms iš dviejų žodžių
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unikalios raidės žodyje a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # raidės, kurios yra a, bet ne b
set(['r', 'd', 'b'])
>>> a | b                              # raidės, kurios yra arba a arba b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                              # raidės, kurios yra ir a ir b
set(['a', 'c'])
>>> a ^ b                              # raidės, kurios yra a arba b, bet ne abiejuose
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Žodynai

Kitas naudingas duomenų tipas Python'e yra *žodynas*. Kitose kalbose žodynai vadinami „asociatyviąją atmintimi“ arba „asociatyviaisiais masyvais“. Skirtingai nuo sekų, kurios yra indeksuojamos skaičiais, žodynai indeksuojami naudojant *raktus*, kuriais gali būti bet kuris nekintamas tipas. Eilutės arba skaičiai visada gali būti naudojami kaip raktai. Kortežai gali būti naudojami kaip raktai, jeigu jie sudaryti tik iš eilučių, skaičių ar kortežų. Jeigu kortežas yra sudarytas iš kintamų objektų (tiesiogiai ar netiesiogiai), jo negalima naudoti raktui. Jūs negalite naudoti sąrašų raktams, kadangi sąrašai gali būti modifikuojami vietoje naudojant indekso priskyrimą, atkarpų priskyrimus arba metodus kaip :meth:`append` ir :meth:`extend`.

Žodynus geriausia įsivaizduoti kaip nesutvarkytą *raktas:reikšmė* porų aibę, kur reikalaujama, kad raktas būtų žodyne unikalus. Tuščia figūrinių skliaustelių pora `{}` sukuria tuščia žodyną. Norėdami sukurti žodyną su pradinėmis reikšmėmis tarp figūrinių skliaustelių surašysite raktas:reikšmė poras atskirtas kableliais. Tokiu pat būdu žodynai yra išvedami.

Pagrindinė žodynų operacija yra reikšmių įdėjimas pagal raktą ir tos

reikšmės ištraukimas pagal raktą. Taip pat galima iš žodyno pašalinti raktas:reikšmė porą naudojant `del`. Jeigu jūs į žodyną rašote reikšmę naudodami raktą, kuris jau žodyne panaudotas, senoji reikšmė yra pamirštama. Jeigu bandysite ištraukti reikšmę iš žodyno naudodami neesantį raktą gausite klaidą.

Metodas :meth:`keys` sugrąžina žodyne naudojamų raktų sąrašą. Sąrašo tvarka nėra apibrėžta, todėl, jei norite surikiuoto sąrašo, tiesiog panaudokite metodą :meth:`sort` raktų sąrašui. Jeigu norite patikrinti, ar raktas yra žodyne, naudokite bazinį žodį :keyword:`in`.

Mažas pavyzdys kaip naudoti žodyną:

```
>>> tel = {'džakas': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'džakas': 4098}
>>> tel['džakas']
4098
>>> del tel['sape']
>>> tel['irvinas'] = 4127
>>> tel
{'guido': 4127, 'irvinas': 4127, 'džakas': 4098}
>>> tel.keys()
['guido', 'irvinas', 'džakas']
>>> 'guido' iš tel
True

>>> 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']
>>> 'guido' in tel
True
```

Funkcija :func:`dict` sukonstruoja žodyną tiesiogiai iš kortežų sąrašo, kur korteže laikomos rakto ir reikšmės poros. Jeigu poros gali būti sudarytos algoritmiškai, žodyno sudarymui galite panaudoti sąrašo užklausą:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # sąrašo užklausos panaudojimas
{2: 4, 4: 16, 6: 36}
```

Vėliau šiame vadovėlyje išmoksime apie Generatoriaus Reiškinius, kurie dar labiau tinka raktas-reikšmė porų sudarymui :func:`dict` funkcijai.

Kai raktai yra paprastos eilutės, tada kartais paprasčiau nurodyti poras naudojant vardinius argumentus:

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

Iteravimo technikos

Norint pereiti per žodyną, raktas ir jį atitinkantį reikšmė gali būti gauta vienu metu naudojant :meth:`iteritems` metodą.[53]

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

Kai einama per seką, pozicijos indeksas ir atitinkanti reikšmė gali būti gauta vienu metu naudojant :func:`enumerate` funkciją.

```
>>> kožnam i, v iš enumerate(['tik', 'tak', 'pabaiga']):
...     rodo i, v
...
0 tik
1 tak
2 pabaiga

>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Norėdami pereiti per dvi ar daugiau sekų vienu metu, galite sujungti įrašus naudodami :func:`zip` funkciją.

[53]
```
riteriai = {'galahadas': 'silpnasis',
'robinas': 'šaunusis'}
kožnam k, v iš riteriai.iteritems():
    print k, v
```

```
>>> klausimai = ['vardas', 'ieškojimas', 'mėgstama spalva']
>>> atsakymai = ['lancelotas', 'šventasis grailis', 'mėlynas']
>>> kožnam q, a iš zip(klausimai, atsakymai):
...     rodo 'Koks jūsų {0}?  Mano yra {1}.'.format(q, a)
...
```

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Norėdami pereiti seką iš kito galo, pirma nurodykite seką normalia tvarka ir tada iškvieskite :func:`reversed`[54] funkciją.[55]

[54] reversed - atabulai

[55]
```
kožnam i iš reversed(xrange(1,10,2)):
    rodo i
```

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

Jei norite pereiti per surikiuotą seką, naudokite :func:`sorted`[56] funkciją, kuri sugrąžina naują surikiuotą sąrašą (bet originalų sąrašą palieka nepakeistą).[57]

[56] sorted - surūšiuota

[57]
```
krepšys = ['obuolys', 'apelsinas',
'obuolys', 'kriaušė', 'apelsinas',
'bananas']
kožnam f iš sorted(set(basket)):
    rodo f
```

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

Daugiau apie sąlygas

Sąlygose naudojamose `while` ir `if` sakiniuose gali būti naudojami operatoriai (ne tik palyginimai).

Palyginimo operatoriai `in` ir `not in` tikrina, ar reikšmė yra sekoje. Operatoriai `is` ir `is not` palygina ar du objektai yra vienas ir tas pats

objektas: tai svarbu tik kintamiems tipams kaip kad sąrašai. Visi palyginimo operatoriai yra to paties prioriteto, bet mažesnio prioriteto negu skaičių operatoriai.

Palyginimai gali būti sujungiami. Pavyzdžiui, `a < b == c` patikrina ar `a` yra mažiau negu `b` ir dar ar `b` lygu `c`.

Palyginimai gali būti sujungti naudojant loginius operatorius `and` ir `or`, ir bet kurio palyginimo (ar bet kokio loginio reiškinio) rezultatas gali būti paneigtas naudojant `not`. Šie operatoriai turi mažesnį prioritetą negu palyginimo operatoriai --- iš jų `not` turi aukščiausia prioritetą ir `or` mažiausią, taigi `A and not B or C` yra tas pats kas `(A and (not B)) or C`. Kaip visada, skliausteliai gali padėti išreikšti norimą kompoziciją.

Loginiai operatoriai `and` ir `or` yra taip pavadinami *trumpos grandinės* operatoriais: jų argumentai yra įvertinami iš kairės į dešinę, ir įvertinimas sustabdomas kai tik rezultatas yra nustatomas. Pavyzdžiui, jei `A` ir `C` yra tiesa, bet `B` yra netiesa, tai `A and B and C` nevertina `C` reikšmės. Kai naudojamos bendros, o ne loginės, reikšmės, trumpos-grandinės operatorių rezultatas yra paskutinis įvertintas argumentas.

Palyginimo (ar kitos loginio reiškinio) rezultatą galima priskirti kintamajam. Pavyzdžiui:

```
>>> eilutė1, eilutė2, eilutė3 = '', 'Trondheim', 'Hamerio šokis'
>>> non_null = eilutė1 or eilutė2 or eilutė3
>>> non_null
'Trondheim'

>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Pastebėkite, kad Python'e (skirtingai nuo C), priskyrimas negali įvykti reiškinyje. C programuotojai gali būti tuo nepatenkinti, bet tai padeda išvengti dažnos C problemos, kai panaudojamas = reiškinyje, kur norėta parašyti ==.

Sekų ir kitų tipų palyginimas

Sekų objektai gali būti palyginami su kitais objektais, kurie turi tą patį sekos tipą. Palyginimas naudoja *leksikografinę* tvarką: pirmiausia palyginami pirmi du nariai, ir jeigu jie skiriasi pagal tai nustatomas rezultatas. Jei jie lygūs, tada lyginami kitu du nariai, ir taip toliau, kol kur nors seka baigiasi. Jeigu patys nariai yra sekos, tada rekursiškai leksikografine tvarka lyginami jie patys. Jei visi sekos nariai vienodi, tada sekos laikomos lygiomis. Jeigu viena seka yra kitos sekos pradžia, tai

trumpesnioji seka yra mažesnė. Leksikografinė tvarka eilutėms naudoja ASCII koduotę atskiroms raidėms. Keletas palyginimo pavyzdžių tarp to paties tipo sekų:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Atkreipkite dėmesį į tai, kad skirtingų tipų palyginimas yra legalus. Rezultatas gali būti nustatytas tiksliai (bet viskas gali būti painu): tipai yra rikiuojami pagal vardus. Taigi sąrašas (**l**ist) yra visada trumpesnis už eilutę (**s**tring), eilutė visada trumpesnė už kortežą (**t**uple) ir t.t. [#]_ Skirtingo tipo skaičiai lyginami pagal jų reikšmę, taigi 0 lygu 0.0 ir t.t.

Moduliai

Jei išjungsite Python interpretatorių ir vėliau vėl jį įjungsite, visi jūsų aprašai (funkcijos ir kintamieji), aprašyti interpretatoriuje, dings. Todėl, jei ketinate rašyti sudėtingesnę programą, geriau naudotis teksto redaktoriumi kintamųjų ir funkcijų aprašymui, kuriuos vėliau galėsite panaudoti interpretatoriuje. Failas, kuriame saugomas jūsų paruoštas kodas, vadinamas *skriptu*. Programai didėjant, tikriausiai kils poreikis išskaidyti ją keliuose failuose, kad būtų lengviau surasti, kas kur aprašyta. Tikriausiai norėsite aprašytą funkciją panaudoti keliose kitose programose nekopijuojant viso funkcijos aprašo į kiekvieną iš tų programų.

Python'as gali nuskaityti jūsų kodo aprašus iš failo ir suteikia galimybę panaudoti kiekvieną aprašytą struktūrą panaudoti kitame skripte ar interpretatoriuje. Toks failas vadinamas *moduliu*: apibrėžtos struktūros iš vieno modulio, gali būti *importuotos* į kitus modulius arba į jūsų pagrindinį modulį (vykdomąjį skriptą arba interaktyvaus interpretatoriaus režimą, kur galite operuoti kintamaisiais).

Modulis yra paprastas failas, kuriame saugomas Python'o kodas. Failo vardas sudaromas iš modulio pavadinimo ir galūnės: :file:`.py`. Modulyje pačio modulio vardas (kaip simbolių eilutė) saugomas globaliame kintamajame `__name__`. Praktiniam išbandymui, atverkite savo mėgiamą teksto redaktorių ir einamajame kataloge sukurkite naują failą pavadinimu :file:`fibo.py`. Failo turinys turėtu būti toks:[58]

```
# Fibonačio skaičių modulis

def fib(n):    # išvedama Fibonačio seka iki n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # grąžinama Fibonačio seka iki n
    result = []
    a, b = 0, 1
```

[58]

```
apif fib(n):
   a, b = 0, 1
   kol b < n:
      rodo b,
      a, b = b, a+b
apif fib2(n):
   resultas = [ ]
   a, b = 0, 1
   kol b < n:
      # append - papildyti
      resultas.append(b)
      a, b = b, a+b
   return resultas
```

```
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Dabar paleiskite Python'o interpretatorių ir importuokite šį moduli naudodami tokią komandą:[59]

[59] naudoja fibo

```
>>> import fibo
```

Ši komanda funkcijų vardų, kurie yra apibrėžti `fibo` modulyje, nepridėda į dabartinę simbolių lentelę --- į lentelę pridedamas tik modulio vardas `fibo`. Naudodami modulio vardą jūs galite pasiekti funkcijas:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Jeigu jūs planuojate naudoti funkciją dažnai jūs galite ją priskirti lokaliam vardui:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Daugiau apie Modulius

Modulyje gali būti sudėti tiek vykdomi sakiniai tiek funkcijų apibrėžtys. Šių sakinių tikslas yra modulio inicializavimas. Jie vykdomi tik *pirmą* kartą, kai modulis yra importuojamas. [#]_

Kiekvienas modulis turi savo privačią simbolių lentelę, kuri naudojama kaip globali simbolių lentelė visoms funkcijoms modulyje. Taigi modulio autorius gali naudoti globalius kintamuosius nesirūpindamas apie atsitiktinius vardų sutapimus su naudotojo globaliais kintamaisiais. Iš kitos pusės, jeigu jūs žinote, ką jus darote, jūs galite pasiekti modulio globalius kintamuosius naudodamiesi ta pačia sintakse, kurią naudojote funkcijoms --- `modname.itemname`.

Moduliai gali importuoti modulius. Yra įprasta (nors nebūtina) visus :keyword:`import` (`naudoja`) sakinius surašyti modulio (ar skripto) pradžioje. Visi importuotų modulių vardai yra įkeliami į importuojančio modulio globalią simbolių lentelę.

:keyword:`import` (`naudoja`) sakinys turi formą leidžiančią importuoti pasirinktus vardus iš modulio tiesiai į importuojančio modulio simbolių lentelę. Pavyzdžiui:[60]

[60] Iš fibo naudoja fib, fib2

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ši forma neįtraukia modulio (iš kurio importuojama) vardo į lokalia simbolių lentelę (taigi šiame pavyzdyje `fibo` yra neapibrėžtas).

Egzistuojate netgi forma leidžianti importuoti visus vardus, kuriuos apibrėžia modulis:[61]

[61] Iš fibo naudoja *

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ši forma importuoja visus vardus išskyrus tuos kurie prasideda pabraukimu (_).

Moduliai kaip skriptai

Kai jūs vykdote Python'o modulį šitaip:

python fibo.py <arguments>

kodas modulyje bus įvykdytas, lyg jis būtų importuotas, bet `__name__` kintamasis bus nustatytas į `"__main__"`. Taigi jeigu jūs pridėsite tokį kodą į modulio pabaigą:[62]

[62] jei __name__ == "__main__":
naudoja sys
fib(int(sys.argv[1]))

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

jūs galite failą naudoti ir kaip importuojamą modulį ir kaip skriptą nes kodas, kuris tikrina komandinę eilutę vykdomas tik tada kaip modulis vykdomas kaip „pagrindinis“ („main“) failas:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Jeigu modulis bus importuotas, kodas nebus vykdomas:[63]

[63] naudoja fibo

```
>>> import fibo
>>>
```

Tai dažniausiai naudojama norint suteikti patogią naudotojo sąsają moduliui arba testavimo tikslais (pvz.: modulį paleidžiant kaip[64] skriptą įvykdomi testai).

[64] path - kelias

Modulio paieškos kelias

Kai modulis pavadintas :mod:`spam` yra importuojamas, interpretatorius ieško failo vardu :file:`spam.py` dabartinėje direktorijoje, o po to direktorijų sąraše, kurias nurodo aplinkos kintamasis :envvar:`PYTHONPATH`. Šis kintamasis naudoja tokią pačią sintaksę kaip ir apvalkalo kintamasis :envvar:`PATH`, t.y. aplankų vardų sąrašas. Jeigu :envvar:`PYTHONPATH` nėra nustatytas, arba kai modulis čia nerandamas, paieška tęsiama aplanke, kuris priklauso nuo įdiegimo. Unix-tipo sistemoje tai dažniausiai yra :file:`.:/usr/local/lib/python`.

Faktiškai, moduliai yra ieškomi aplankų sąraše, kuris laikomas kintamajame `sys.path`. Šis kintamasis yra inicializuojamas dabartiniu aplanku (t.y. tuo kur paleidžiamas skriptas), :envvar:`PYTHONPATH` ir aplanku, kuris priklauso nuo įdiegimo. Tai leidžia Python'o programoms, kurios supranta, kas yra daroma, modifikuoti arba pakeisti modulių paieškos kelius. Kadangi skripto aplankas yra įdedamas į paieškos kelią nevadinkite savo skripto vardu sutampančiu su kokiu nors standartiniu Python'o moduliu, nes kitaip Python'as importuojant kokį nors modulį importuos jūsų skriptą, o tai dažniausiai yra klaida. Daugiau informacijos apie tai skyriuje :ref:`tut-standardmodules`.

„Kompiliuoti“ Python'o failai

„Kompiliuoti“ Python'o failai yra svarbus pagreitinimas mažoms programoms, kurios naudoja daug modulių. Jeigu :file:`spam.pyc` egzistuoja tame pačiame aplanke kaip :file:`spam.py`, daroma prielaida, kad modulis :mod:`spam` jau turi „baitinę-kompiliuotą“ versiją. Į :file:`spam.pyc` įrašomas failo :file:`spam.py` pakeitimo laikas, todėl kai laikas esantis :file:`.pyc` nesutampa jis yra ignoruojamas.

Įprastai, jums nieko nereikia daryti, kad failas :file:`spam.pyc` būtų sukurtas. Kaskart, kai :file:`spam.py` yra sėkmingai sukompiliuojamas, jo kompiliuotą versiją bandoma rašyti į :file:`spam.pyc`. Jeigu tai nepavyksta, tai nėra klaida. Jei tarkime dėl kokių nors priežasčių įrašomas ne visas failas, tai vėliau jis bus atpažįstamas kaip neteisingas ir bus ignoruojamas. :file:`spam.pyc` turinys yra nepriklausomas nuo platformos, todėl Python'o modulių aplankas gali būti naudojamas vienu metu skirtingų architektūrų mašinų.

Keletas patarimų ekspertams:

- Kai Python'o interpretatorius yra iškviečiamas su :option:`-O`, sugeneruojamas optimizuotas kodas ir padedamas į failą :file:`.pyo`. Optimizatorius šiuo metu daug nepadeda: jis tik pašalina :keyword:`assert` [65] sakinius. Kai :option:`-O` yra naudojamas, *visas* :term:`baitinis kodas` yra optimizuotas, `.pyc` failai yra ignoruojami ir `.py` failai kompiliuojami į optimizuotą baitinį kodą.

[65] assert - teiginys, pranešimas

- Jeigu perduosite du :option:`-O` Python'o interpretatoriui (:option:`-OO`), tada baitinio kodo kompiliatorius atliks optimizacijas, kurios kai kuriais retais atvejais gali pagaminti neteisingai veikiančias programas. Šiuo metu tik `__doc__` eilutės yra pašalinamos iš baitinio kodo, dėl ko :file:`.pyo` failai yra mažesni. Kadangi, kai kurios programos gali remtis šių eilučių buvimo, jūs turite naudoti šią galimybę tik tada, kai tikrai žinote, ką darote.
- Programa paleista iš :file:`.pyc` arba :file:`.pyo` neveikia greičiau negu tada, kai ji vykdoma iš :file:`.py` failo --- vienintelis dalykas, kuo :file:`.pyc` arba :file:`.pyo` failai yra greitesni, tai greitis, kuriuo jie yra užkraunami.
- Kai skriptas vykdomas nurodant jo vardą komandinėje eilutėje, baitinis kodas niekada nėra rašomas į :file:`.pyc` ar :file:`.pyo` failą. Taigi, skripto paleidimo laikas gali būti sumažintas jo kodo dalis perkeliant į modulį ir naudojant mažą skriptą, kuris tą modulį importuoja.
- Galima turėti failą :file:`spam.pyc` (arba :file:`spam.pyo` kai naudojama :option:`-O`) be :file:`spam.py` failo tam pačiam moduliui. Tai gali būti panaudojama Python bibliotekų platinimui, nes tokią formą yra pakankamai sunku vėl atgal paversti skaitomu kodu.
- Modulis :mod:`compileall` gali sukurti :file:`.pyc` failus (arba :file:`.pyo` failus kai naudojama :option:`-O`) visiems moduliams aplanke.

Standartiniai moduliai

Įprastas Python'o įdiegimas turi standartinių modulių biblioteką. Kai kurie moduliai yra interpretatoriaus dalis --- jie leidžia atlikti operacijas, kurios nėra kalbos branduolio dalis, bet jie yra įtaisyti arba dėl efektyvumo arba, kad būtų galima atlikti operacinės sistemos primityvus (kaip kad sisteminius kvietimus). Tokių modulių aibė priklauso nuo platformos. Pavyzdžiui, modulis :mod:`winreg` yra aktualus tik Windows sistemai. Vienas konkretus modulis nusipelno šiek tiek dėmesio: :mod:`sys`, kuris yra kiekvieno Python'o interpretatoriaus dalis. Kintamieji `sys.ps1` ir `sys.ps2` apibrėžia eilutes, kurios naudojamos kaip pirminis ir antrinis raginimas:[66]

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
```

[66] naudoja sys
sys.ps1
'> > > '
sys.ps2
'... '
sys.ps1 = 'C> '
C> rodo 'Liuks!'
Liuks!
C>

```
C> print 'Yuck!'
Yuck!
C>
```

Šiek du kintamieji yra apibrėžti tik jei interpretatorius yra interaktyviame rėžime

Kintamasis `sys.path` yra eilučių sąrašas, kuris nustato interpretatoriaus modulių paieškos kelią. Jis inicializuojamas keliu paimtu iš aplinkos kintamojo :envvar:`PYTHONPATH` arba standartiniu keliu, jei :envvar:`PYTHONPATH` nėra nustatytas. Jūs galite jį modifikuoti naudodami standartines sąrašo operacijas:

```
>>> naudoja sys
>>> sys.path.append('/ufs/guido/lib/python')
```

:func:`dir` funkcija

Įtaisytoji funkcija :func:`dir` yra naudojama, kai norime sužinoti kokius vardus apibrėžia modulis. Ši funkcija sugrąžina surikiuotą eilučių sąrašą:

```
>>> naudoja fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

Jei funkcija :func:`dir` vykdoma be argumentų, tai surašomi vardai, kurie yra apibrėžti dabar:

```
>>> a = [1, 2, 3, 4, 5]
>>> naudoja fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Pastebėsime, kad surąšomi visų tipų vardai: kintamieji, moduliai, funkcijos ir t.t.

Funkcija :func:`dir` neišrašo įtaisytų funkcijų ir kintamųjų. Jeigu jūs norite gauti jų sąrašą, jie yra apibrėžti standartiniame modulyje :mod:`__builtin__`:

```
>>> naudoja __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UserWarning', 'ValueError', 'Warning', 'WindowsError',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', 'abs', 'apply', 'basestring', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex',
 'id', 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit', 'range',
 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set',
 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

Pakuotės

Pakuotės yra būdas struktūrizuoti Python'o modulių vardų edrvę naudojant „taškais atskirtus modulių vardus“. Pavyzdžiui, modulis :mod:`A.B` nurodo submodulį `B` pakuotėje `A`. Lygiai taip kaip moduliai apsaugo skirtingų modulių autorius nuo rūpesčio dėl vienodų globalių kintamųjų vardų, taip taškais atskirti modulių vardai apsaugo multi-modulinių pakuočių (tokių kaip NumPy ar Python Imaging Library) autorius nuo rūpesčio dėl vienodų modulių vardų.

Tarkime jūs norite sukurti modulių kolekciją („pakuotę“) universalią garso failų ir duomenų tvarkymui. Kadangi egzistuoja įvairūs failų formatai (dažniausiai atpažįstami pagal plėtinį, pvz.: :file:`.wav`, :file:`.aiff`, :file:`.au`), todėl jums reikės sukurti ir palaikyti augančią

modulių kolekciją įvairiems failų formatams. Taip pat egzistuoja daug skirtingų operacijų, kurias jus norite atlikti su garso duomenimis (miksavimas, aido pridėjimas, ekvalaizerio pritaikymas, dirbtinio stereo efekto sukūrimas), taigi jūs taip pat tūrėsite rašyti daug modulių, kurie atliks šias operacijas. Tai galėtų būti jūsų pakuotės struktūra (pavaizduota kaip hierarchinė failų sistema):

```
sound/                          Aukščiausio lygio pakuotė
      __init__.py               Garso pakuotės inicializacija
      formats/                  Subpakuotė failų formatų konvertavimui
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpakuotė garso efektams
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpakuotė filtrams
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Kai importuojame pakuotė, Python'as ieško pakuotės aplanko aplankuose nurodytose `sys.path`.

:file:`__init__.py` failas yra reikalingas tam, kad Python'as aplanką atpažintų kaip pakuotę. Taip daroma todėl, kad dažni vardai (tarkim `string`), netyčia nepaslėptų galiojančių modulių, kurie randami vėliau modulių paieškos kelyje. Paprasčiausiu atveju :file:`__init__.py` gali būti tiesiog tuščias failas, bet jei reikia jis gali įvykdyti inicializacijos kodą pakuotei arba nustatyti kintamąjį `__all__` (aprašytas vėliau).

Pakuotės naudotojai gali importuoti individualius modulius iš pakuotės. Pavyzdžiui:

```
naudoja sound.effects.echo
```

Ši komanda paleidžia submodulį :mod:`sound.effects.echo`. Norint jį naudoti reikia nurodyti pilną vardą:

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Alternatyvus būdas importuotį submodulį yra:

```
Iš sound.effects naudoja echo
```

Ši komanda taip pat paleidžia submodulį :mod:`echo` ir leidžia jį naudoti nenurodant pakuotės. Todėl jį galima naudoti taip:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Kitas variantas yra importuoti norimą funkciją ar kintamąjį tiesiogiai:

```
Iš sound.effects.echo naudoja echofilter
```

Vėlgi, tai užkrauną submodulį :mod:`echo`, bet jo funkcija :func:`echofilter` dabar pasiekiama tiesiogiai:

```
echofilter(input, output, delay=0.7, atten=4)
```

Atkreipkime dėmesį, kad naudojant `from pakuotė import narys`, *narys* gali būti pakuotės submodulis (ar subpakuotė), koks nors vardas, arba koks nors vardas apibrėžtas pakuotėje (funkcija, klasė ar kintamasis). `import` sakinys pirmiausia testuoja ar narys yra apibrėžtas pakuotėje ir jeigu ne tada daro prielaidą, kad narys yra modulis ir bando jį pakrauti. Jeigu ir tada jo neranda, tada sukelia išimtis :exc:`ImportError`.

Priešingai, kai naudojama sintaksė `import item.subitem.subsubitem`, kiekvienas narys išskyrus paskutinį privalo būti pakuotė. Paskutinis narys gali būti modulis, pakuotė, bet negali būti klasė, funkcija, kintamasis (kaip tai buvo galima daryti prieš tai).

** Importavimas iš pakuotės*

Kas atsitiks, jeigu naudotojas parašys `from sound.effects import *`? Idealiu atveju naudotojas tikėsis, kad ši komanda kažkaip nueis į failų sistemą, ras kurie submoduliai priklauso pakuotei ir juos visus importuos. Nelaimei ši operacija nedirba gerai Windows platformoje, kur failų sistema ne visada turi informaciją apie failo vardo raidžių lygį! Šioje platformoje, nėra garantuoto būdo žinoti ar :file:`ECHO.PY` turi būti importuotas kaip :mod:`echo`, :mod:`Echo` ar :mod:`ECHO`. (Pavyzdžiui, Windows 95 turi erzinantį įprotį rodyti visus failų vardus naudojant pirmą didžiąją raidę). DOS 8+3 failų vardų apribojimai prideda kitą įdomią problemą ilgiems modulių vardams.

Vienintelis sprendimas yra pakuotės autoriui išskirtinai išvardinti pakuotės turinį. *import* sakinys naudoja tokį susitarimą: jei pakuotės :file:`__init__.py` faile apibrėžiamas sąrašas `__all__` kintamajame,

jis turi būti naudojamas, kai modulių vardai importuojami naudojant `from package import *`. Pakuotės autorius yra atsakingas, kad šis sąrašas būtų atnaujinamas su kiekviena nauja pakuotės versija. Pakuotės autorius gali taip pat nuspręsti jo nepalaikyti, jeigu jis nemato prasmės naudoti * importavimą su jo pakuote. Pavyzdžiui :file:`sounds/effects/__init__.py` failo turinys gali būti toks:

```
__all__ = ["echo", "surround", "reverse"]
```

Tai reiškia, kad `from sound.effects import *` importuos tris išvardinus submodulius iš :mod:`sound` pakuotės.

Jeigu `__all__` nėra apibrėžtas, sakinys `from sound.effects import *` *neimportuoja* visų submodulių iš pakuotės :mod:`sound.effects` į dabartinę vardų erdvę --- tai tik užtikrins, kad pakuotė :mod:`sound.effects` bus importuota, įvykdys inicializacijos kodą :file:`__init__.py` faile ir tada importuos visus vardus apibrėžtus pakuotėje. Į tai įeina visi vardai apibrėžti (ir submoduliai išskirtinai paleisti) faile :file:`__init__.py`. Į tai taip pat įeina bet kokie pakuotės submoduliai, kurie buvo išskirtinai paleisti senesnių importavimo sakinių. Pažiūrėkite į šį kodą:

```
naudoja sound.effects.echo
naudoja sound.effects.surround
Iš sound.effects naudoja *
```

Šiame pavyzdyje, echo ir surround moduliai yra importuojami į dabartinę vardų erdvę, kadangi jie yra apibrėžti :mod:`sound.effects` pakuotėje, kai `from...import` sakinys yra įvykdomas. (Tai dirba taip pat, kai `__all__` yra apibrėžtas)

Pastebėsime, kad geriau nenaudoti * improtavimo iš modulių ar pakuočių, nes dažniausiai dėl to kodas tampa prastai skaitomas. Tačiau, nieko blogo tai daryti interaktyvioje sesijoje norint sutaupyti klaviatūros spausdinimo laiko, o taip pat kai kurie moduliai yra sukurti taip, kad būtų išeksportuojami tik tam tikri vardai.

Prisiminkite, kad nėra nieko blogo naudoti `from Pakuotė import specifinis_submodulis`! Tiesa sakant, tai rekomenduojamas būdas nebent submodulis iš kitos pakuotės naudoja tokį patį vardą.

Nuorodos tarp pakuočių

Submoduliai dažnai rodo vienas į kitą. Tarkime, modulis :mod:`surround` gali naudoti modulį :mod:`echo`. Tiesa sakant, tokios nuorodos yra tokios dažnos, kad :keyword:`import` sakinys pirmiausiai tikrina jau turimas pakuotes prieš ieškodamas modulių paieškos kelyje. Taigi, modulis :mod:`surround` gali tiesiog naudoti `import echo` arba `from echo import echofilter`. Jeigu importuojamas modulis nerandamas dabartinėje pakuotėje (t.y. pakuotėje, kurioje šis modulis yra

submodulis), tada :keyword:`import` sakinys ieško aukščiausio lygio modulio duotu vardu.

Kai pakuotė yra struktūrizuota į subpakuotes (pavyzdžiui kaip tai padaryta :mod:`sound` pakuotėje), jūs galite naudoti absoliučius importavimus norėdami nurodyti į gretimų pakuočių submodulius. Pavyzdžiui, jei modulis :mod:`sound.filters.vocoder` turi naudoti :mod:`echo` modulį iš :mod:`sound.effects` pakuotės, tai tada galima naudoti `from sound.effects import echo`.

Pradedant Python 2.5 versija, prie netiesiogiai nurodytų reliatyvių importavimų aprašytų viršuje, jūs galite naudoti griežtai nurodytus reliatyvius importavimus naudodami `from module import name` formą. Šie griežtai nurodyti reliatyvūs importavimai naudoją tašką, kuris nurodo dabartinę arba tėvinę pakuotę. Pavyzdžiui, iš :mod:`surround` modulio jūs galite naudoti:

```
Iš . naudoja echo
Iš .. naudoja formats
Iš ..filters naudoja equalizer
```

Atkreipsime dėmesį, kad tiek netiesiogiai tiek griežtai nurodyti importavimai paremti dabartinio modulio vardu. Kadangi pagrindinio modulio vardas visada yra `"__main__"` moduliai, kurie bus naudojami kaip pagrindiniai Python'o programos moduliai turi naudoti absoliučius importavimus.

Pakuotės keliose direktorijose

Pakuotės palaiko dar vieną specialų atributą --- :attr:`__path__`. Šis atributas inicializuojamas kaip sąrašas, kuriame yra aplankas, kuriame yra pakuotės :file:`__init__.py` failas (prieš tai, kai kodas yra įvykdomas tame faile). Šis kintamasis gali būti modifikuojamas ir tai paveiks kaip atliekama modulių ir subpakuočių paieška pakuotėje.

Nors ši savybė nėra dažnai naudojama, ji gali būti panaudota norint išplėsti modulių randamų pakuotėje aibę.

Įvedimas ir išvedimas

Egzistuoja keli būdai išvesti programos rezultatus. Duomenys gali būti išspausdinti žmogui skaitoma forma arba įrašyti į failą ateities naudojimui. Ši dalis apžvelgs kelias galimybes.

Gražesnis išvedimo formatavimas

Kol kas mes sutikome du reikšmių rašymo būdus: *išraiškos sakinius* ir :keyword:`print` sakinį. (trečias būdas :meth:`write` metodo naudojimas su failo objektais. Standartinis išvedimo failas gali būti pasiektas per `sys.stdout`.)

Dažnai jums reikės daugiau kontrolės formatuojant išvedamus duomenis negu tiesiog spausdinti tarpu atskirtas reikšmes. Egzistuoja du būdai suformatuoti jūsų išvedimo duomenis. Pirmas būdas yra visą formatavimą atlikti patiems: naudodami eilučių karpymo ir sujungimo operacijas jūs galite sukurti tokį išdėstymą kokį tik įsivaizduojate. Standartiniame modulyje :mod:`string` galima rasti kelias naudingas operacijas eilučių užpildymui iki reikiamo stulpelio pločio (jas peržvelgsime netrukus). Antras būdas yra naudoti metodą :meth:`str.format`.

Žinoma, išlieka vienas klausimas: kaip konvertuoti reikšmes į eilutes? Laimei Python'as turi būdą reikšmes paversti eilutėmis: perduokite reikšmę funkcijoms: :func:`repr` arba :func:`str`.

Funkcijos :func:`str` paskirtis yra sugrąžinti reikšmę, kurią galėtų perskaityti žmogus, tuo tarpu :func:`repr` paskirtis yra sugeneruoti reprezentaciją, kurią galėtų perskaityti interpretatorius (arba jei tokias formos nėra grąžintų :exc:`SyntaxError` klaidą). Objektams, kurie neturi žmogui skaitomos formos, :func:`str` sugrąžins tokią pačią reikšmę kaip :func:`repr`. Dauguma reikšmių, kaip kad skaičiai, struktūros, sąrašai ar žodynai, turi tą pačia reprezentaciją abiems funkcijoms. Eilutės ir slankaus kablelio skaičiai --- turi dvi skirtingas reprezentacijas.

Keletas pavyzdžių:

```
>>> s = 'Sveikas, pasauli.'
>>> str(s)
'Sveikas, pasauli.'
```

```
>>> repr(s)
"'Sveikas, pasauli.'"
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Reikšmė x yra ' + repr(x) + ', ir y yra ' + repr(y) + '...'
>>> rodo s
Reikšmė x yra 32.5, and y is 40000...
>>> # eilutės repr() prideda kabutes ir kairinį brūkšnį:
>>> sveikas = 'sveikas, pasauli\n'
>>> sveiki = repr(sveikas)
>>> rodo sveiki
'sveikas, pasauli\n'
>>> # repr() argumentu gali būti bet koks Python'o objektas:
... repr((x, y, ('šlamštas', 'kiaušiniai')))
"(32.5, 40000, ('šlamštas', 'kiaušiniai'))"
```

Čia yra du būdai parašyti kvadratų ir kubų lentelę:[67,68]

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Atkreipkite dėmesį į kablelį eilutės gale
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000

>>> for x in range(1,11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
```

[67]
```
  kožnam x iš range(1, 11):
      rodo repr(x).rjust(2),
repr(x*x).rjust(3),
      rodo repr(x*x*x).rjust(4)
```

[68]
```
  kožnam x iš range(1,11):
      rodo '0:2d 1:3d 2:4d'.format(x, x*x,
x*x*x)
```

```
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Atkreipkite dėmesį į tai, kad pirmame pavyzdyje vienas tarpas tarp stulpelių buvo pridėtas dėl to kaip :keyword:`print` dirba: ši komanda visada prideda tarpus tarp argumentų.)

Šis pavyzdys demonstruoja eilučių objektų metodą :meth:`rjust`, kuris sulygina eilutes pagal dešinę pusę jas užpildydamas tarpais kairėje. Egzistuoja panašūs metodai :meth:`ljust` ir :meth:`center`. Šie metodai nieko nerašo --- jie tiesiog sugrąžina eilutę. Jei paduota eilutė yra per ilga, ji nėra sutrumpinama, bet grąžinama nepakeista. Tai žinoma sugadins jūsų išdėstymą, bet tai geriau negu sutrumpinta eilutė (dėl ko būtų rodoma neteisinga reikšmė). Jeigu jus tenkina, kad eilute bus sutrumpinta tai galite atlikti kirpdami, pvz.: `x.ljust(n)[:n]`.

Yra kitas metodas :meth:`zfill`, kuris užpildo skaitines reikšmes nuliais iš kairės pusės. Šis metodas taip pat supranta plius ir minus ženklus:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Paprastas :meth:`str.format` metodo naudojimas atrodo taip:

```
>>> rodo 'Mes esame {0} kurie sako "{1}!"'.format('riteriai', 'Šlovė')
Mes esame riteriai kurie sako "Šlovė!"
```

Figūriniai skliausteliai ir simboliai tarp jų (vadinami formatavimo laukais) yra pakeičiami objektais perduotais formatavimo metodui. Skaičius tarp figūrinių skliaustelių nurodo formatavimo metodui perduoto objekto poziciją:

```
>>> rodo '{0} and {1}'.format('šlamštas', 'kiaušiniai')
šlamštas ir kiaušiniai
>>> rodo '{1} and {0}'.format('šlamštas', 'kiaušiniai')
kiaušiniai ir šlamštas
```

Jei vardiniai argumentai yra naudojami formatavimo metode, jų reikšmės yra nurodomos naudojant argumento vardą.

```
>>> rodo 'Šis {maistas} yra {būdvardis}.'.format(
...     maistas='šlamštas', būdvardis='visiškai siaubingas')
Šis šlamštas yra visiškai siaubingas.
```

Poziciniai ir vardiniai argumentai gali būti naudojami vienu metu:

```
>>> rodo '{0}, {1}, ir {kitas} istorija.'.format('Billo', 'Manfredo',
...                                              kitas='Georgo')
Billo, Manfredo ir Georgo istorija.
```

Papildomai po lauko vardo gali eiti `':'` ir formatavimo nurodymas.
Tai taip pat leidžia labiau kontroliuoti kaip reikšmė yra formatuojama.
Sekantis pavyzdys sutrumpina Pi reikšmę iki trijų ženklų po kablelio:

```
>>> naudoja math
>>> rodo 'PI reikšmė apytiksliai yra  {0:.3f}.'.format(math.pi)
PI reikšmė apytiksliai yra 3.142.
```

Jeigu po `':'` bus perduotas skaičius, tai reikš, kad laukas užims mažiausiai tiek simbolių. Tai patogu norint parodyti lenteles gražiai:

```
>>> lentelė = {'Sjoerdas': 4127, 'Jackas': 4098, 'Dcabas': 7678}
>>> kožnam vardas, telefonas iš lentelė.items():
...     rodo '{0:10} ==> {1:10d}'.format(vardas, telefonas)
...
Jackas       ==>       4098
Dcabas       ==>       7678
Sjoerdas     ==>       4127
```

Jeigu jūs turite labai ilgą formatavimo eilutę, kurios jūs nenorite skaidyti, būtų labai patogu, jei galėtumėte pasiekti kintamuosius pagal vardą, o ne pagal poziciją. Tai galima padaryti tiesiog perduodant žodyną ir naudojant laužtinius skliaustelius `[]`, kai reikia pasiekti raktus

```
>>> lentelė = {'Sjoerdas': 4127, 'Jackas': 4098, 'Dcabas': 8637678}
>>> rodo ('Jackas: {0[Jackas]:d}; Sjoerdas: {0[Sjoerdas]:d}; '
...        'Dcabas: {0[Dcabas]:d}'.format(lentelė))
Jackas: 4098; Sjoerdas: 4127; Dcabas: 8637678
```

Tą patį galima atlikti perduodant lentelę kaip vardinius argumentus naudojant '**' užrašą:

```
>>> lentelė = {'Sjoerdas': 4127, 'Jackas': 4098, 'Dcabas': 8637678}
>>> rodo 'Jackas: {Jackas:d}; Sjoerdas: {Sjoerdas:d}; Dcabas: {Dcabas:d}'.format(**lentelė)
Jackas: 4098; Sjoerdas: 4127; Dcabas: 8637678
```

Tai ypač patogu naudoti kartu su įtaisyta funkcija :func:`vars`, kuri sugrąžina žodyną, kuriame laikomi visi lokalūs kintamieji.

Senas eilučių formatavimas

Operatorius % taip pat gali būti naudojamas eilučių formatavimui. Jis interpretuoja argumentą panašiai kaip C funkcija `sprintf`. Pavyzdžiui:

```
>>> naudoja math
>>> naudoja 'PI reikšmė apytiksliai lygi %5.3f.' % math.pi
PI reikšmė apytiksliai lygi 3.142.
```

Kadangi :meth:`str.format` yra pakankamai naujas, dauguma Python'o kodo vis dar naudoja % operatorių. Tačiau, kadangi šis seno stiliau formatavimas ilgainiui iš kalbos bus pašalintas rekomenduojama naudoti :meth:`str.format`.

Failų skaitymas ir rašymas

:func:`open` grąžina failo objektą, ir yra dažnai naudojama su dviem argumentais: `open(failovardas, rėžimas)`.

```
>>> f = open('/tmp/workfile', 'w')
>>> rodo f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Pirmas argumentas yra eilutė nurodanti failo vardą. Antras argumentas yra kita eilutė, kurioje yra keli simboliai nurodantys kaip failas bus naudojamas. *rėžimas* gali turėti reikšmes `'r'` kai failas bus tik skaitomas, `'w'` kai tik rašomas (egzistuojantis failas tokiu pat vardu bus ištrintas), ir `'a'` atidaro failą papildymui. Bet kokie duomenys rašomi į failą pridedami į jo galą. `'r+'` atidaro failą ir skaitymui ir rašymui. *rėžimo* argumentas yra nebūtinas: jeigu jis bus praleistas, tai bus daroma prielaida, kad rėžimas yra `'r'`.

Windows sistemoje, `'b'` pridėjimas prie rėžimo atidaro failą dvejetainiame rėžime. Taigi rėžimai gali atrodyti taip: `'rb'`, `'wb'`, ir `'r+b'`. Windows sistemoje yra skirtumas tarp tekstinių ir dvejetainių failų: eilutės pabaigos simbolis tekstiniuose failuose yra modifikuojamas, kai duomenys yra skaitomi arba rašomi. Ši modifikacija nepakenkia ASCII failams, bet dvejetainius failus (kaip kad :file:`JPEG` arba :file:`EXE`) ji sugadina. Būkite labai atsargūs ir nepamirškite naudoti dvejetainio rėžimo, kai skaitote ar rašote failus. Unix sistemose `'b'` simbolio pridėjimas į rėžimo eilutę žalos nedaro, taigi pridėkite jį, kad jūsų kodas būtų nepriklausomas nuo platformos.

Failų objektų metodai

Likę pavyzdžiai šioje dalyje tikėsis, kad failo objektas `f` yra jau sukurtas.

Tam, kad perskaitytumėte failo turinį, iškvieskite `f.read(size)`. Tai perskaitys dalį duomenų ir grąžins juos kaip eilutę. *size* yra nebūtinas skaitinis argumentas. Jeigu *size* yra praleidžiamas arba neigiamas, tada perskaitomas visas failo turinys ir sugrąžinamas. Tačiau, jei failas yra dukart didesnis negu yra atminties jūsų mašinoje, tai jau jūsų problema. Kitu atveju daugiausiai *size* baitų yra perskaitoma ir sugrąžinama. Jeigu pasiekiamas failo galas `f.read()` sugrąžina tuščią eilutę (`""`).

```
>>> f.read()
'Tai yra failo turinys.\n'
>>> f.read()
''
```

`f.readline()` perskaito vieną eilutę iš failo: naujos eilutės simbolis (`\n`) yra paliekamas eilutės gale ir jo ten nebus tik tuo atveju, jei paskutinė eilutė neturi eilutės pabaigos simbolio. Dėl to sugrąžinta reikšmė tampa aiški, jei `f.readline()` sugrąžina tuščią eilutę, reiškia failo pabaiga yra pasiekta. Tuo tarpu tuščią eilutę nurodo `'\n'` --- eilutė, kurioje yra tik naujos eilutės simbolis.

```
>>> f.readline()
'Tai yra pirma eilute is failo.\n'
>>> f.readline()
'Antra eilute is failo\n'
>>> f.readline()
''
```

`f.readlines()` sugrąžina visas eilutes iš failo. Jeigu paduodamas nebūtinas parametras *sizehint*, ši funkcija perskaito tiek baitų iš failo ir dar šiek tiek, kad pabaigtų eilutę ir tada sugrąžina eilutę. Tai dažnai naudojama norint efektyviai skaityti didelius failus eilutėmis ir tuo pačiu neužkrauti viso failo į atmintį. Tik pilnos eilutės bus sugrąžintos:

```
>>> f.readlines()
['Tai yra pirma eilute is failo.\n', 'Antra eilute is failo\n']
```

Alternatyvus būdas skaityti eilutes iš failo yra pereiti per failą. Tai daug efektyviau atminties atžvilgiu, greičiau veikia ir kodas daug paprastesnis:

```
>>> kožnam eilutė iš f:
        rodo eilutė,

Tai yra pirma eilute is failo.
Antra eilute is failo
```

Alternatyvus būdas yra paprastesnis, bet nesuteikia tiek kontrolės. Kadangi du būdai valdo eilučių buferį skirtingai, jei neturėtų būti maišomi.

`f.write(string)` įrašo *eilutės* turinį į failą ir grąžina `None`.:

```
>>> f.write('Tai yra testas\n')
```

Jeigu norite įrašyti, ką nors kitą negu eilutė, pirmiausia turite tai konvertuoti į eilutę:

```
>>> reikšmė = ('atsakymas', 42)
>>> s = str(reikšmė)
>>> f.write(s)
```

`f.tell()` sugrąžina skaitinę reikšmę nurodančią dabartinę failo objekto poziciją faile, kuri matuojama baitas nuo failo pradžios. Jei norite pakeisti failo objekto poziciją, naudokite `f.seek(poslinkis, nuo_ko)`. Pozicija yra apskaičiuojama pridedant *poslinkio* reikšmę nuo atskaitos taško. Atskaitos tašką apsprendžia *nuo_ko* argumentas. *nuo_ko* gali turėti tokias reikšmes: 0 --- poslinkis nuo failo pradžios, 1 --- dabartinė pozicija, 2 --- nuo failo pabaigos. Jeigu *nuo_ko* yra praleidžiamas, tai naudojama reikšmė 0, t.y. nuo failo pradžios:

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Eik prie 6-to failo baito
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Eik prie 3-io baito nuo galo
>>> f.read(1)
'd'
```

Jeigu baigėte darbą su failų, iškvieskite `f.close()` tam, kad uždarytumėte jį ir atlaisvintumėte sisteminius resursus, kuriuos šio failas atidarymas yra paėmęs. Po `f.close()` iškvietimo bet kokie bandymai naudoti failo objektą automatiškai nepavyks.

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

Gera praktika yra naudoti :keyword:`with` bazinę žodį dirbant su failų objektais. Šio bazinio žodžio privalumas yra tas, kad failas yra tinkamai uždaromas, kai jo naudojimas yra baigiamas, net jei pakeliui yra sukeliama išimtis. Be to tai daug trumpiau negu rašyti :keyword:`try` - :keyword:`finally` blokus:

```
>>> su open('/tmp/workfile', 'r') kaip f:
...     skaito_duomenis = f.read()
>>> f.closed
True
```

Failų objektai turi kelis papildomus metodus, kaip kad :meth:`isatty` ir :meth:`truncate`, kurie yra daug rečiau naudojami.

:mod:`pickle` modulis

Eilutes yra paprasta rašyti ir skaityti iš failo. Skaitines reikšmes reikalauja šiek tiek daugiau pastangų, kadangi :meth:`read` metodas grąžina tik eilutes, kurias po to reikia perduoti funkcijoms kaip kad :func:`int`, kurios gavusios eilutę `'123'` grąžina skaitinę reikmę 123. Tačiau, jeigu jums reikia saugoti sudėtingesnius duomenų tipus kaip sąrašus, žodynus ar klasių egzempliorius, viskas pasidaro daug sudėtingiau.

Tam, kad naudotojui nereiktų nuolat vargti rašant ir derinant kodą norint išsaugoti sudėtingus duomenų tipus, Python'as turi standartinį modulį :mod:`pickle`. Tai yra nerealus modulis, kuris gali paimti beveik bet kurį Python'o objektą (netgi tam tikras Python'o kodo formas!) ir sukonvertuoti jį į eilutę. Šis procesas vadinamas marinavimu (angl. :dfn:`pickling`). Objekto rekonstravimas iš eilutės yra vadinamas išmarinavimu (angl. :dfn:`unpickling`). Tarp marinavimo ir išmarinavimo, objektas eilutėje gali būti padėtas į failą arba persiųstas per tinklą į kitą mašiną.

Jeigu jūs turite objektą `x` ir failo objektą `f` atidarytą rašymui, paprasčiausias būdas marinuoti objektą užima tik vieną eilutę:

```
pickle.dump(x, f)
```

Jei norite išmarinuoti objektą ir turite failo objektą `f`, kuris atidarytas skaitymui naudokite:

```
x = pickle.load(f)
```

(Yra ir daugiau variantų kaip tai padaryti, kurie naudojami marinuojant daug objektų arba kai jūs nenorite rašyti marinuotų duomenų į failą)

:mod:`pickle` yra standartinis būdas Python'e duomenų saugojimui ir naudojimui kitose programose (arba toje pačioje programoje, kai ji bus iškviesta ateityje). Techninis terminas tam yra ilgalaikis (angl. :dfn:`persistent`) objektas. Kadangi :mod:`pickle` yra dažnai naudojamas, daugelis autorių rašančių Python'o plėtinius pasirūpina, kad naujus duomenų tipus (pvz.: matricas) būtų galima tinkamai marinuoti ir išmarinuoti.

Klaidos ir išimtys

Iki šiol klaidų pranešimai buvo tik minimi, bet jeigu jūs bandėte vykdyti pavyzdžius jūs tikriausiai kelias matėte. Egzistuoja mažiausiai dvi skirtingos klaidų rūšys: *sintaksės klaidos* ir *išimtys*.

Sintaksės klaidos

Sintaksės klaidos taip pat žinomos kaip teksto analizės klaidos, yra turbūt dažniausiai pasitaikanti klaidų rūšis, kol jūs dar mokotės Python'o:

::

```
>>> kol True rodo 'Sveikas pasauli'
    ^
File "<stdin>", line 1, in ?
while True print 'Sveikas pasauli'

SyntaxError: invalid syntax
```

Teksto analizatorius pakartoja eilutę su klaida ir parodo mažą 'rodyklę', kur pirmiausiai buvo surasta klaida. Klaida yra sukelta ar bent jau nustatyta ties komanda, kuri yra aprašyta prieš rodyklę. Pavyzdyje klaida rasta bazinio žodžio :keyword:`rodo` naudojime, nes prieš jį trūksta dvitaškio (`':'`). Failo vardas ir eilutės numeris yra atspausdinami, kad jūs žinotumėte, kur žiūrėti, jei klaida įvyko vykdant skriptą.

Išimtys

Net jei sakinys ar reiškinys yra sintaksiškai teisingi, tai gali sukelti klaidą, kai juos bandoma įvykdyti. Klaidos, kurios nustatomos vykdymo metu yra vadinamos *išimtimis* ir yra besąlygiškai neišvengiamos: jūs netrukus išmoksite kaip jas suvaldyti Python'o programose. Tačiau dauguma išimčių nėra suvaldomos programoje ir baigiasi klaidos pranešimu kaip parodyta čia:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Paskutinė klaidos pranešimo eilutė nurodo, kas įvyko. Išimtys būna skirtingų tipų ir tipas parašomas kaip dalis pranešimo. Pavyzdyje tipai yra :exc:`ZeroDivisionError`, :exc:`NameError` ir :exc:`TypeError`. Eilutė, kuri parašyta kaip išimties tipas, yra įtaisytos išimties, kuri įvyko, vardas. Tai galioja visoms įtaisytoms išimtims, bet nebūtinai yra tiesa naudotojo apibrėžtoms išimtims (nors tai ir yra naudingas susitarimas). Standartinių išimčių vardai yra įtaisyti identifikatoriai (ne rezervuoti baziniai žodžiai).

Likusi eilutės dalis pateikia detalią informaciją priklausomai nuo išimties tipo ir kas sukėlė išimtį.

Priešais einanti klaidos pranešimo dalis parodo kontekstą, kuriame įvyko išimtis dėklo pėdsakų (angl. traceback) forma. Bendrai jame yra dėklo pėdsakai nurodantys kodo eilutes, tačiau eilutės perskaitytos iš standartinės įvesties.

Išimčių valdymas

Programą galima parašyti taip, kad ji suvaldytų pasirinktas išimtis. Pažiūrėkime į žemiau esantį pavyzdį, kuris prašo naudotoją įvesti teisingą skaičių tol kol jis yra įvedamas, bet taip pat leidžia naudotojui nutraukti programą (naudojant :kbd:`Control-C` ar bet kokį kitą būdą, kurį palaiko operacinė sistema). Pastebėsime, kad naudotojo sugeneruotas nutraukimas yra signalizuojamas sukeliant :exc:`KeyboardInterrupt` išimtį:[69]

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

[69]
```
kol True:
    bando:
        x = int(raw_input("Prašom įvesti skaičių: "))
        lūžis
    išimtis ValueError:
        rodo "Ne! Tai ne skaičius. Bandykite dar..."
```

:keyword:`try` (bando) sakinys dirba taip:

- Pirmiausia, vykdomas *try (bando) paragrafas* (tarp :keyword:`try` (bando) ir :keyword:`except` (išimtis) bazinių žodžių)
- Jei išimtis neįvyksta, *except (išimtis) paragrafas* yra praleidžiamas ir :keyword:`try` (bando) sakinys yra baigiamas.
- Jei išimtis įvyksta *try* (bando) vykdymo metu, likusi paragrafo dalis yra praleidžiama. Jeigu išimties tipas sutampa su vardu po :keyword:`except` (išimtis) bazinio žodžio, tada except (išimtis) paragrafas yra įvykdomas ir tada vykdymas tęsiamas po :keyword:`try` (bandos) sakinio.
- Jei įvyksta išimtis, kuri nesutampa su išimtimi paminėtoje except (išimtis) paragrafe, ji perduodama išoriniam :keyword:`try` (bando) sakiniui. Jeigu nesurandama, kas suvaldytų išimtį, ji tampa *nesuvaldyta išimtimi*, vykdymas yra sustabdomas ir parodomas klaidos pranešimas kaip parodyta aukščiau.

:keyword:`try` (bando) sakinys gali turėti daugiau negu vieną except (išimtis) paragrafą, jei norima suvaldyti skirtingas išimtis. Daugiausiai bus įvykdytas vienas paragrafas. *except* (išimtis) dalyje suvaldomos tik išimtys, kurios ateina iš try (bando) paragrafo, bet ne iš kitų to paties :keyword:`try` (bando) sakinio except (išimtis) dalių. Except (išimtis) paragrafe galima išvardinti kelias išimtis kortežo formoje, pavyzdžiui:

```
... išimtis (RuntimeError, TypeError, NameError):
...     pass
```

Paskutiniame išimtis paragrafe galima praleisti išimties vardą ir tai bus interpretuojama kaip bet kokia išimtis. Naudokite šią galimybę ypač atsargiai, nes taip galima paslėpti tikras programavimo klaidas! Tai taip pat galima naudoti klaidos pranešimo atspausdinimui ir tada vėl sukelti išimtį (taip leidžiant kvietėjui suvaldyti išimtį taip pat):[70]

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
```

[70]

```
  naudoja sys
  bando:
     f = open('manofailas.txt')
     s = f.readline()
     i = int(s.strip())
  išimtis IOError kaip (errno, strerror):
     rodo "I/O klaida(0): 1".format(errno,
strerror)
  išimtis ValueError:
     rodo "Neįmanoma konvertuoti
duomenų į sveikuosius."
  išimtis:
     rodo "Nenumatyta klaida:",
sys.exc_info()[0]
     išveda
```

```
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

:keyword:`try` ... :keyword:`except` sakinys turi papildomą *else paragrafą*, kuris, jam esant, turi sekti po visų except paragrafų. Jis naudingas tuo atveju, jei try paragrafas nesukelia išimties. Pavyzdžiui:[71]

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

:keyword:`else` naudojimas yra geresnis variantas negu papildomo kodo rašymas :keyword:`try` paragrafe, kadangi tai padeda išvengti atsitiktinio išimties pagavimo, kuri nebuvo sukelta kodo, kuris yra apsaugotas :keyword:`try` ... :keyword:`except` sakiniu.

Kai įvyksta išimtis, ji gali turėti susietą reikšmę, taip pat žinomą kaip išimties *argumentas*. Argumento egzistavimas ir tipas priklauso nuo išimties tipo.

Išimties sakinys gali nurodyti kintamąjį po išimties vardo (arba kortežo). Kintamasis yra susietas su išimties egzemplioriumi per kintamąjį padėtą į `instance.args`. Patogumo dėlei, išimties egzempliorius apibrėžia :meth:`__getitem__` ir :meth:`__str__` tam, kad argumentus būtų galima pasiekti arba atspausdinti tiesiogiai nesikreipiant į `.args`.

Be to `.args` naudojimas yra nerekomenduotinas. Vietoje to, rekomenduojama perduoti vieną argumentą išimčiai (kas gali būti kortežas, jeigu reikia perduoti kelis argumentus) ir susieti jį su `message` atributu. Taip pat galima inicializuoti išimtį prieš ją sukeliant ir pridėti bet kokius norimus atributus:[72]

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print type(inst)     # the exception instance
...     print inst.args      # arguments stored in .args
...     print inst           # __str__ allows args to printed directly
...     x, y = inst          # __getitem__ allows args to be unpacked directly
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
```

71

```
kožnam arg iš sys.argv[1:]:
    bando:
        f = open(arg, 'r')
    išimtis IOError:
        rodo 'cannot open', arg
    kitaip:
        rodo arg, 'has', len(f.readlines()),
'lines'
        f.close()
```

72

```
bando:
    išveda Exception('spam', 'eggs')
išimtis Exception kaip inst:
    # išimties sritis
    print type(inst)
    # argumentai išsaugoti .args
    rodo inst.args
    # __str__ įgalina args rodyti tiesiogiai
    rodo inst
    # __getitem__ įgalina args išpakuoti
tiesiogiai
    x, y = inst
    rodo 'x =', x
    rodo 'y =', y
```

```
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Jei išimtis turi argumentą, jis yra atspausdinamas kaip paskutinė pranešimo dalis nesuvaldytai išimčiai.

Išimtys yra suvaldomos ne tik tada, kai jos iškart seka try paragrafe, bet taip pat jeigu jos įvyksta funkcijos viduje, kuri yra iškviesta try paragrafe. Pavyzdžiui:[73]

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

[73]
```
apif tai_lūžta():
    x = 1/0
...
bando:
    tai_lūžta()
išimtis ZeroDivisionError kaip detail:
    rodo 'Handling run-time error:',
detail
```

Išimčių sukėlimas

:keyword:`raise` sakinys leidžia programuotojui sukelti norimą išimtį. Pavyzdžiui:[74]

[74] išveda NameError, 'HiThere'

```
>>> raise NameError, 'HiThere'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

Pirmas :keyword:`raise` argumentas yra išimties, kurią norima sukelti, vardas. Papildomas antrasis argumentas nurodo išimties argumentą. Alternatyviai tą patį galima parašyti kaip `raise NameError('HiThere')`. Abi formos dirba gerai, bet atrodo, kad žmonėms pastaroji forma patinka labiau.

Jeigu jums tik reikia sužinoti ar išimtis buvo sukelta bet jūs nenorite jos suvaldyti, paprastesne :keyword:`raise` sakinio forma leidžia vėl sukelti išimtį:[75]

```
>>> try:
...     raise NameError, 'HiThere'
... except NameError:
...     print 'An exception flew by!'
...     raise
```

[75]
```
bando:
    išveda NameError, 'HiThere'
išimtis NameError:
    rodo 'An exception flew by!'
    išveda
```

```
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

Naudotojų apibrėžtos išimtys

Programos gali sukurti savo išimtis sukurdamos naujas išimčių klases. Išimtys įprastai turėtų būti paveldėtos iš :exc:`Exception` klasės (tiesiogiai ar netiesiogiai). Pavyzdžiui:[76]

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Šiame pavyzdyje numatytasis :class:`Exception` klasės metodas :meth:`__init__` buvo perrašytas. Naujas elgesys paprasčiausiai sukuria *value* atributą. Tai pakeičia įprasta *args* atributo sukūrimą.

Išimčių klasės gali apibrėžti bet ką, ką daro kitos klasės, bet dažniausiai būną paprastos, dažniausiai leidžiančios tik daugiau atributų saugoti informacijai apie klaidą. Kai kuriamas modulis, kuris gali sukelti kelias skirtingas išimtis, dažna praktika yra sukurti bazinę klasę išimtims apibrėžtoms tame modulyje ir tada kitas išimčių klases paveldėti iš tos klasės:[77]

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.
```

[76]

```
klasė ManoKlaida(Exception):
    apif __init__(self, value):
        self.value = value
    apif __str__(self):
        grąžina repr(self.value)
...
bando:
    išveda ManoKlaida(2*2)
išimtis ManoKlaida kaip e:
    rodo 'Čia mano išimtis, jos reikš-
mė:', e.value
```

[77]

```
klasė Klaida(Exception):
    """ bazinė išimties klasė modulyje """
    pass
klasė ĮvedimoKlaida(Klaida):
    """Išimtis išvedama kai klaidingas
įvedimas
    Atributai:
        expression -- įvedimo expression
kuriame yra saugoma klaida
        message -- klaidos paaiškinimas
    """
    apif __init__(self, expression, messa-
ge):
        self.expression = expression
        self.message = message
```

```
    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message
```

78

```
class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

78

```
klasė VeiksmoKlaida(Klaida):
    """Išvedama kai vykdomas veiksmas nėra leistinas
    Atributai:
        previous -- būsena veiksmo pradžioje
        next -- pasiekta nauja būsena
        message -- paaiškinimas kodėl transakcija neleistina
    """
    apif __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Dauguma išimčių pavadinimas baigiasi „Error“, taip kaip ir standartinės išimtys.

Dauguma standartinių modulių apibrėžia savo išimtis klaidų, kurios gali įvykti modulyje apibrėžtose funkcijose, pranešimui. Daugiau informacijos apie klases galima rasti skyriuje :ref:`tut-classes`.

Išvalymo apibrėžimas

:keyword:`try` sakinys turi dar vieną papildomą paragrafą, kuris yra skirtas išvalymui ir yra įvykdomas bet kokiomis sąlygomis. Pavyzdžiui:[79]

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
```

79

```
bando:
    išveda KeyboardInterrupt
galutinai:
    rodo 'Goodbye, world!'
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

finally paragrafas yra vykdomas visada prieš paliekant :keyword:`try` sakinį, nepaisant to ar išimtys įvyko ar ne. Jeigu išimtis įvyko :keyword:`try` paragrafe ir nebuvo suvaldyta nei :keyword:`except` paragrafe (arba jei ji įvyko :keyword:`except` arba :keyword:`else` paragrafuose), ji yra vėl sukeliama po :keyword:`finally` paragrafo įvykdymo. :keyword:`finally` paragrafas yra vykdomas ir tuo atveju, jei kitas :keyword:`try` sakinio paragrafas yra paliekamas įvykdžius :keyword:`break`, :keyword:`continue` ar :keyword:`return` sakinį. Šiek tiek sudėtingesnis pavyzdys (:keyword:`except` ir :keyword:`finally` paragrafai tame pačiame :keyword:`try` sakinyje dirba nuo Python 2.5 versijos):[80]

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Kaip matote :keyword:`finally` paragrafas yra įvykdomas bet kokiu atveju. :exc:`TypeError` sukelta dalinant dvi eilutes nėra suvaldoma :keyword:`except` paragrafe todėl vėl sukeliama po to kai :keyword:`finally` paragrafas baigia savo darbą.

Realiame pasaulyje, :keyword:`finally` paragrafas praverčia atlaisvinant išorinius resursus (failai ar tinklo jungtys), nepaisant to ar resursų panaudojimas buvo sėkmingas ar ne.

[80]
```
apif dalina(x, y):
    bando:
        resultas = x / y
    išimtis ZeroDivisionError:
        rodo "dalyba iš nulio!"
    kitaip:
        rodo "rezultatas yra", resultas
    galutinai:
        rodo "vykdoma galutinė dalis"
...
```

Numatytas išvalymas

Kai kurie objektai apibrėžia standartinius išvalymo veiksmus tam atvejui kai paimtas objektas yra nebereikalingas --- nesvarbu ar operacija, kuri naudoja tą objektą pavyko ar baigėsi nesėkme. Pažiūrėkime į sekantį pavyzdį, kuriame bandoma atidaryti failą ir atspausdinti jo turinį į ekraną:[81]

```
for line in open("myfile.txt"):
    print line
```

[81]
```
kožnam eilutė iš open("myfile.txt"):
    rodo eilutė
```

Šio kodo problema yra ta, kad failas yra paliekamas atidarytas neapibrėžtam laiko tarpui po to, kai kodas baigia vykdymą. Tai nėra problema paprastame skripte, bet gali būti problema didelėje programoje. :keyword:`with` sakinys leidžia objektus naudoti taip, kad jie visada teisingai ir laiku būtų išvalyti:[82]

```
with open("myfile.txt") as f:
    for line in f:
        print line
```

[82]
```
su open("myfile.txt") kaip f:
    kožnam eilutė iš f:
        rodo eilutė
```

Po sakinio vykdymo, failas *f* yra visada uždaromas, netgi jei kilo kažkokių problemų skaitant eilutes. Jei objektai turi numatytą išvalymą, tai būna paminėta dokumentacijoje.

Klasės

Python'o klasių mechanizmas prideda klases į kalbą naudodamas minimalų kiekį naujos sintaksės ir semantikos. Naudojamas klasių mechanizmo randamo C++ ir Modula-3 kalbose mišinys. Lygiai taip pat kaip ir su moduliais, Python'o klasėse nėra absoliučios ribos tarp apibrėžimo ir naudotojo -- tiesiog tikimasi naudotojo suprantingumo „nesulaužant apibrėžimo“. Visgi svarbiausios klasių savybės yra pilnai išsaugotos: paveldėjimo mechanizmas leidžia naudoti kelias bazines klases, paveldėta klasė gali perrašyti bazinės klasės/bazinių klasių metodus ir metodai gali iškviesti bazinės klasės metodus tuo pačiu vardu. Objektuose galima laikyti reikiamą privačių duomenų kiekį.

Jei naudosime C++ terminologiją, visi klasės nariai (taip pat duomenys) yra *vieši*, visos klasės funkcijos yra *virtualios*. Taip pat neegzistuoja specialūs konstruktoriai ar destruktoriai. Lygiai taip pat kaip Modula-3 neegzistuoja sutrumpinimas norint pasiekti objekto narius iš jo metodo: metodo funkcija yra apibrėžiama naudojant pirmą argumentą, kuris nurodo objektą, ir šis argumentas visada paduodamas kviečiant funkciją. Kaip ir Smalltalk pačios klasės yra objektai, arba platesne žodžio prasme: Python'e visi duomenų tipai yra objektai. Tai suteikia importavimo ir pervadinimo semantiką. Skirtingai nuo C++ ir Modula-3, standartiniai tipai gali būti naudojami kaip bazinės klasės kai naudotojui reikia juos praplėsti. Taip pat kaip C++ bet ne kaip Modula-3, dauguma standartinių operatorių naudojant specialią sintaksę (aritmetiniai operatoriai ir t.t.) klasėse gali būti perrašyti.

Žodis apie terminologiją

Kadangi trūksta universalios terminologijos šnekant apie klases, aš kartais naudosiu Smalltalk ir C++ terminus (aš naudočiau Modula-3 terminologiją, nes jos objektiškų-orientiškumo semantika artimesnė Python'ui negu C++, bet aš nesitikiu, kad daug skaitytojų yra apie ją girdėję).

Kiekvienas objektas yra individualus, bet keletas vardų (skirtingose vietose) gali nurodyti į tą patį objektą. Kitose kalbose tai vadinama susitapatinimu. Tai dažniausiai nėra įvertinama iš pirmo žvilgsnio ir netgi

gali būti saugiai ignoruojama, kai dirbama su paprastais nekintamais tipais (skaičiais, eilutėmis ar kortežais). Tačiau susitapatinimas turi (planuotą!) efektą semantikai Python'o kode kai dirbama su kintamais objektais kaip sąrašai, žodynai ir dauguma tipų, kurie reprezentuoja objektus už programos ribų (failus, langus ir t.t.). Tai dažniausiai naudojama programos naudai, nes susitapatinimas elgiasi kaip rodyklė tam tikrais atžvilgiais. Pavyzdžiui, perduoti objektą yra pigu, nes perduodama tik nuoroda. Ir jeigu funkcija modifikuoja objektą, kuris buvo perduotas kaip argumentas, kvietėjas taip pat matys tuos pakeitimus --- dėka to mums nereikia dviejų skirtingų argumentų perdavimo mechanizmų kaip Paskalyje.

Python'o sritys ir vardų erdvės

Prieš klasių pristatymą, aš pirmiausia turiu papasakoti apie Python'o sričių taisykles. Klasių apibrėžtis naudoja kelis triukus su vardų erdvėmis ir jūs turite žinoti kaip sritys ir vardų erdvės veikia, kad visiškai suprastūmėte kas vyksta. Tuo pačiu šios žinios naudingos bet kuriam patyrusiam Python'o programuotojui.

Pradėkime nuo kelių apibrėžimų.

Vardų erdvė yra vardų į objektus atvaizdis. Dauguma vardų erdvių yra įgyvendintos naudojant Python'o žodynus, bet įprastai tai nėra matoma (išskyrus greityje) ir ateityje tai gali pasikeisti. Vardų erdvių pavyzdžiai: standartinių vardų aibė (funkcijos kaip :func:`abs`, ir standartinių išimčių vardai); globalūs vardai modulyje, lokalūs vardai funkcijoje. Tam tikra prasme objekto atributų aibė taip pat suformuoja vardų erdvę. Svarbiasias dalykas, kurį reikia suprasti apie vardų erdves, kad nėra visiškai jokio ryšio tarp vardų skirtingose vardų erdvėse. Pvz.: du skirtingi moduliai gali apibrėžti tą pačią funkciją „maksimizuoti“ --- tam, kad naudotojai nesusimaišytų, modulio naudotojai turėtų pridėti modulio vardą prieš funkciją.

Tarp kitko, aš naudoju žodį *atributas* kiekvienam vardui po taško ---pvz.: reiškinyje `z.real`, `real` yra objekto `z` atributas. Griežtai šnekant, nuorodos į vardus moduliuose yra atributų nuorodos: reiškinyje `modname.funcname`, `modname` yra modulio objektas ir `funcname` yra jo atributas. Šiuo atveju įvyksta tiesioginis atvaizdavimas tarp modulio atributo ir globalių vardų apibrėžtų modulyje: jie naudojasi ta pačia vardų erdve! [#]_

Atributas gali būti tik-skaitomi arba rašomi. Pastaruoju atveju, priskyrimas atributams yra galimas. Modulių atributai yra rašomi: jūs galite parašyti `modname.the_answer = 42`. Rašomi atributai gali būti ištrinti naudojant bazinio žodžio :keyword:`del` sakinį. Pvz.: `del modname.the_answer` pašalins atributą :attr:`the_answer` iš objekto pavadinto `modname`.

Vardų erdvės yra sukuriamos skirtingais momentais ir gyvuoja skirtingą laiko tarpą. Vardų erdvės, kuriose laikomi įtaisyti vardai, yra sukuriamos kai Python'o interpretatorius startuoja, ir niekada nėra ištrinamos. Globali vardų erdvė moduliui yra sukuriama kaip perskaitomas modulio apibrėžimas. Paprastai, modulio vardų erdvė gyvuoja kol interpretatorius baigia savo darbą. Sakiniai, kurie yra vykdomi interpretoriuje aukščiausiame lygyje (tiek skaitant iš skripto failo arba interaktyviai) yra laikomo modulio :mod:`__main__` dalimi, taigi jie turi savo vardų erdvę. Įtaisyti vardai taip pat laikomi modulyje pavadintame :mod:`__builtin__`.

Lokali vardų erdvė funkcijai yra sukuriama kai funkcija yra iškviečiama ir yra ištrinama (tiesa sakant, „pamirštama“ yra tinkamesnis žodis apibūdinti tam kas iš tikro atsitinka) kai funkcija sugrįžta arba sukelia išimtį, kuri nėra suvaldoma funkcijoje. Žinoma, kiekvienas rekursyvūs šaukimas turi savo lokalią vardų erdvę.

Sritis yra tekstinis regionas Python'o programoje, kur vardų erdvė yra tiesiogiai pasiekiama. „Tiesiogiai pasiekiama“ šiuo atveju reiškia, kad neapribotos nuorodos į vardą bando surasti vardą vardų erdvėje.

Nors sritys yra apibrėžtos statiškai, jos naudojamos dinamiškai. Bet kuriuo vykdymo momentu egzistuoja bent trys vidinės sritys, kurių vardų erdvės yra tiesiogiai pasiekiamos: vidinė sritis, kurioje ieškoma pirmiausia, saugo lokalius vardus; vardų sritys bet kuriai funkcijai, kurios yra ieškomos nuo artimiausios uždarančios srities; vėliau ieškoma vidurinėje srityje, kurioje laikomi dabartinio modulio globalūs vardai. Išorinė sritis ieškoma paskutinė ir joje laikomi įtaisyti vardai.

Jeigu vardas yra paskelbtas globaliai, tada visos nuorodos ir priskyrimai vykdomi vidurinei sričiai, kurioje yra modulio globalūs vardai. Kitu atveju, visi kintamieji esantis vidinėje srityje yra tik-skaitomi (bandymas rašyti į tokį kintamąjį paprasčiausiai sukurs *naują* lokalų kintamąjį vidinėje srityje ir nepalies taip pat pavadinto išorinio kintamojo).

Įprastai lokali sritis nurodo į lokalius vardus dabartinėje funkcijoje. Už funkcijų ribų lokali sritis nurodo tą pačią vardų erdvę kaip ir globali sritis: modulio vardų erdvę. Klasės apibrėžtis taip pat sukuria dar vieną vardų erdvę lokalioje srityje.

Svarbu suprasti, kad sritys yra nustatomos pagal tekstą: globali funkcijos sritis yra apibrėžta modulyje yra to modulio vardų erdvė, nesvarbu iš kur ar kokiu vardu ta funkcija yra kviečiama. Iš kitos pusės, vardų paieška yra atliekama dinamiškai vykdymo metu ---tačiau kalbos apibrėžimas juda link statinio vardų nustatymo „kompiliavimo metu“, taigi nepasitikėkite dinamišku vardų nustatymu! (Tiesa sakant, lokalūs kintamieji jau dabar nustatinėjami statiškai.)

Ypatinga Python'e yra tai, kad jei sakinyje nenaudojamas :keyword:`global`, tada priskyrimas vykdomas vidinėje srityje. Priskyrimai nekopijuoja duomenų --- jei tik susieja vardą su objektu. Tas pats ga-

lioja ir trynimui: sakinys `del x` pašalina `x` susiejimą lokalios srities vardų erdvėje. Tiesa sakant, visos operacijos kurios pristato naujus vardus naudoja lokalią sritį: ypač, importavimo sakiniai ir funkcijų apibrėžtys susieja modulį ar funkcijos vardą toje lokalioje srityje. (Bazinis žodis :keyword:`global` sakinyje gali būti naudojamas norint nurodyti, kad tam tikras konkretus kintamasis turi būti ieškomas globalioje srityje).

Pirmas žvilgsnis į klases

Klasės prideda šiek tiek naujos sintaksės, tris naujus objektų tipus ir šiek tiek naujos semantikos.

Klasių apibrėžties sintaksė

Paprasčiausia klasės apibrėžtis atrodo taip:[83]

[83]
klasė KlasėsVardas:
<išraiška-1>
.
.
.
<išraiška-N>

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Klasių apibrėžties, kaip ir funkcijų apibrėžties (:keyword:`def` reiškinys) turi būti įvykdyti, kad jie turėtų kokį nors efektą. (Jūs galite įdėti klasės apibrėžtį į sąlyginio sakinio :keyword:`if` šaką, arba į funkciją.)

Praktikoje, reiškiniai klasės apibrėžtyje dažniausiai bus funkcijų apibrėžtys, bet kitokie reiškiniai taip pat yra leidžiami, ir kartais tai netgi naudinga (mes dar prie to sugrįšime). Funkcijų apibrėžtys klasėje dažniausiai turi savitą argumentų sąrašo formą, kuri yra kilusi iš sutarto metodų kvietimo (tai bus paaiškinta vėliau).

Kai klasės apibrėžtis yra vykdoma yra sukuriama nauja vardų erdvė ir naudojama kaip lokali sritis --- taigi visi priskyrimai lokaliems kintamiesiems patenka į šią vardų erdvę. Ypatingai tai galioja funkcijų apibrėžtims, kurios susieja naujų funkcijų vardus.

Kai klasės apibrėžtis įvykdoma (dažniausiai pabaigoje) yra sukuriamas naujas *klasės objektas*. Tai iš esmės yra apgaubta vardų erdvė, kuri yra sukurta iš klasės apibrėžties (mes išmoksime daugiau apie klasių objektus kitame skyriuje). Originali lokali sritis (ta kuri buvo naudojama prieš įžengiant į klasės apibrėžtį) yra atstatoma ir klasės objektas yra susietas su klasės vardu duotu jai klasės apibrėžties antraštėje (pvz.: :class:`ClassName`).

Klasių objektai

Klasių objektai palaiko dviejų rušių operacijas: atributų nuorodos ir egzemplioriaus sukūrimą.

Atributų nuorodos naudoja standartinę sintaksę, kurią naudojame visoms atributų nuorodoms Python'e: `obj.name`. Validūs atributų vardai yra visi vardai, kurie buvo klasės apibrėžtyje kai klasės objektas buvo sukurtas. Taigi, jei klasės apibrėžtis atrodo taip:[84]

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

[84]
```
klasė ManoKlasė:
    """Paprastas klasės pavyzdys"""
    i = 12345
    apif f(self):
        rodo 'sveikas pasauli'
```

tada `MyClass.i` ir `MyClass.f` yra validžios atributų nuorodos, kurios grąžina sveikąjį skaičių ir funkcijos objektą, atitinkamai. Klasių atributai taip pat gali būti priskiriami, taigi `MyClass.i` reikšmė gali būti pakeista priskyrimu. :attr:`__doc__` taip pat yra validus atributas, kuris grąžina dokumentacijos eilutę, kuri priklauso klasei: `"A simple example class"`.

Klasių *egzempliorių sukūrimas* naudoja funkcijų užrašą. Tiesiog apsimeskite, kad klasės objektas yra funkcija be parametrų kuri grąžina nauja klasės egzempliorių. Pavyzdžiui (jei naudojame aukščiau apibrėžtą klasę):[85]

```
x = MyClass()
```

[85]
```
x = ManoKlasė()
```

sukurią naują klasės *egzempliorių* ir priskiria šį objektą lokaliam kintamajam `x`.

Egzemplioriaus sukūrimo operacija („kviečiant“ klasės objektą) sukuria naują objektą. Dauguma klasių sukuria objektus, kurių egzemplioriai yra pritaikomi speficiniai pradinei pozicijai. Todėl klasė gali apibrėžti specialų metodą pavadintą :meth:`__init__`, pvz.:

```
def __init__(self):
    self.data = []
```

Kai klasė apibrėžia :meth:`__init__` metodą, klasės egzemplioriaus sukūrimas automatiškai iškviečia meth:`__init__` naujai sukurtam klasės egzemplioriui. Taigi šiame pavyzdyje naujas inicijuotas egzempliorius gali būti gautas iškviečiant:[86]

```
x = MyClass()
```

[86]
```
x = ManoKlasė()
```

Žinoma metodas :meth:`__init__` gali turėti ir argumentus didesniam lankstumui. Tokiu atveju argumentai kurie paduodami klasės egzemplioriaus sukūrimo operatoriui yra perduodami :meth:`__init__`. Pavyzdžiui

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

[87]

[87]
klasė Kompleksiniai:
def __init__(self, realidalis, menamadalis):
self.r = realidalis
self.i = menamadalis
...
x = Kompleksiniai(3.0, -4.5)
x.r, x.i
(3.0, -4.5)

Egzempliorių objektai

Taigi ką mes galime daryti su egzempliorių objektais? Vienintelė operacija kurią supranta egzempliorių objektai yra atributų nuorodos. Yra dviejų rūšių tinkami atributų vardai: duomenų atribudai ir metodai.

duomenų atributai atitinka Smalltalk „egzempliorių kintamuosius“ ir „duomenų narius“ C++. Duomenų atributai turi būti aprašyti. Kaip ir lokalūs kintamieji jie pradeda egzistuoti, kai jiems pirmą kartą yra priskiriama reikšmė. Pavyzdžiui, jei `x` yra anksčiau sukurtos klases :class:`MyClass` egzempliorius, tai žemiau esantis kodas išspausdins reikšmę `16` ir nepaliks jokios žymės:[88]

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

[88]
x.skaitliukas = 1
kol x.skaitliukas < 10:
x.skaitliukas = x.skaitliukas * 2
rodo x.skaitliukas
trina x.skaitliukas

Kitas egzemplioriaus atributo nuorodos tipas yra *metodas*. Metodas yra funkcija, kuri „priklauso“ objektui. (Python'e, terminas metodas nėra unikalus klasės egzemplioriams: kiti objektų tipai gali turėti metodus taip pat. Pvz, sąrašo objektas turi metodus append, insert, remove, sort ir t.t. Tačiau toliau sekančioje diskusijoje mes naudosime terminą „metodas“ išskirtinai klasių egzempliorių objektams, nebent bus nurodyta kitaip).

Validus metodų namai egzemplioriaus objektui priklauso nuo jo klasės. Pagal apibrėžimą, visi klasės atributai, kurie yra funkcijų objektai, apibrėžia atitinkamus egzemplioriaus metodus. Pavyzdžiui, `x.f` yra validus metodas, nes `MyClass.f` yra funkcija, bet `x.i` nėra, nes `MyClass.i` nėra. Bet `x.f` nėra tas pats kas `MyClass.f` --- tai yra *metodo objektas*, ne funkcijos objektas.

Metodų objektai

Įprastai, metodai yra iškviečiamas iškart po to, kai jis susiejamas:

```
x.f()
```

Klasės :class:`MyClass` pavyzdyje, tai sugrąžins eilutę `'hello world'`. Tačiau, nebūtina kviesti metodą iškart: `x.f` yra metodo objektas ir gali būti saugomas vėliasniam laikui, Pvz.:[89]

```
xf = x.f
while True:
    print xf()
```

Spausdins `hello world` begale kartų.

Kas konkrečiai atsitinka, kai metodas yra iškviečiamas? Jūs galėjote pastebėti, kad `x.f()` (viršuje) buvo iškviestas be argumentų, nors apibrėžtis metodui :meth:`f` argumentą ir nurodė. Kas atsitiko argumentui? Žinoma Pythonas sukelia išimtį kai funkcija, kuriai reikia argumentų yra iškviečiama be jų --- netgi jei argumentas nėra naudojamas...

Tiesa sakant, atsakymą jūs galbūt jau nuspėjote: metodų ypatybė yra ta, kad objektas yra perduodamas kaip pirmas argumentas funkcijai. Mūsų pavyzdyje, kvietimas `x.f()` yra ekvivalentus `MyClass.f(x)`. Apibendrinus, metodo su *n* argumentų sąrašu kvietimas yra ekvivalentiškas atitinkamos funkcijos atitinkamos funkcijos kvietimui su argumentu sąrašu, kuris yra sukuriamas pridedant metodo objektą prieš pirmą argumentą.

Jei jūs vis dar nesuprantate kaip dirba metodas, žvilgsnis į įgyvendinimą gali viską paaiškinti. Pirmiausia klasėje ieškoma egzemplioriaus atributo, kuris nėra duomenų atributas. Jeigu vardas nurodo validų klasės atributą, kuris yra funkcijos objektas, metodo objektas yra sukuriamas supakuojant kartu egzemplioriaus objektą ir ką tik surastos funkcijos objektą į abstraktų objektą ---tai ir yra metodo objektas. Kai metodo objektas yra iškviečiamas su argumentų sąrašu, jis išpakuojamas vėl, naujas argumentų sąrašas yra sukonstruojamas iš egzemplioriaus objekto ir originalaus argumentų sąrašo ir tada funkcijos objektas yra iškviečiamas naudojant šį naują argumentų sąrašą.

Atsitiktinės pastabos

Duomenų atributai yra svarbesni už metodų atributus tokiu pačiu vardu. Tam kad išvengtumėte vardų konflikto, dėl ko gali kilti sunkiai randamos klaidos didelėse programose, patartina naudoti tam tikrus susitarimus, kad konfliktų tikimybė būtų minimizuota. Galimi susitarimai gali būti metodų vardų rašymas iš didžiųjų raidžių, pridedant tam tikra unikalią eilutę prie duomenų atributų pradžioje (pvz.: pabraukimą) arba metodų vardams naudoti veiksmažodžius, o daiktavardžius naudoti duomenų atributams.

[89] xf = x.f
kol True:
rodo xf()

Duomenų atributai gali būti pasiekiami tiek metodų tiek paprastų objekto naudotojų („klientų“). Kitaip sakant, klasės nėra tinkamos įgyvendinti abstrakčius duomenų tipus. Tiesa sakant, Python'e nėra nieko kas leistų paslėpti duomenis --- viskas yra paremta susitarimu. (Iš kitos pusės, Python'o implementacija parašyta C gali visiškai paslėpti įgyvendinimo detales ir kontroliuoti priėjimą prie objekto, jei tik to reikia. Tai gali būti padaryta naudojant Python'o išplėtimus parašytus C.)

Klientai turi naudoti duomenų atributus atsargiai --- klientai gali sujaukti metodų prižiūrimus invariantus jei bus pakeisti jų duomenų atributai. Pastebėsime, kad klientai gali pridėti savo duomenų atributus į egzemplioriaus objektą nepaveikiant metodų tinkamumo, tol kol išvengiama vardų konflikto --- vėlgi vardų kūrimo susitarimai gali padėti išvengti daug problemų.

Python'e nėra sutrumpinimo norint pasiekti duomenų atributus (ar kitus metodus!) iš metodo. Aš asmeniškai manau, kad tai padidina metodų skaitomumą: nėra galimybės sumaišyti lokalių ir egzemplioriaus kintamųjų besižvalgant metode.

Dažniausiai, pirmas metodo argumentas yra vadinamas `self`. Tai tėra nieko daugiau tik susitarimas: vardas `self` neturi jokios specialios reikšmės Python'e. (Tačiau pastebėkite, kad jums nesilaikant šio susitarimo jūsų kodas gali būti mažiau skaitomas kitiems Python'o programuotojams, ir labai tikėtina, kad *klasių naršyklės* programa gali būti parašyta laikantis šio susitarimo).

Kiekvienas funkcijos objektas, kuris yra klasės atributas, apibrėžia metodą šios klasės egzemplioriams. Nėra būtina, kad funkcijos apibrėžtis būtų apgaubta klasės apibrėžties tekste: funkcijos priskyrimas lokaliam klasės kintamajam taip pat yra galimas. Pvz.:[90]

```
# Funkcija apibrėžta už klasės ribų
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

[90]
```
# Funkcija apibrėžta už klasės ribų
apif f1(self, x, y):
   grąžina min(x, x+y)

klasė C:
   f = f1
   apif g(self):
      grąžina 'sveikas pasauli'
   h = g
```

Dabar `f`, `g` and `h` yra klasės :class:`C` atributai, kurie nurodo į funkcijų objektus yra to pasekoje jie visi yra klasės :class:`C` egzemplioriaus metodai --- `h` yra ekvivalentus `g`. Pastebėsime, kad tokia praktika dažniausiai naudojama norint tik sumaišyti programos skaitytoją.

Metodai gali kviesti kitus metodus naudodami argumento `self` metodo atributus:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

[91] Metodai gali kreiptis į globalius vardus tokiu pačiu būdų kaip paprastos funkcijos. Globali sritis susieta su metodu yra modulis kuriame yra klasės apibrėžtis (klasė pati savaime niekada nėra naudojama globalioje srityje!). Nors surasti gerą priežastį naudoti globalius duomenis metode yra labai sunku, yra daug teisėtų būdų naudoti globalią sritį: metodai gali naudoti funkcijas ar modulius importuotus į globalią sritį, taip pat metodai gali naudoti funkcijas ar klases apibrėžtas globalioje srityje. Dažniausiai, klasė kurioje yra metodai yra taip pat apibrėžta globalioje srityje, ir kitoje dalyje mes surasime keletą gerų priežasčių kodėl metode gali prireikti nurodyti savo paties klasę!

Kiekviena reikšmė yra objektas, ir to pasekoje turi *klasę* (taip pat vadinama *tipu*). Ji laikoma `object.__class__`.

[91]
klasė Kuprinė:
  apif __init__(self):
    self.data = []
  apif prideda(self, x):
    self.data.append(x)
  apif pridedadukart(self, x):
    self.prideda(x)
    self.prideda(x)

Paveldėjimas

Žinoma, kalbos savybė nebūtų verta „klasės“ vardo jeigu nepalaikytų paveldėjimo. Paveldėtos klasės apibrėžties sintaksė atrodo taip:[92]

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

[92]
klasė IšvestinėsKlasėsVardas(BazinėsKlasėsVardas):
  <išraiška-1>
  .
  .
  .
  <išraiška-N>

Vardas :class:`BaseClassName` turi būti apibrėžtas srityje, kur yra laikoma paveldėtos klasės apibrėžtis. Viršklasio vardo vietoje, kitos norimos išraiškos taip pat yra leidžiamos. Tai tarkim gali būti naudinga kai viršklasis yra apibrėžtas kitame modulyje:[93]

```
class DerivedClassName(modname.BaseClassName):
```

[93] klasė IšvestinėsKlasėsVardas(moduliovardas.BazinėsKlasėsVardas):

Paveldėtos klasės apibrėžties vykdymas vyksta taip pat kaip ir viršklasiui. Kai klasės objektas yra konstruojamas, taip pat yra prisimenamas ir viršklasis. Tai naudojama nustatant atributų nuorodas: jeigu norimas atributas nėra randamas klasėje, tada jo paieška vykdoma jos

viršklasyje. Ši taisyklė yra taikoma rekursiškai jeigu pats viršklasis yra paveldėtas iš kokios nors kitos klasės.

Taip pat nevyksta nieko ypatingo kuriant paveldėtos klasės egzempliorių: `DerivedClassName()` sukuria naują klasės egzempliorių. Metodų nuorodos surandamos taip: pirmiausia ieškoma atitinkamos klasės atributo ir jeigu reikia tada ieškoma grandine žemyn per viršklasius, ir metodo nuoroda yra validi jei ji gražina funkcijos objektą.

Paveldėtos klasės gali perrašyti viršklasio metodus. Kadangi metodai neturi specialių teisių kviečiant kitus to pačio objekto metodus, viršklasio metodas, kuris kviečia kitą metodą apibrėžta tame pačiame viršklasyje, galiausiai gali iškviesti paveldėtos klasės metodą (kuris perrašė norimą iškviesti metodą). (C++ programuotojams: visi metodai Python'e yra `virtualūs`.)

Metodo perrašymas paveldėtoje klasėje gali praplėsti (užuot tiesiog pakeitus) viršklasio metodą tuo pačiu vardu. Bazinės klasės metodą iškviesti tiesiogiai yra labai paprasta: tiesiog iškvieskite `BaseClassName.methodname(self, arguments)`. Tai kartais naudinga ir naudotojams. (Pastebėkite, kad tai veikia tik tada, kai bazinė klasė yra apibrėžta arba importuota tiesiogiai į globalią sritį).

Python'as turi dvi įtaisytas funkcijas, kurios dirba su paveldėjimu:

- Naudokite funkciją :func:`isinstance` norėdami patikrinti objekto tipą: `isinstance(obj, int)` bus `True` tik tada kai `obj.__class__` yra :class:`int` arba kokia nors klasė paveldėta iš :class:`int`.
- Naudokite funkciją :func:`issubclass` norėdami patikrinti paveldėjimą: `issubclass(bool, int)` yra `True` kadangi :class:`bool` yra klasės :class:`int` poklasis. Tačiau, `issubclass(unicode, str)` grąžins `False` kadangi :class:`unicode` nėra :class:`str` poklasis (jie tik turi bendrą protėvį :class:`basestring`).

Sudėtinis paveldėjimas

Python'as palaiko apribotą sudėtinio paveldėjimo formą. Klasės apibrėžtis su keliais viršklasiais atrodo taip:[94]

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

[94] klasė IšvestinėsKlasėsVardas(Bazinė1, Bazinė2, Bazinė3):
    <išraiška-1>
    .
    .
    .
    <išraiška-N>

Seno stiliaus klasėms, vienintelė taisyklė yra pirmiausia-gilyn, iš kairės į dešinę. Taigi jei atributas nėra randamas klasėje :class:`DerivedClassName`, tada jo ieškoma klasėje :class:`Base1`, tada (rekursyviai) klasės

:class:`Base1` viršklasyje ir tik tada, jei jo neranda ten, ieškoma klasėje :class:`Base2` ir t.t.

(Kai kuriems žmonėms ieškojimas klasėse :class:`Base2` ir :class:`Base3` prieš ieškant klasės :class:`Base1` viršklasiuose atrodo natūralus. Tačiau tai reikalautų žinoti ar tam tikras :class:`Base1` atributas yra iš ties apibrėžtas klasėje :class:`Base1` arba viename iš jos viršklasių prieš galėdamas numatyti vardų konflikto pasekmes su klasės :class:`Base2` atributu. Pirmiausia-gilyn taisyklė neskiria tiesioginių ir paveldėtų klasės :class:`Base1` atributų.

:term:`naujo stiliaus klasėse <naujo stiliaus klasė>` metodų nustatymo tvarka keičiasi dinamiškai, kad galėtų palaikyti bendradarbiaujančius funkcijos :func:`super` kvietimus. Šis sprendimas keliose kitose sudėtinį paveldėjimą palaikančiose kalbose yra žinomas kaip kviesk-kitą-metodą ir yra galingesnis negu super kvietimas randamas paveldėjimo iš vieno viršklasio kalbose.

Naujo stiliaus klasėms dinamiškas paveldėjimas yra reikalingas nes visi sudėtinio paveldėjimo atvejai naudoja vieną ar daugiau deimantinių sąsajų (kai bent viena iš tėvinių klasių gali būti pasiektos keletu kelių iš tos pačios viršutinės klasės). Pvz.: visos naujo stiliaus klasės yra paveldėtos iš :class:`object`, taigi bet kuris sudėtinio paveldėjimo atvejis leidžia daugiau negu vieną būdą pasiekti :class:`object`. Norint viršklasius apsaugoti nuo panaudojimo kelis kartus, dinaminis algoritmas liniarizuoja paieškos tvarką taip, kad būtų išlaikyta iš kairės į dešinę tvarka nurodyta kiekvienoje klasėje, kuri kviečią tėvą tik kartą, ir būtų monotoniška (monotoniška reiškia, kad klasė gali būti poklasiu nepaveikdama jos tėvų pirmenybės tvarkos). Visą tai paimant kartu, šios savybės leidžia kurti patikimas ir išplečiamas klases naudojant sudėtinį paveldėjimą. Daugiau informacijos čia: http://www.python.org/download/releases/2.3/mro/.

Privatūs kintamieji

Privatūs klasės identifikatoriai egzistuoja, bet su tam tikrais apribojamais. Jei identifikatorius turi formą `__spam` (bent du pabraukimo brūkšniai priekyje ir daugiausiai vienas pabaigoje) jis yra pakeičiamas į `_classname__spam`, kur `classname` yra klasės vardas be pabraukimo brūkšnių, jei jie buvo klasės vardo gale. Šis pakeitimas atliekamas nepaisant sintaksinės identifikatoriaus pozicijos, todėl tai gali būti naudojama apibrėžti klasės kintamuosius, metodus, kintamuosius laikomus tarp globalių kintamųjų ir netgi kintamuosius laikomus egzemplioriuose, kurie yra privatūs šiai klasei *kitos* klasės egzemplioriuje. Gali įvykti sutrumpinimas, jei pakeistas vardas yra ilgesnis negu 255 simboliai. Jei identifikatorius nėra klasėje arba kai klasės vardas sudarytas tik iš pabraukimo brūkšnių pakeitimas neįvyksta.

Vardų pakeitimo tikslas yra leisti paprastai apibrėžti „privačius“ egzemplioriaus kintamuosius ir metodus nesijaudinant apie egzemplioriaus kintamuosius apibrėžtus paveldėtoje klasėje arba egzemplioriaus kintamųjų keitimus už klasės ribų. Pastebėsime, kad pakeitimo taisyklės yra sukurtos taip, kad būtų išvengta sutapimų. Tačiau visgi įmanoma pasišventusiai sielai pasiekti ir modifikuoti kintamuosius, kurie laikomi privačiais. Tai netgi gali būti naudinga tam tikrose situacijose, kaip kad derintuvėje ir tai vienintelė priežastis kodėl ši skylė yra palikta. Paveldėjimas naudojant tokį patį klasės vardą kaip viršklasio leidžia naudoti viršklasio privačius kintamuosius.

Likučiai

Kartais yra naudinga turėti duomenų tipą panašų į Paskalio „record“ arba C „struct“, kuriame būtų laikomi keli vardiniai duomenų nariai. Tuščia klasės apibrėžtis tam puikiai tinka:[95]

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

[95]
```
klasė Samdiniai:
   pass

# Kuriama tuščias samdinio įrašas
jonas = Samdiniai()

# Užpildomos įrašo laukų reikšmės
jonas.vardas = 'Jonas Doje'
jonas.skyrius = 'kompiuterių lab'
jonas.uždarbis = 1000
```

Python'o kodo gabaliukui, kuris tikisi abstraktaus tipo, galima perduoti klasę, kuri emuliuoja to duomenų tipo metodus. Tarkime, jei jūs turite funkcija, kuri formuoja kažkokį duomenų tipą iš failo objekto, jūs galite apibrėžti klasę su metodais :meth:`read` ir :meth:`readline` kurie gauna duomenis iš eilutės buferio ir perduoti tą buferį kaip argumentą.

Egzemplioriaus metodų objektai turi atributus taip pat: `m.im_self` yra egzemplioriaus objektas su metodu :meth:`m`, ir `m.im_func` yra funkcijos objektas atitinkantis tą metodą.

Išimtys yra klasės taip pat

Naudotojų apibrėžtos išimtys yra identifikuojamos pagal klases taip pat. Naudojant šį mechanizmą galima sukurti praplečiamas išimčių hierarchijas.

Egzistuoja dvi naujos tinkamos (semantinės) formos išimčių sukėlimui:

```
išveda Class, narys
```

```
išveda narys
```

Pirmoje formoje `instance` privalo būti klasės :class:`Class` egzempliorius arba paveldėta klasė iš jo. Antroji forma yra šio reiškinio sutrumpinimas:[96]

```
raise instance.__class__, instance
```

Klasė except reiškinyje yra suderinama su išimtimi jei ji yra tos pačios klasės ar viršklasio (bet ne atvirkščiai --- jei except reiškinyje yra poklasis, tai jis nėra suderinamas su viršklasiu). Pavyzdžiui, šis kodas atspausdins B, C, D iš eilės:[97]

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Jeigu except reiškiniai būtų surašyti atvirkštine tvarka (su `except B` pirmiausia), tai bus atspausdinta B, B, B --- vykdomas pirmas sutampantis except reiškinys.

Kai spausdinamas klaidos pranešimas nesutvarkytai išimčiai, išimties klasės vardas yra atspausdinamas, tada dvitaškis ir tarpas, o galiausiai egzempliorius paverstas eilute naudojant įtaisyta funkciją :func:`str`.

Iteratoriai

Iki dabar jūs tikriausiai pastebėjote, kad dauguma konteinerių objektų gali būti pereiti naudojant :keyword:`for` reiškinį:[98]

```
for element in [1, 2, 3]:
    print element
```

[96]
```
išveda narys.__class__, narys
```

[97]
```
klasė B:
    pass
klasė C(B):
    pass
klasė D(C):
    pass

kožnam c iš [B, C, D]:
    bandomo:
        išveda c()
    išimtis D:
        rodo "D"
    išimtis C:
        rodo "C"
    išimtis B:
        rodo "B"
```

[98]
```
kožnam elementas iš [1, 2, 3]:
    rodo elementas
kožnam elementas iš (1, 2, 3):
    rodo elementas
kožnam raktas iš 'vienas':1, 'du':2:
    rodo raktas
kožnam simbolis iš "123":
    rodo simbolis
kožnam eilutė iš
open("manofailas.txt"):
    rodo eilutė
```

```
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

Šis priėjimo būdas yra aiškus, trumpas ir patogus. Iteratorių naudojimas apima ir suvienodina Python'ą. Už scenos, bazinio žodžio :keyword:`for` reiškinys iškviečia :func:`iter` konteinerio objektui. Funkcija sugrąžina iteratoriaus objektą kuris apibrėžią metodą :meth:`next`, kuris pasiekia konteinerio elementus po vieną vienu metu. Kai daugiau elementų nėra, metodas :meth:`next` sukelia išimtį :exc:`StopIteration` kuri pasako baziniam žodžiui :keyword:`for` pabaigtį ciklą. Šis pavyzdys rodo kaip visa tai veikia:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Pamačius kaip veikia iteratoriaus protokolas, yra labai paprasta pridėti iteratoriaus elgesį į jūsų klases. Apibrėžkite metodą :meth:`__iter__`, kuris sugrąžina objektą su metodu :meth:`next`. Jei klasė apibrėžia metodą :meth:`next`, tada :meth:`__iter__` gali tiesiog sugrąžinti `self`:[99]

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
```

[99] klasė Reverse:
"iteratorius perbėgti nariams atbulai"
def __init__(self, data):
self.data = data
self.index = len(data)
def __iter__(self):
grąžina self
def toliau(self):
jei self.index == 0:
išveda StopIteration
self.index = self.index - 1
grąžina self.data[self.index]

```
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
...     print char
...
m
a
p
s
```

Generatoriai

:term:`Generatoriai <generatorius>` yra paprastas ir galingas įrankis iteratorių kūrimui. Jie aprašomi kaip paprasčiausios funkcijos, bet naudojamas bazinio žodžio :keyword:`yield` reiškinys kiekvienąkart kai jiems reikia grąžinti duomenis. Kaskart kai iškviečiamas metodas :meth:`next`, generatorius sugrįžta į tą vietą, kur jis sustojo (jis prisimena visas duomenų reikšmes ir koks reiškinys buvo paskutinį kartą vykdomas). Pavyzdys parodo, kad generatorius sukurti yra visiškai paprasta:[100]

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

Viską ką galima padaryti su generatoriais galima padaryti su klasėmis paremtais iteratoriais kaip aprašyta praeitoje dalyje. Generatorių forma yra tokia kompaktiška dėl to, kad metodai :meth:`__iter__` ir :meth:`next` yra sukuriami automatiškai.

Kita esminė savybė yra ta, kad lokalūs kintamieji ir jų vykdymo būsena yra automatiškai išsaugoma tarp kvietimų. Tai leidžia daug

[100]
```
apif apversti(duomenis):
    kožnam indeksas iš
range(len(duomenys)-1, -1, -1):
  yield data[indeksas]

  kožnam simbolis iš apversti('golf'):
    rodo simbolis
  ...
```

lengviau rašyti funkcijas ir jos yra daug aiškesnės negu naudojant egzemplioriaus kintamuosius kaip `self.index` ir `self.data`.

Prie viso to, kad automatiškai yra sukuriami metodai ir išsaugoma programos būsena, generatoriai automatiškai iškelia :exc:`StopIteration`, kai jie baigia darbą. Visos kartu šios savybės leidžia paprastai sukurti iteratorius nevargstant daugiau negu rašant paprastą funkciją.

Generatorių reiškiniai

Kai kurie paprasti generatoriai gali būti parašyti kaip reiškiniai naudojant sintaksę, kuri panaši į sąrašo užklausą, bet vietoje laužtinių skliaustelių naudojant paprastus skliaustelius. Šie reiškiniai yra skirti situacijoms kai generatoriai naudojami iškarto uždarančioje funkcijoje. Generatorių reiškiniai yra kompaktiškesni negu pilnos generatorių apibrėžtys ir linkę labiau taupyti atmintį negu atitinkamos sąrašo užklausos.

Pavyzdžiai:[101]

```
>>> sum(i*i for i in range(10))                 # kvadratų suma
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # skaliarinė sandauga
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

[101]

```
# kvadratų suma
sum(i*i kožnam i iš range(10))
285

xvec = [10, 20, 30]
yvec = [7, 5, 3]
# skaliarinė sandauga
sum(x*y kožnam x,y iš zip(xvec, yvec))
260

Iš math naudoja pi, sin
sine_lentelė = dict((x, sin(x*pi/180))
kožnam x iš range(0, 91))

unikalūs_žodžiai = set(žodis kožnam
eilutė iš puslapis kožnam žodis iš
eilutė.split())

valedictorian = max((studentai.gpa,
studentai.vardas) kožnam studentai iš
grupės)

data = 'golf'
list(data[i] kožnam i iš range(len(data)-
1,-1,-1))
```

Trumpa standartinės bibliotekos apžvalga

Operacinės sistemos sąsaja

:mod:`os` modulis suteikia galimybę bendrauti su operacine sistema:[102]

[102] naudoja os

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()      # grąžina dabartinę darbinę direktoriją
'C:\\Python26'
>>> os.chdir('/server/accesslogs')
```

Būtinai naudokite `import os` stilių vietoje `from os import *`. Tai neleis funkcijai :func:`os.open` paslėpti įtaisytos funkcijos :func:`open`, kuri dirba visai kitaip.

Įtaisytos funkcijos :func:`dir` ir :func:`help` yra naudingos interaktyviai dirbant su dideliais moduliais kaip :mod:`os`:[103]

[103] naudoja os

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Kasdieniam darbui su failais ir direktorijomis modulis :mod:`shutil` suteikia aukštesnio lygio sąsają, kurią lengviau naudoti:[104]

[104] naudoja shutil

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

Failų paieška

:mod:`glob` modulyje galima rasti funkciją, kuri leidžią į sąrašą iš direktorijos surinkti norimus failus nurodant tų failų vardo požymius:[105]

[105] naudoja glob

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

Komandinės eilutės argumentai

Pagalbinei skriptai dažnai turi dirbti su komandinės eilutės argumentais. Šiek argumentai laikomi modulio :mod:`sys` atributų sąraše *argv*. Pavyzdžiui jei įvykdysite komandą `python demo.py one two three` komandinėje eilutėje gausite:[106]

[106] naudoja sys

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

:mod:`getopt` modulis sutvarko *sys.argv* naudodamas Unix'ės :func:`getopt` funkcijos susitarimą. Daugiau galimybių ir lankstumo dirbant su komandine eilute gali suteikti :mod:`optparse` modulis.

Klaidų išvesties nukreipimas ir programos pabaigimas

:mod:`sys` modulyje taip pat galima rasti atributus *stdin*, *stdout*, and *stderr*. Pastarasis yra naudingas norint parodyti klaidų pranešimus netgi tada kai *stdout* (standartinė išvestis) yra nukreipta (t.y. nematoma naudotojui):

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

Pats tiesiausias būdas pabaigti skripto veikimą yra naudoti `sys.exit()`.

Eilučių rašto atpažinimas

:mod:`re` modulis suteikia galimybę naudotis reguliariais reiškiniais (angl. regular expression) sudėtingesniam eilučių tvarkymui. Sudėtingam rašto atpažinimui ir manipuliacijoms, reguliarūs reiškiniai siūlo trumpą ir optimizuotą sprendimą:[107]

[107] naudoja re

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Jeigu reikia paprasto sprendimo, rekomenduojama naudoti eilučių metodus, kuriuos paprasčiau skaityti ir derinti:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

Vertėjo pastaba: iš savo patirties galiu pasakyti, kad reguliarių reiškinių reikia vengti. Yra toks posakis: žmogus turėjo problemą, pagalvojo, kad reguliarūs reiškiniai padės ją išspręsti --- dabar jis turi dvi problemas. Yra priemonių, kurios teksto analizei tinka daug labiau (pvz.: pyparsing). Vėlgi universalaus atsakymo, ką geriausiai naudoti nėra, nes jeigu jums reikia kažko veikiančio tikrai greitai reguliarūs reiškiniai gali būti ir geriausias pasirinkimas.

Matematika

:mod:`math` modulis leidžia prieiti prie C bibliotekos slankaus kablelio funkcijų:[108]

[108] naudoja math

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

:mod:`random` modulis leidžia dirbti su atsitiktiniu pasirinkimu:[109]

[109] naudoja random

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   # atsitiktinis parinkimas
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # atsitiktinis realusis skaičius
0.17970987693706186
>>> random.randrange(6)    # atsitiktinis sveikas skaičius iš pasirinktos atkarpos
4
```

Interneto prieiga

Egzistuoja ne vienas modulis priėjimui prie interneto ir interneto protokolų valdymui. Du paprasčiausi yra :mod:`urllib2` duomenų parsisiuntimui pagal nurodytą URL ir :mod:`smtplib` laiškų siuntimui:[110]

[110] naudoja urllib2
kožnam eilutė iš urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
rodo rytų laiką
jei 'EST' iš eilutė arba 'EDT' iš eilutė:
rodo eilutė

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print line
```

```
<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Pastaba: antrasis pavyzdys reikalauja lokalaus pašto serverio)

Datos ir laikai

:mod:`datetime` modulyje rasite klasių datos ir laiko manipuliavimui. Nors datos ir laiko aritmetines operacijos yra palaikomos, pagrindinis tikslas yra efektyvus informacijos ištraukimas duomenų formatavimui ir išvedimui. Modulis taip pat palaiko objektus, kurie supranta laiko zonas[111]

[111] Iš datetime naudoja date
šiandien = date.today()
šiandien

```
# dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

# dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

Duomenų suspaudimas

Dažni duomenų archyvavimo ir suspaudimo formatai yra tiesiog palaikomi modulių, pvz.: :mod:`zlib`, :mod:`gzip`, :mod:`bz2`, :mod:`zipfile` ir :mod:`tarfile`.[112]

[112] naudoja zlib

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
```

```
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

Našumo matavimas

Kai kurie Python naudotojai nori žinoti kaip skirtingi tos pačios problemos sprendimai paveikia našumą. Python'e yra priemonių matavimui, kurios šiuos klausimus atsako iškart.

Pavyzdžiui, argumentų sukeitimui galime norėti naudoti kortežų išpakavimo ir supakavimo savybes vietoje tradicinio būdo sukeisti argumentus vietomis. :mod:`timeit` modulis greitai parodys mažą našumo privalumą:[113]

[113] Iš timeit naudoja Timer

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

Jeigu :mod:`timeit` modulis yra labai tikslus matuojant mažas operacijas, :mod:`profile` ir :mod:`pstats` moduliai gali būti naudojami dideliuose kodo blokuose norint identifikuoti kritines sekcijas.

Kokybės kontrolė

Vienas būdas rašyti aukštos kokybės programinę įrangą yra testų rašymas kiekvienas funkcijai, kai ji yra kuriama, ir jų dažnas vykdymas kūrimo procese.

:mod:`doctest` modulis suteikia priemones modulių peržiūrai ir testų, kurie surašyti į dokumentacijos eilutes, tikrinimui. Testavimo konstrukcija yra paprasta: tereikia tipinį kvietimą ir jo rezultatus iš python interpretatoriaus perkelti į dokumentacijos eilutę. Taip dokumentacijoje naudotojui pateikiami pavyzdžiai ir jie leidžia doctest moduliui užtikrinti, kad kodas sutampa su dokumentacija:

```
apif vidurkis(reikšmės):
    """Computes the arithmetic mean of a list of numbers.

    >>> rodo vidurkis([20, 30, 70])
```

```
    40.0
    """
    grąžina sum(reikšmės, 0.0) / len(reikšmės)

naudoja doctest
doctest.testmod()   # automatiškai patikrina vidinius testus
```

:mod:`unittest` modulio naudojimas nėra toks paprastas kaip :mod:`doctest` modulio, bet jis leidžia rašyti detalesnius testus, kuriuos galima laikyti atskirame faile:

```
naudoja unittest

klasė TestuojaStatististinesFunkcijas(unittest.TestCase):

    apif testuoja_vidurkį(self):
        self.assertEqual(vidurkis([20, 30, 70]), 40.0)
        self.assertEqual(round(vidurkis([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, vidurkis, [])
        self.assertRaises(TypeError, vidurkis, 20, 30, 70)

unittest.main() # Kviečiama iš komandinės eilutės visiems testams atlikti
```

Baterijos pridedamos

Python'as laikosi „baterijos pridedamos“ filosofijos. Tai geriausiai matoma žiūrint į rafinuotas ir užtikrintas didesnių pakuočių galimybes. Pavyzdžiui:

- :mod:`xmlrpclib` ir :mod:`SimpleXMLRPCServer` moduliai nuotolinį procedūrų kvietimą padaro trivialia užduotimi. Nepaisant modulių vardų, jums nereikia nieko žinoti apie XML ar su juo dirbti tiesiogiai.
- :mod:`email` pakuotė yra biblioteka darbui su el. laiškų pranešimais, įskaitant ir MIME ir kitus RFC 2822-paremtus žinučių dokumentus. Skirtingai nuo :mod:`smtplib` ir :mod:`poplib`, kurie iš tikro gauna ir siunčia pranešimus, ši pakuotė turi pilną įrankinę sudėtingų pranešimų struktūrų (taip pat priedų) kūrimui ir iškodavimui, taip pat internetiniam kodavimui ir antraščių protokolų valdymui.
- :mod:`xml.dom` ir :mod:`xml.sax` pakuotės pilnai palaiko šį populiarų duomenų pasikeitimo formatą. :mod:`csv` modulis palaiko dažno duombazių formato skaitymą ir rašymą. Kartu, šie moduliai ir pakuotės labai supaprastina duomenų pasikeitimą tarp python'o programų ir kitų priemonių.

- Internacionalizacija yra palaikoma naudojant ne vieną modulį tarp kurių yra :mod:`gettext`, :mod:`locale` ir the :mod:`codecs` pakuotė.

Trumpa standartinės bibliotekos apžvalga -- antra dalis

Ši dalis apžvelgia sudėtingesnius modulius, kurie reikalingi profesionaliam programavimui. Šie moduliai retai naudojami mažuose skriptuose.

Išvesties formatavimas

:mod:`repr` modulyje rasite :func:`repr` funkcijos versiją skirtą dideliems arba giliai įdėtiems konteinerių objektams:[114]

[114] naudoja repr

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

:mod:`pprint` modulis leidžia rafinuočiau atspausdinti tiek įtaisytus tiek naudotojo apibrėžtus objektus interpretatoriui skaitomu būdu. Jeigu rezultatas yra ilgesnis negu viena eilutė, „pretty printer“ prideda eilučių pabaigos simbolius ir pastumia tekstą, kad duomenų struktūra būtų suprantamesnė:[115]

[115] naudoja pprint

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

:mod:`textwrap` modulis suformatuoja teksto paragrafą, kad jis tilptų į nurodytą ekrano plotį:[116]

[116] naudoja textwrap
// ...
rodo textwrap.fill(doc, width=40)

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

:mod:`locale` modulis prieina prie nuo kultūros priklausančių datos formatų duombazės. Lokalės formatavimo funkcijos grupavimo atributas leidžia tiesiogiai formatuoti skaičius naudojant grupių atskyrimą:[117]

[117] naudoja locale

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format("%s%.*f", (conv['currency_symbol'],
...               conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

Šablonai

:mod:`string` modulyje rasite visapusišką :class:`Template` klasę, kuri leis naudoti supaprastintą sintaksę, kurią gali suprasti ir paprastas naudotojas. Tai leidžia naudotojui modifikuoti programos elgesį nekeičiant pačios programos.

Rezervuotų vietų vardai formuojami su `$` ir teisingu Python'o identifikatoriumi (t.y. žodis, kurį gali sudaryti skaičiai, raidės ir pabraukimo brūkšniai). Jeigu rezervuotą vietą apskliaudus figūriniais skliausteliais, tai po to gali sekti kiti simboliai be tarpų. Parašius `$$` jis paverčiamas vienu `$`:[118]

[118] Iš string naudoja Template

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

:meth:`substitute` metodas pakelia :exc:`KeyError`, jeigu rezervuotai vietai nepateikiamas argumentas. Jeigu mums reikia, kad funkcija

veiktų net tuo atveju, kai naudotojas pateikia ne visą informaciją, galima naudoti :meth:`safe_substitute` metodą, jeigu ji tinkama. Naudojant šia funkciją rezervuota vieta bus nepakeista, jei duomenų truks:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Šablonų poklasiai gali apibrėžti kitokį rezervuotos vietos atpažinimo ženklą. Pavyzdžiui nuotraukų pervadinimo programėlė gali naudoti procento ženklą rezervuotai vietai atpažinti (pvz. datai, paveikslėlio numeris ar failo formatas):

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f
```

119

```
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '{0} --> {1}'.format(filename, newname)

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Kitas šablonų naudojimo tikslas yra programos logikos ir skirtingų formatų išvedimo atskyrimas. Tai leidžia rezultatus išvesti kaip XML failą, paprastą tekstinę ar HTML ataskaitą.

Darbas su dvejetainiais duomenų įrašais

:mod:`struct` modulio funkcijos :func:`pack` ir :func:`unpack` leidžia dirbti su kintamo ilgio dvejetainiais įrašų formatais. Sekantis pavyzdys

119

```
  naudoja time, os.path
  nuotraukos = ['img_1074.jpg',
'img_1076.jpg', 'img_1077.jpg']
  klasė Pervadintojas(Template):
      skirtukas = '%'
  fmt = raw_input('Įveskite pervadinimo
stilių (%d-data %n-eilesnr %f-formatas):
')
  t = Pervadintojas(fmt)
  data = time.strftime('%d%b%y')
  kožnam i, failovardas iš enumera-
te(nuotraukos):
      bazė, plėt =
os.path.splitext(filovardas)
      naujasvardas = t.substitute(d=data,
n=i, f=plėt)
      rodo '0 --> 1'.format(failovardas,
naujasvardas)
```

parodo kaip perskaityti ZIP failo antraštę nenaudojant :mod:`zipfile` modulio. Pakavimo kodai `"H"` ir `"I"` reprezentuoja dviejų ir keturių baitų skaitinę reikšmę be ženklo atitinkamai. `"<"` reiškia, kad jie yra įprasto dydžio ir išsidėstę didėjančių baitų tvarka:

```
naudoja struct

duomenys = open('myfile.zip', 'rb').read()
startas = 0
kožnam i iš range(3):                      # parodo pirmas 3 failo antraštes
    startas += 14
    laukai = struct.unpack('<IIIHH', duomenys[start:start+16])
    crc32, comp_dydis, uncomp_dydis, failovardodydis, extra_dydis = laukai

    startas += 16
    failovardas = duomenys[startas:startas+failovardodydis]
    startas += failovardodydis
    extra = duomenys[startas:startas+extra_dydis]
    rodo failovardas, hex(crc32), comp_dydis, uncomp_dydis

    startas += extra_dydis + comp_dydis     # peršoka prie sekančios antraštės
```

Daugiagijiškumas

Gijos naudojamos norint atskirti užduotis, kurios nėra viena nuo kitos priklausomos. Gijos gali būti naudojamos norint pagerinti programos reagavimo laiką, kuri gauna naudotojo įvedamus duomenis ir tuo tarpu vykdo kitas užduotis fone. Susijęs panaudojimo variantas yra duomenų skaitymas ir rašymas vykdant skaičiavimus kitoje gijoje.

Žemiau esantis kodas rodo kaip aukšto lygio modulis :mod:`threading` gali vykdyti užduotis fone kol pagrindinė programa yra vykdoma:

```
naudoja threading, zipfile

klasė AsyncZip(threading.Thread):
    apif __init__(self, infailas, outfailas):
        threading.Thread.__init__(self)
        self.infailas = infailas
        self.outfailas = outfailas
    apif run(self):
        f = zipfile.ZipFile(self.outfailas, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infailas)
        f.close()
        rodo 'Baigtas foninis zipas: ', self.infailas
```

```
fone = AsyncZip('manodata.txt', 'manoarchivas.zip')
fone.start()
rodo 'Main programa tęsia pirmąplanį darbą.'

fone.join()    # Laukia kol foninė užduotis baigsis
rodo 'Main programa laukė kol foninė baigė darbą.'
```

Pagrindinė problema daugiagijiškoje programoje yra gijų, kurios dalinasi duomenimis ir kitais resursais, koordinavimas. Gijų modulis leidžia naudoti sinchronizavimo primityvus kaip spynas (angl. lock), įvykius (angl. event), sąlygų kintamuosius (angl. condition variable) ar semaforus.

Nors šios priemonės yra galingos, mažiausia klaida gali baigtis klaida, kurią sunku atgaminti. Rekomenduojamas būdas užduočių koordinavimui yra sukoncentruoti priėjimą prie visų resursų vienoje gijoje ir tada naudoti :mod:`Queue` modulį, tam kad tai gijai perduoti užsakymus iš kitų gijų. Programos, kurios naudoja :class:`Queue.Queue` objektus bendravimui tarp gijų yra lengviau sukuriamos, skaitomesnės ir patikimesnės.

Žurnalas

:mod:`logging` modulis leidžia naudoti pilną ir lanksčią žurnalo sistemą. Paprasčiausias variantas yra visus žurnalo pranešimus siųsti į failą arba `sys.stderr`:

```
naudoja logging
logging.debug('Riktinimo informacija')
logging.info('Informacinė žinutė')
logging.warning('Įspėjimas:config failas %s nerastas', 'server.conf')
logging.error('Veikimo klaida')
logging.critical('Kritinė klaida -- shutting down')
```

To rezultatas yra toks:

```
WARNING:root:Įspėjimas:config failas server.conf nerastas
ERROR:root:Veikimo klaida
CRITICAL:root:Kritinė klaida -- shutting down
```

Įprastai, informaciniai ir derinimo pranešimai yra sulaikomi ir rezultatas siunčiamas į standartinių klaidų rašymo vietą. Kiti išvedimo rezultatai yra pranešimų siuntimas paštu, į serverį ir t.t. Nauji filtrai gali pasirinkti skirtingus pranešimo rašymo/siuntimo būdus priklausomai nuo pranešimo prioriteto: :const:`DEBUG`, :const:`INFO`, :const:`WARNING`, :const:`ERROR`, and :const:`CRITICAL`.

Žurnalo sistema gali būti konfigūruojama tiesiogiai iš Python'o arba gali būti užkrauta iš naudotojo konfigūruojamo failo.

Silpnos nuorodos

Python'as atmintį valdo automatiškai (daugumai objektų skaičiuojamos nuorodos ir naudojamas :term:`šiukšlių surinkimas` ciklų eliminavimui). Atmintis atlaisvinama ne už ilgo po to, kai paskutinė nuorodą į ją yra panaikinama.

Šis būdas tinka daugumai programų bet kartais reikia sekti objektus tik tiek kiek jie yra naudojami kažkur kitur. Nelaimei vien tam, kad juos sekti, yra sukuriama nuoroda, kuri padaro juos amžinus. :mod:`weakref` modulis leidžia sekti objektus nesukūrus jiems nuorodų. Kai objektas tampa nebereikalingu jis automatiškai išimamas iš *weakref* lentelės. Tipiškai programos saugo objektus, kuriuos yra brangu sukurti:

```
>>> naudoja weakref, gc
>>> klasė A:
...     apif __init__(self, reikšmė):
...             self.reikšmė = reikšmė
...     apif __repr__(self):
...             grąžina str(self.reikšmė)
...
>>> a = A(10)                   # sukuriame nuorodą
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # nuoroda nesukuriama
>>> d['primary']                # gauname objektą, jei jis dar gyvas
10
>>> del a                       # pašaliname nuorodą
>>> gc.collect()                # paleidžiame šiukšlių surinkimą
0
>>> d['primary']                # įrašas buvo automatiškai pašalintas
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # įrašas buvo automatiškai pašalintas
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

Priemonės darbui su sąrašais

Dauguma duomenų struktūrų poreikių gali būti pasiekti naudojant įtaisytus tipus. Tačiau kartais reikia alternatyvaus įgyvendinimo su skirtingais našumo reikalavimais.

:mod:`array` modulyje rasite :class:`array()` objektą, kuris yra panašus į sąrašą, kuris laiko vienodus objektus ir laiko juos kompaktiškiau.

Sekantis pavyzdys parodo skaičių masyvą, laikomą kaip dviejų baitų dvejetainius skaičius be ženklo (tipo kodas `"H"`) vietoje 16 baitų kiekvienam nariui kaip tai daroma paprastame Python'o sąraše:

```
>>> Iš array naudoja array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

:mod:`collections` modulyje rasite :class:`deque()` objektą, panašų į sąrašą su greitesniais sąrašo papildymais ir išėmimais iš kairės, bet lėtesne peržiūra iš vidurio. Šie objektai labai tinka eilių ir kai kurių paieškos medžių įgyvendinimui:

```
>>> Iš collections naudoja deque
>>> d = deque(["užduotis1", "užduotis2", "užduotis3"])
>>> d.append("užduotis4")
>>> rodo "Vykdo", d.popleft()
Vykdo užduotis1

neieškotas = deque([pradinis_mazgas])
def rasti_pirmą_plotyje(neieškotas):
    mazgas = neieškotas.popleft()
    kožnam m iš gen_moves(mazgas):
        jei is_goal(m):
            grąžina m
        neieškotas.append(m)
```

Papildomai alternatyviems sąrašų įgyvendinimams, biblioteka taip pat siūlo priemones kaip :mod:`bisect` modulis su funkcijomis skirtoms surikiuoti sekų manipuliavimui:

```
>>> naudoja bisect
>>> taškai = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(taškai, (300, 'ruby'))
>>> taškai
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

:mod:`heapq` modulyje rasite funkcijas reikalingas krūvų (angl. heap) paremtų sąrašais įgyvendinimui. Mažiausia reikšmė visada laikoma nulinėje pozicijoje. Tai praverčia programose, kurioms dažnai reikia pasiekti mažiausią elementą nerikiuojant sąrašo pilnai:

```
>>> Iš heapq naudoja heapify, heappop, heappush
```

```
>>> duomenys = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(duomenys)                      # surikiuojame sąrašą krūvos tvarka
>>> heappush(duomenys, -5)                 # pridedame naują narį
>>> [heappop(duomenys) kožnam i iš range(3)]  # gauname tris mažiausius narius
[-5, 0, 1]
```

Dešimtainė slankaus kablelio aritmetika

:mod:`decimal` modulis leidžia naudoti :class:`Decimal` duomenų tipą skirta dešimtainei slankaus kablelio aritmetikai. Jeigu palyginsime šį modulį su įtaisyta :class:`float` klase skirta dvejetainėms slankaus kablelio operacijoms, tai ši nauja klasė labai praverčia finansinėms programoms ir kitiems atvejams, kurie reikalauja tikslios dešimtainės išraiškos, tikslumo kontrolės, tikslumo apvalinant norint atitikti legalumo ar reguliavimo reikalavimus, sekimo iki pasirinktos svarbios dešimtainės vietos, arba programoms, kur naudotojas tikisi, kad rezultatas sutaps su ranka atliktais skaičiavimais.

Pavyzdžiui, 5% mokesčio skaičiavimas 70 centų telefono sąskaitai duos skirtingus rezultatus naudojant dešimtainį ir dvejetainį slankų kablelį. Skirtumas pasidaro akivaizdus, jeigu rezultatas yra suapvalinamas iki artimiausio cento:

```
>>> kožnam decimal naudoja *
>>> Decimal('0.70') * Decimal('1.05')
Decimal("0.7350")
>>> .70 * 1.05
0.73499999999999999
```

:class:`Decimal` rezultatas palieka paskutinį nulį, automatiškai nurodantį keturių vietų po kablelio svarbą iš dviejų daugiklių su dviem svarbiomis vietomis po kablelio. Dešimtainis skaičiavimas atspindi matematiką atliekamą ranka ir padeda išvengti problemų, kurios kyla kai dvejetainis slankus kablelis negali tiksliai atspindėti dešimtainių kiekių.

Tiksli reprezentacija leidžia :class:`Decimal` klasei atlikti modulio skaičiavimus ir lygybės testus, kurie nėra įmanomi naudojant dvejetainį slankų kablelį:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal("0.00")
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
```

```
>>> sum([0.1]*10) == 1.0
False
```

:mod:`decimal` modulis leidžia atlikti aritmetines operacijas su tokiu tikslumu kokio reikia:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857")
```

Kas dabar?

Šis vadovėlis tikriausiai sustiprino jūsų norą naudoti Python'ą --- jūs tikriausiai nekantraujate panaudoti Python'ą realių pasaulio problemų sprendime. Kur jūs turėtumėte eiti, kad išmoktumėte daugiau?

Šis vadovėlis yra Python'o dokumentacijos dalis. Kai kurie kiti dokumentai esantys joje yra:

- Python Library Reference_

  Jūs turite naršyti šį vadovą, kuris pilnai (nors ir glaustai) aprašo tipus, funkcijas ir standartinės bibliotekos modulius. Standartinė Python'o distribucijoje yra daug papildomo kodo. Čia rasite modulių skirtų skaityti Unix'ines pašto dėžutes, gauti dokumentus naudojantis HTTP, generuoti atsitiktinius skaičius, analizuoti komandinę eilutę, rašyti CGI programas, suspausti duomenis ir atlikti kitas užduotis. Jei tiesiog peržvelgsite Library Reference dokumentą turėsite susidarę bendrą įspūdį, ką galima padaryti.

- Installing Python Modules_ paaiškina kaip įdiegti išorinius modulius parašytus kitų Python'o naudotojų.

- Python Reference Manual_: detalus Python'o sintaksės ir semantikos išaiškinimas. Jis sunkiai skaitomas, bet čia rasite naudingos informacijos apie pačią kalbą.

Daugiau Python'o resursų:

- http://www.python.org: pagrindinis Python'o tinklapis. Jame rasite kodą, dokumentacija ir nuorodų į kitus su Python'u susijusius tinklapius.

- http://docs.python.org: greitai pasiekiama Python'o dokumentacija.

- http://pypi.python.org: Python Package Index (seniau dar vadinosi Cheese Shop) yra naudotojų sukurtų Python'o modulių indeksas, kuriuos galima parsisiųsti. Kai tik pradėsite leisti kodą, jūs galėsite užsiregistruoti čia, kad kiti galėtų jį rasti.

- http://aspn.activestate.com/ASPN/Python/Cookbook/: The Python Cookbook yra kodo pavyzdžių kolekcija, didesni moduliai ir naudingi skriptai. Geresni pavyzdžiai yra surinkti ir išleisti kaip knyga pavadinimu Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

Apie su Python'u susijusias problemas ir iškilusius klausimus galite rašyti į naujienų grupę :newsgroup:`comp.lang.python` arba siųsti juos į el. konferenciją python-list@python.org. Naujienų grupė ir el. konferencija yra susietos, todėl jei žinutę siųsite į vieną iš jų, tai ji bus automatiškai persiunčiama į kitą. Prieš klausdami galite pasitikrinti Frequently Asked Questions (Dažniausiai Užduodami Klausimai). El. konferencijos archyvą galima rasti http://mail.python.org/pipermail/.

Slankaus kablelio aritmetika: problemos ir apribojimai

Slankaus kablelio skaičiai kompiuterio atmintyje yra reprezentuojami kaip dvejetainės dalys. Pavyzdžiui, dešimtainis skaičius:

```
0.125
```

yra dešimtainių dalių suma 1/10 + 2/100 + 5/1000. Tuo tarpu skaičius užrašytas dvejetainėje formoje:

```
0.001
```

yra lygus 0/2 + 0/4 + 1/8. Šie du skaičiai yra lygūs. Vienintelis skirtumas tarp jų yra toks, kad pirmasis yra parašytas dešimtainėje sistemoje, o antrasis dvejetainėje.

Nelaimei, dauguma realių dešimtainių skaičių negali būti tiksliai pavaizduoti dvejetainėje formoje. To pasekmė yra tokia, kad dauguma dešimtainių skaičių yra tik apytiksliai išreiškiami dvejetaine forma laikoma kompiuteryje.

Problemą iš pradžių lengviau suprasti dešimtainėje sistemoje. Tarkime turime 1/3. Apytiksliai dešimtainėje sistemoje tai galima užrašyti kaip:

```
0.3
```

arba, geriau,

```
0.33
```

arba, dar geriau,

```
0.333
```

ir t.t. Nesvarbu kiek skaičių jūs užrašysite, rezultatas niekada nebus tiksliai lygus 1/3, bet tik bus tikslesnė 1/3 aproksimacija.

Tuo pačiu būdu, nesvarbu kiek dvejetainio skaičiaus skaitmenų naudosite, dešimtainė reikšmė 0.1 negali būti tiksliai pavaizduota dvejetainėje sistemoje. Dvejetainėje sistemoje 1/10 yra amžinai pasikartojanti seka:

```
0.0001100110011001100110011001100110011001100110011...
```

Sustojate ties tam tikru skaitmeniu ir jūs turite apytikslę reikšmę. Štai todėl jūs matote tokius dalykus:

```
>>> 0.1
0.10000000000000001
```

Tai yra būtent tai, ką jūs matysite jei įvesite 0.1 reikšmę Python'e. Aišku, galbūt jūs matysite kitokį vaizdą, nes skirtingi kompiuteriai gali naudoti skirtingą bitų skaičių laikyti slankaus kablelio skaičius. Python'as atspausdina dvejetainės aproksimacijos dešimtainėje formoje. Jeigu Python'as turėtų atspausdinti tikrą dešimtainę 0.1 reikšmę, jis turėtų parodyti:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Python'as naudoja įtaisytą funkciją :func:`repr`, kad parodytų objekto eilutės versiją. Slankaus kablelio skaičiams `repr(float)` suapvalina tikrą dešimtainę reikšmę iki 17 ženklo:

```
0.10000000000000001
```

`repr(float)` parodo 17 ženklų po kablelio kadangi to pakanka daugeliui kompiuterių, todėl `eval(repr(x)) == x` yra teisinga visiems slankaus kablelio *x*, bet jeigu skaičius bus suapvalintas iki 16 skaičiaus po kablelio tai nebebus tiesa.

Pastebėsime, kad taip tiesiog veikia dvejetainiai slankaus kablelio skaičiai ---tai nėra Python ar klaida jūsų kode. Jūs galite pamatyti tokį patį veikimą ir kitose kalbose, kurios palaiko aparatinę slankaus kablelio aritmetiką. Nors kai kurios kalbos gali ir neparodyti *skirtumo* įprastai.

Python'o įtaisytoji funkcija :func:`str` panaudoja tik 12 skaičių po kablelio ir jūs dažniausiai ją ir norėtsite naudoti. `eval(str(x))` tikriausiai neatgamins *x* reikšmės, bet rezultatas gali būti mielesnis akiai:

```
>>> rodo str(0.1)
0.1
```

Labai svarbu suprasti, kad tai yra iliuzija: reikšmė mašinoje nėra tiksliai 1/10 --- jūs tiesiog apvalinate tikrąją kompiuterio reikšmę.

Iš čia seka kita siurprizai. Pavyzdžiui, pamatę:

```
>>> 0.1
0.10000000000000001
```

jūs galite užsimanyti panaudot :func:`round` (apvalinimo) funkciją, taip tikėdamiesi gauti reikšmę su vienu skaičiumi po kablelio. Tačiau tai nieko nepakeis:

```
>>> round(0.1, 1)
0.10000000000000001
```

Problema yra ta, kad slankaus kablelio formoje išsaugota „0.1“ reikšmė jau yra geriausia įmanoma dvejetainė 1/10 aproksimacija, taigi bandymas ją apvalinti nieko nepakeis --- ji jau buvo tokia gera kokia tik gali būti.

Kita problema yra, tokia, kad 0.1 nėra tiksliai 1/10, taigi susumavus 0.1 dešimt kartų negausime tiksliai 1.0:

```
>>> sum = 0.0
>>> kožnam i iš range(10):
...     sum += 0.1
...
>>> sum
0.99999999999999989
```

Dvejetainė slankaus kablelio aritmetika slepia ne vieną tokį siurpriza. Problema su „0.1“ detaliau paaiškinta žemiau skyriuje „Atvaizdavimo Klaida“. The Perils of Floating Point rasite daugiau dažnų siurprizų pavyzdžių.

Kaip galiausiai sakoma „lengvų atsakymų nėra“. Visgi nebūkite per daug atsargūs dirbdami su slankiu kableliu. Klaidos Python'e kylančios su slankiu kableliu yra paveldėtos iš aparatinės įrangos ir daugumoje mašinų klaidos galimybė yra ne didesnė negu 1 iš 2**53. Tai daugiau negu adekvatu daugumai užduočių, bet jūs turite prisiminti, kad tai ne dešimtainė aritmetika ir kad kiekviena slankaus kablelio operacija prideda apvalinimo klaidą.

Nors egzistuoja patologiniai atvejai, dauguma atvejų jūs matysite tą rezultatą, kurio ir tikėjotės, jei galiausiai rezultatą suapvalinsite iki norimo skaičiaus po kablelio. Dažniausiai pakanka :func:`str` funkcijos, o jei reikia daugiau galimybių naudokite :meth:`str.format` metodą.

Atvaizdavimo klaida

Šis skyrius paaiškina „0.1“ pavyzdį detaliau ir paaiškina, kaip jūs galite atlikti tokių atvejų analizę patys. Darome prielaidą, kad su slankaus kablelio dvejetainiais skaičiais skaitytojas yra susipažinęs.

:dfn:`Atvaizdavimo klaida` reiškia, kad kai kurie (tiesa sakant, dauguma) dešimtainių trupmenų negali būti išreikštos kaip dvejetainės trupmenos. Tai yra pagrindinė priežastis kodėl Python'as (Perl, C,

C++, Java, Fortran ir daug kitų kalbų) dažnai tiksliai neatvaizduos dešimtainio skaičiaus kaip jūs tikitės:

```
>>> 0.1
0.10000000000000001
```

Kodėl tai atsitinka? 1/10 nėra tiksliai reprezentuojama kaip dvejetainė trupmena. Beveik visos mašinos šiandien (2000 Lapkritis) naudoja IEEE-754 slankaus kablelio aritmetiką ir beveik visose platformose Python'e naudojamas „dvigubo tikslumo“ IEEE-754 slankaus kablelio skaičius. 754 naudoja 53 tikslumo bitus, taigi 0.1 konvertuojamas į artimiausią trupmeną *J*/2 *N* formoje, kur *J* yra sveikasis skaičius sudarytas iš 53 bitų. Perrašius:

```
1 / 10 ~= J / (2**N)
```

kaip

```
J ~= 2**N / 10
```

ir tarus, kad *J* turi lygiai 53 bitus (yra `>= 2**52` bet `< 2**53`), tinkamiausia reikšmė *N* yra 56:

```
>>> 2**52
4503599627370496L
>>> 2**53
9007199254740992L
>>> 2**56/10
7205759403792793L
```

Taip jau yra, kad 56 yra vienintelė *N* reikšmė, kurią naudojant *J* turi tiksliai 53 bitus. Tinkamiausia *J* reikšmė tada yra suapvalintas dalmuo:

```
>>> q, r = divmod(2**56, 10)
>>> r
6L
```

Kadangi liekana yra didesnė negu 10 pusė, geriausia aproksimacija gaunama apvalinant viršun:

```
>>> q+1
7205759403792794L
```

Taigi geriausia 1/10 aproksimacija 754 formoje yra 2**56, arba

```
7205759403792794 / 72057594037927936
```

Pastebėkite, kad dėl to jog mes ją suapvalinome aukštyn, tai yra šiek tiek daugiau negu 1/10. Jei mes nebūtumėm apvalinę dalmuo būtų buvęs truputi mažesnis negu 1/10. Bet jokiu būtų jis negali būti *tiksliai* lygus 1/10.

Taigi kompiuteris niekada „nemato“ 1/10. Tai ką jis mato yra tiksli trupmena duota aukščiau. Geriausia 254 dviguba aproksimacija, kurią jis gali gauti yra:

```
>>> .1 * 2**56
7205759403792794.0
```

Jei mes padaugintumėme tą trupmeną iš 10**30 mes galėtumėme pamatyti (nukirptą) reikšmę iki 30 ženklų po kablelio:

```
>>> 7205759403792794 * 10**30 / 2**56
100000000000000005551115123125L
```

kas reiškia, jog tiksli reikšmė laikoma kompiuteryje apytiksliai yra lygi 0.100000000000000005551115123125. Suapvalinus ją iki 17 ženklų po kablelio gauname 0.10000000000000001, kurią ir rodo Python'as. (tiksliau, taip bus rodoma ant bet kurios 754-formą palaikančios platformos, kurioje C biblioteka padaro geriausią įmanomą įvesties ir išvesties konvertavimą --- jūsų sistemą gali būti kitokia!).

Terminų žodynas

``>>>``
Numatytasis Python'o raginimas interaktyviajame apvalkale. Dažnai naudojamas pavyzdžiuose, kurie gali būti paleidžiami interpretatoriuje.

``...``
Numatytasis Python'o raginimas interaktyviajame apvalkale kai įvedamas kodas pastumtam kodo blokui arba tarp poros sutampančių kairiojo ir dešiniojo skirtukų (skliaustų, laužtinių arba figūrinių skliaustų).

2to3
Įrankis, kuris bando konvertuoti Python'o 2.x kodą į Python'ą 3.x kodą. Konvertuojami tie kodo nesuderinamumai, kurie gali būti nustatyti analizuojant kodą ir pereinant analizės medį.

2to3 įrankis yra standartinėje bibliotekoje kaip :mod:`lib2to3`; Įrankis pateikiamas kaip :file:`Tools/scripts/2to3`. Žiūrėti :ref:`2to3-reference`.

abstrakčioji bazinė klasė
Abstrakčioji bazinė klasė (ABC, ang. abstract base class) papildo neišreikštinį tipizavimą suteikdama būdą apibrėžti klasės šabloninę sąsają kai kitų technikų (pvz.: :func:`hasattr`) naudojimas yra nepatogus. Python'e yra daug standartinių ABC duomenų struktūroms (:mod:`collections` modulyje), skaičiams (:mod:`numbers` modulyje), srautams (:mod:`io` modulyje). Galima kurti savo ABC naudojant :mod:`abc` modulį.

argumentas
Reikšmė perduota funkcijai arba metodui ir priskirta lokaliam kintamajam su vardu funkcijoje. Funkcija arba metodas gali turėti tiek pozicinius argumentas tiek vardinius argumentus apibrėžime. Poziciniai ir vardiniai argumentai gali turėti kintamą ilgį: ``*`` priima arba perduoda (jei funkcijos apibrėžime arba kvietime) kelis pozicinius argumentus sąrašu, kai tuo tarpu ``**`` atlieka tą patį su vardiniais argumentais žodyne.

Betkoks reiškinys gali būti naudojamas argumentų sąraše ir apskaičiuota reikšmė yra perduodama lokaliam kintamajam.

atkarpa (Ang. slice)

Objektas, kuris paprastai turi dalį :term:`sekos <seka>`. Atkarpa paprastai sukuriama naudojant ``[]`` išraišką su dvitaškiais tarp skaičių, kai keli paduodami, pvz.: ``variable_name[1:3:5]``. Laužtiniai skliaustai naudoja klasės :class:`slice` objektus (arba senesnėse versijose :meth:`__getslice__` ir :meth:`__setslice__` metodus).

atributas

Reikšmė susieta su objektu, kuri yra pasiekiama pagal vardą naudojant taško išraišką. Pvz.: jei objektas *o* turi atributą *a* tai jis gali būti pasiektas kaip *o.a*.

atvaizdis (Ang. mapping)

Konteinerio objektas (pvz.: :class:`dict`) kuris palaiko reikšmių paiešką pagal raktus naudojant specialų metodą :meth:`__getitem__`.

baitinis kodas

Python'o kodas yra kompiliuojamas į baitinį kodą – vidinę Python'o programos reprezentaciją interpretatoriuje. Python'o baitinis kodas yra išsaugojamas ``.pyc`` ir ``.pyo`` failuose, kad to paties failo vykdymas būtų greitesnis vykdant antrą kartą (išvengiama perkompiliavimo iš kodo į baitinį kodą). Sakoma, kad "tarpinė kalba" yra vykdoma :term:`virtualiojoje mašinoje <virtualioji mašina>` kuri vykdo mašininį kodą kiekvienam baitiniam kodui.

BFDL

Dosnusis Diktatorius Visam Gyvenimui (angliško termino Benevolent Dictator For Life sutrumpinimas), taip pat žinomas kaip `Guido van Rossum <http://www.python.org/ guido/>`_, Python'o kūrėjas.

CPython

Pagrindinis Python'o programavimo kalbos įgyvendinimas. Terminas „CPython“ naudojamas, kai reikia atskirti šį įgyvendinimą nuo kitų, pvz.: Jython arba IronPython.

dekoratorius

Funkcija, kuri grąžina kitą funkciją, dažniausiai naudojant funkcijos transformacijos sintaksę ``@wrapper``. Dažnas pavyzdys dekoratoriams yra :func:`classmethod` ir :func:`staticmethod`.

Dekoratoriaus sintaksė yra sukurta tik dėl patogumo. Pavyzdžiui šie du funkcijos apibrėžimai yra lygiaverčiai::[120]

```
def f(...): ... f = staticmethod(f)
@staticmethod def f(...): ...
```

[120]

```
apif f(...): ... f = statinismetodas(f)
@staticmethod def f(...): ...
```

deskriptorius

Bet kuris *naujo stiliaus* objektas, kuris apibrėžia metodus :meth:`__get__`, :meth:`__set__` arba :meth:`__delete__`. Kai klasės atributas yra deskriptorius, jo specialūs metodai kviečiami tuo metu, kai bandoma pasiekti atributą. Paprastai norint gauti, nustatyti ar ištrinti *a.b* atributą yra ieškoma objekto *b* klasės *a* žodyne, bet jei *b* yra deskriptorius, iškviečiamas atitinkamas deskriptoriaus metodas. Deskriptorių

supratimas yra reikalingas išsamiam Python'o supratimui, nes jie yra pagrindas daugeliui savybių: funkcijoms, metodams, savybėms, klasės metodams, statiniams metodams ir rodyklėms į super klases.

dokumentavimo eilutė (Ang. docstring)

Eilutės objektas, kuris yra pirmas reiškinys klasėje, funkcijoje ar modulyje. Nors šis objektas yra ignoruojamas, kai kodas yra vykdomas, kompiliatorius jį atpažįsta ir priskiria :attr:`__doc__` atributui. Kadangi jis yra pasiekiamas naudojant introspekciją, tai yra įprasta vieta objektų dokumentavimui.

EAFP

Lengviau paprašyti gailestingumo nei leidimo. Programuojant šiuo įprastu Python'o stiliumi daroma prielaida, kad raktai ar atributai egzistuoja ir gaudomos išimtys, jei prielaida yra neteisinga. Šis švarus ir greitas stilius yra charakterizuojamas raktinių žodžių :keyword:`try` ir :keyword:`except` egzistavimu. Ši technika kontrastuoja su :term:`LBYL` stiliaus programavimu dažnu daugelyje kitų programavimo kalbų (pvz.: C).

eilutė su trigubomis kabutėmis (Ang. triple-quoted string)

Eilutė, kuri yra apsupta trimis kabutėmis (") arba apostrofais ('). Nors jose nėra jokio funkcionalumo, kurio negalima būtų padaryti su paprastomis eilutėmis, jos yra naudingos dėl kelių priežasčių. Jos leidžia naudoti viengubas arba dvigubas kabutes be kaitos (

) ženklo ir leidžia sujungti kelias eilutes be pratęsimo simbolio. Dėl to jos labai naudingos dokumentavimo eilutėse.

funkcija

Sakinių grupė grąžinanti reikšmę. Funkcijai gali būti perduoti argumentai, kurie gali būti naudojami skaičiavimuose. Taip pat žiūrėti terminus :term:`argumentas` ir :term:`metodas`.

__future__

Pseudo-modulis, kurį programuotojai gali naudoti norėdami įjungti kalbos savybes, kurios nėra suderinamos su dabartine interpretatoriaus versija. Pavyzdžiui reiškinys ``11/4`` apskaičiuojamas kaip ``2``. Jei modulyje kuriame šis reiškinys yra vykdomas įjungiama *true division* įvykdant::

from __future__ import division

reiškinys ``11/4`` bus apskaičiuotas kaip ``2.75``. Importavę :mod:`__future__` modulį ir įvertinę jo kintamuosius, jūs galite matyti kada naujos savybės buvo pridėtos į kalbą ir kada jos bus numatytosios::

>>> import __future__ >>> __future__.division _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)

generatorius

Funkcija, kuri grąžina iteratorių. Ji panaši į normalią funkciją išskyrus tai, kad jos reikšmės yra grąžinamos naudojant :keyword:`yield` sakinį vietoje :keyword:`return` sakinio. Generatoriaus funkcijos

dažnai susideda iš vieno arba daugiau :keyword:`for` arba :keyword:`while` ciklų, kurie grąžina :keyword:`yield` elementus. Funkcijos vykdymas yra sustabdomas ties :keyword:`yield` raktiniu žodžiu (sugrąžinant rezultatą) ir pratęsiamas, kai pareikalaujama kito elemento iškviečiant iteratoriaus :meth:`next` metodą.

.. index:: single: generator expression

generatorinis reiškinys

Reiškinys, kuris grąžina generatorių. Jis panašus į normalų reiškinį po kurio seka :keyword:`for` reiškinys apibrėžiantis ciklo kintamuosius, ribas ir papildomą :keyword:`if` reiškinį. Kombinuotas reiškinys sugeneruoja reikšmes reiškinį apimančiai funkcijai::

>>> sum(i*i for i in range(10)) # sumuojame kvadratu pakeltus skaičius 0, 1, 4, ... 81 285

GIL

Žiūrime :term:`globalus interpretatoriaus užrakinimas`.

globalus interpretatoriaus užrakinimas (Ang. global interpreter lock)

Python'o gijų naudojamas užrakinimas, kuris garantuoja, kad tik viena gija vykdo :term:`CPython` :term:`virtualiąją mašiną <virtualioji mašina>` vienu metu. Tai supaprastina CPython įgyvendinimą užtikrinant, kad du procesai negali pasiekti tos pačios atminties vienu metu. Viso interpretatoriaus užrakinimas leidžia interpretatoriui lengviau vykdyti kelias gijas vienu metu. Žinoma, dėl to Python'o interpretatorius pralošia daugiaprocesorinėse mašinose. Praeityje buvo ne vienas bandymas sukurti laisvų gijų interpretatorių (tokį, kuris užrakina bendrus duomenis žemesniame lygyje), bet nė vienas nebuvo sėkmingas, nes nukentėdavo greitis dažnai pasitaikančiose vieno-procesoriaus mašinose.

IDLE

Integruota Programavimo Aplinka Python'ui. IDLE yra paprastas redaktorius ir interpretuojama aplinka, kuri pateikiama su standartiniu Python'u. Tinkama pradedantiesiems, bet taip pat gali būti naudojama kaip pavyzdinis kodas tiems, kurie nori sukurti multi-platforminę GUI aplikaciją.

interaktyvus

Python'as turi interaktyvų interpretatorių, o tai reiškia, kad sakinius ir reiškinius galima įvesti interpretatoriaus raginime, iškart juos įvykdyti ir pamatyti rezultatą. Tiesiog paleiskite ``python`` be argumentų (tikriausiai galite jį rasti pagrindiniame kompiuterio meniu). Tai labai galingas įrankis išbandyti naujas idėjas arba analizuoti modulius (prisiminkite ``help(x)``).

interpretuojamas

Python'as yra interpretuojama kalba, o ne kompiliuojama, nors skirtumas gali būti ir nepastebimas dėl baitinio kodo kompiliatoriaus. Tai reiškia, kad kodo failai gali būti vykdomi tiesiogiai nekuriant vykdomųjų

failų, kurie vėliau turi būti paleisti. Interpretuojamos kalbos dažniausiai padeda greičiau programuoti ir derinti, negu dirbant su kompiliuojančiomis kalbomis, tačiau įprastai jos yra lėtesnės. Žiūrime taip pat :term:`interaktyvus`.

iteruojamas

Objektas, kuris gali grąžinti savo narius po vieną. Iteruojamo objekto pavyzdys galėtų būti bet kuris sekos tipas (pvz.: :class:`list`, :class:`str`, ir :class:`tuple`) ir kai kurie ne sekos tipai (pvz.: :class:`dict` ir :class:`file`) ir bet kurios klasės objektai, kurie apibrėžia :meth:`__iter__` arba :meth:`__getitem__` metodus. Iteruojami objektai gali būti naudojami :keyword:`for` cikle ir daugelyje kitų vietų, kur reikalingos sekos (:func:`zip`, :func:`map`, ...). Kai iteruojamas objektas perduodamas funkcijai :func:`iter` kaip argumentas, ji grąžina objektui iteratorių. Šis iteratorius yra naudingas, kai reikia pereiti per aibės reikšmes. Kai naudojami iteruojami objektai nėra būtina kviesti :func:`iter` ar tvarkytis su iteratoriaus objektu pačiam. ``for`` sakinys tai atlieka automatiškai sukurdamas laikiną bevardį kintamąjį, kuriame laikomas iteratorius ciklui. Taip pat žiūrėti: :term:`iteratorius`, :term:`seka`, ir :term:`generatorius`.

iteratorius

Objektas, kuris reprezentuoja duomenų srautą. Pakartotini iteratoriaus metodo :meth:`next` kvietimai grąžina sekantį narį sraute. Kai daugiau duomenų nebėra, sukeliama :exc:`StopIteration` išimtis. Nuo šios akimirkos iteratoriaus objektas yra išnaudotas ir bet kuris tolesnis :meth:`next` metodo kvietimas tiesiog išmes :exc:`StopIteration` vėl. Iteratoriai taip pat turi turėti :meth:`__iter__` metodą, kuris sugrąžina iteratoriaus objektą tam, kad pats iteratorius galėtų būti naudojamas kitoje vietoje, kur reikalingi iteruojami objektai. Viena pažymėtina išimtis yra kodas, kuris bando įvykdyti kelis iteracijos žingsnius. Konteinerio objektas (pvz.: :class:`list`) grąžina naują iteratorių kiekvieną kartą, kai jį perduodi :func:`iter` funkcijai arba naudoji :keyword:`for` cikle. Bandymai atlikti tą patį su iteratoriumi tiesiog sugrąžins tą patį išnaudotą iteratoriaus objektą naudotą praeitame iteracijos vykdyme ir dėl to jis atrodys kaip tuščias konteineris.

Daugiau informacijos galima rasti :ref:`typeiter`.

išplėtimo modulis (Ang. extension module)

Modulis parašytas C arba C++ naudojant Python'o C API skirtas bendradarbiavimui tarp Python'o branduolio ir vartotojo kodo.

įdėtinė sritis (Ang. nested scope)

Galimybė kreiptis į kintamuosius ribotame apibrėžime. Pavyzdžiui, funkcija apibrėžta kitos funkcijos viduje gali kreiptis į kintamąjį išorinėje funkcijoje. Atkreipkite dėmesį, kad įdėtinė sritis dirbs tik su nuorodomis, bet net priskyrimams, kurie yra visada rašomi artimiausioje srityje.

klasė

Šablonas, kuris naudojamas kurti vartotojo apibrėžtus objektus. Klasės apibrėžimas dažniausiai susideda iš metodų apibrėžimų, kurie operuoja su tos klasės objektais.

klasikinė klasė

Bet kuri klasė, kuri nėra paveldėta iš :class:`object`. Taip pat žiūrėti :term:`naujo stiliaus klasė`. Klasikinės klasės bus išimtos iš Python'o 3.0.

keitimas (Ang. coercion)

Neišreikštinis vieno tipo objekto konvertavimas į kitą tipą per operaciją, kurioje naudojami du to paties tipo argumentai. Pavyzdžiui ``int(3.15)`` konvertuoja slankiojo kablelio skaičių į sveiką skaičių ``3``, bet operacijoje ``3+4.5`` kiekvienas argumentas yra skirtingo tipo (vienas yra sveikas skaičius, kitas slankaus kablelio) ir abu turi būti sukonvertuojami į tą patį tipą prieš tai, kai jie yra sudedami arba kitaip bus išmetama ``TypeError`` klaida. Keitimas tarp dviejų objektų gali būti atliktas naudojantis ``coerce`` funkcija. Taigi ``3+4.5`` yra tolygu ``operator.add(*coerce(3, 4.5))`` kvietimui ir tokios operacijos rezultatas yra ``operator.add(3.0, 4.5)``. Be keitimo visi argumentai (net jei argumentų tipai yra suderinami) turėtų būti normalizuojami programuotojo, pvz.: ``float(3)+4.5`` užuot tiesiog rašius ``3+4.5``.

kintamas (Ang. mutable)

Kintamas objektas gali pakeisti savo reikšmę bet išlaikyti :func:`id`. Taip pat žiūrėti :term:`nekintamas`.

kompleksinis skaičius

Pažįstamos realiųjų skaičių sistemos plėtinys, kuriame visi skaičiai yra išreiškiami kaip realios ir menamosios dalių suma. Menamieji skaičiai yra realieji skaičiai padauginti iš menamojo vieneto (kvadratinė šaknis iš ``-1``), dažnai žymimo kaip ``i`` matematikoje arba ``j`` inžinerijoje. Python'as palaiko kompleksinius skaičius, kurie naudoja pastarąjį žymėjimą – menamoji dalis yra rašoma naudojant ``j``, pvz.: ``3+1j``. Jei jums reikia :mod:`math` modulio ekvivalento kompleksiniams skaičiams naudokite :mod:`cmath`. Kompleksinių skaičių naudojimas yra pakankamai sudėtinga matematikos tema. Jei nesate tikras ar jums jų reikia, galite ramiai juos ignoruoti.

konteksto valdiklis (Ang. context manager)

Objektas, kuris valdo aplinką sakinyje su :keyword:`with` konstrukcija. Objekte reikia apibrėžti :meth:`__enter__` ir :meth:`__exit__` metodus. Daugiau informacijos :pep:`343`.

lambda

Anoniminė funkcija, susidedanti iš vieno reiškinio, kuris apskaičiuojamas, kai funkcija kviečiama. Lambda funkcijų sintaksė yra ``lambda [argumentai]: reiškinys``.

LBYL

Pažiūrėk prieš šokdamas. Programuojant šiuo stiliumi patikrinamos sąlygos prieš vykdant tolimesnius veiksmus. Šis stilius kontrastuoja :term:`EAFP` stiliui ir gali būti atpažintas pagal didelį :keyword:`if` sakinių kiekį.

maišomas (Ang. hashable)

Objektas yra *maišomas* jei jo maišos reikšmė niekada nepasikeičia per jo gyvavimo laiką (tam reikia :meth:`__hash__` metodo) ir jis gali būti palygintas su kitais objektais (tam reikia :meth:`__eq__` arba :meth:`__cmp__` metodų). Maišomi objektai, kurie turi lygias reikšmes lyginant, privalo turėti lygias maišos reikšmes.

Maišomumas leidžia objektą naudoti žodynuose ir aibėse, nes šios duomenų struktūros naudoja maišos reikšmes.

Visi nekintami Python'o objektai yra maišomi, tuo tarpu kintami objektai nėra (pvz.: konteineriai - sąrašai arba žodynai). Objektai, kurie yra sukuriami iš vartotojo apibrėžtų klasių yra maišomi visada, nes jie niekada nėra lygūs ir jų maišos reikšmė yra jų :func:`id`.

metaklasė

Klasės klasė. Klasės apibrėžimas sukuria klasės vardą, klasės žodyną ir bazinių klasių sąrašą. Metaklasė yra atsakinga už šių trijų argumentų paėmimą ir klasės sukūrimą. Dauguma objektiškai orientuotų programavimo kalbų pateikia standartinį įgyvendinimą. Python'as šiuo atveju yra išskirtinis, nes jame galima sukurti savo metaklases. Daugumai vartotojų niekada nereikia šios priemonės, bet kai atsiranda poreikis, metaklasės gali pasiūlyti galingus ir elegantiškus sprendimus. Jos yra naudojamos atributų pasiekimo žurnalų rašymui, gijų-saugumo įgyvendinime, objektų sukūrimo sekimui, singltonų įgyvendinime ir daugelyje kitų užduočių.

Daugiau informacijos gali rasti :ref:`metaclasses`.

metodas

Funkcija, kuri apibrėžiama klasės viduje. Jei ji iškviečiama kaip klasės egzemplioriaus atributas, metodas gaus egzemplioriaus objektą kaip pirmąjį savo argumentą (žiūrime :term:`argumentas`). Dažniausiai jis vadinamas ``self``. Taip pat žiūrėti :term:`funkcija` and :term:`įdėtinė sritis`.

naujo stiliaus klasė (Ang. new-style class)

Bet kuri klasė, kuri paveldi iš :class:`object`. Į tai įeina visi standartiniai tipai kaip :class:`list` ar :class:`dict`. Tik naujo stiliaus klasės gali naudoti naujas visapusiškas Python'o ypatybes kaip :attr:`__slots__`, deskriptorius, savybes ir :meth:`__getattribute__`.

neišreikštinis tipizavimas (Ang. duck-typing)

Python'iškas programavimo stilius, kuris nustato objekto tipą analizuojant jo metodus arba atributus užuot analizuojant objekto santykį su tam tikru tipu (jei versti tiesiogiai iš anglų kalbos, tai būtų anties tipizavimas „Jei tai atrodo kaip antis, kvaksi kaip antis, tai turi būti

antis“). Naudojant sąsajas vietoje specifinių tipų gerai suprojektuotas kodas yra lankstesnis, nes leidžia naudoti polimorfinį pakeitimą. Naudojant neišreikštinį tipizavimą išvengiama tikrinimo naudojant :func:`type`arba :func:`isinstance` funkcijas (Pastaba: neišreikštinis tipizavimas gali būti papildytas naudojant abstrakčiąsias bazines klases). Vietoje to įprastai naudojama :func:`hasattr` funkcija arba :term:`EAFP` programavimas.

nekintamas (Ang. immutable)

Objektas su fiksuota reikšme. Nekintamais objektais gali būti skaičiai, eilutės ir kortežai. Tokie objektai negali būti keičiami. Turi būti sukurtas naujas objektas, jei norime sukurti kitokią reikšmę. Jie yra svarbūs situacijose, kur reikia konstantinės maišos reikšmės, pvz.: rakto žodynui.

nuorodų skaičius (Ang. reference count)

Nuorodų skaičius į objektą. Kai nuorodų skaičius nukrenta iki nulio, jo užimama atmintis yra atlaisvinama. Nuorodų skaičiavimas dažniausiai yra nematomas Python'o kode, bet tai yra svarbiausias elementas :term:`CPython` įgyvendinime. :mod:`sys` modulis apibrėžia :func:`getrefcount` funkciją, kurią programuotojai gali iškviesti norėdami gauti nuorodų skaičių į tam tikrą objektą.

reiškinys (Ang. expression)

Sintaksės gabalas, kuris gali būti apskaičiuotas. Kitais žodžiais, reiškinys yra tokių elementų kaip vardai, atributų pasiekimai, operatorių arba funkcijų kvietimai, junginys, kurio visi nariai grąžina reikšmę. Kitaip, nei daugelyje kitų programavimo kalbų, ne visos Python'o konstrukcijos yra reiškiniai. Dar būna :term:`sakiniai <sakinys>`, kurie negali būti naudojami kaip reiškiniai. Pvz.: raktiniai žodžiai :keyword:`print` arba :keyword:`if`. Priskyrimai taip pat yra sakiniai, o ne reiškiniai.

objektas

Bet kuris duomenų vienetas su būsena (atributų ar reikšmės) ir apibrėžtu elgesiu (metodais). Taip pat pirmine bazine klase, jei tai :term:`naujo stiliaus klasė`.

pozicinis argumentas

Argumentai priskirti lokaliems vardams funkcijoje ar metode nustatant jų eilę pagal tai kaip jie buvo kviečiami. ``*`` yra naudojamas kai reikia priimti kelis pozicinius argumentus (apibrėžime) arba kai reikia perduoti kelis argumentus kaip sąrašą funkcijai. Žiūrėti :term:`argumentas`.

Python 3000

Kodinis pavadinimas kitai svarbiai Python'o versijai, 3.0 (sugalvota seniai, kai 3 versija dar buvo tolimoje ateityje). Taip pat trumpinama „Py3k“.

Python'iškas

Idėja arba kodo gabalas, kuris atitinka daugumą Python'o kalbos idiomų užuot įgyvendina kodą naudojant kitų kalbų koncepcijas. Pavyzdžiui dažna Python'o idioma yra pereiti per visus iteruojamo objekto elementus naudojant :keyword:`for` sakinį. Dauguma kitų kalbų neturi tokio tipo konstrukcijos, taigi žmonės nesusipažinę su Python'u naudoja skaitliukus::

for i in range(len(food)): print food[i]

Kai tuo tarpu galima naudoti Python'išką metodą::

for piece in food: print piece

Python'o Zen

Python'o dizaino principų ir filosofijų sąrašas kuris padeda suprasti ir naudoti kalbą. Sąrašą galima rasti surinkus "``import this``" interaktyviajame raginime.

__slots__ Apibrėžimas naujo stiliaus klasėje (žr. :term:`naujo stiliaus klasė`), kuris sutaupo atminties, rezervuodamas erdvę egzemplioriaus atributams ir išvengdamas egzemplioriaus žodyno. Nors technika yra populiari, kartais gali būti sudėtinga tai atlikti teisingai, todėl geriau tai palikti situacijoms, kai reikalingas didelis egzempliorių skaičius, o atmintį reikia išnaudoti efektyviai.

sakinys (Ang. statement)

Sakinys yra kodo bloko dalis. Sakinys yra arba :term:`reiškinys` arba viena iš kelių konstrukcijų su raktiniu žodžiu (pvz.: :keyword:`if`, :keyword:`while` arba :keyword:`print`).

sąrašas (Ang. list)

Standartinė Python'o :term:`seka`. Nepaisant pavadinimo jis artimesnis kitų programavimo kalbų masyvams, nei rodyklėmis susietais sąrašais, nes priėjimas prie elementų yra O(1).

sąrašo užklausa / list comprehension

Kompaktiškas būdas apdoroti visus ar dalį sekos elementų ir sugrąžinti sąrašą su rezultatais. ``result = ["0xsugeneruoja eilučių sąrašą susidedančių iš šešioliktainių skaičių (0x..) iš lyginių skaičių aibės nuo 0 iki 255. :keyword:`if` sakinys yra papildomas. Jei jis bus praleistas duotame pavyzdyje bus apdoroti visi elementai iš aibės nuo 0 iki 255.

seka (Ang. sequence)

:term:`iteruojamas` objektas, kuris palaiko efektyvų elementų pasiekimą per indeksus naudojant specialų :meth:`__getitem__` metodą ir apibrėžia :meth:`len` metodą, kuris sugrąžina sekos ilgį. Kai kurie standartiniai sekos tipai yra :class:`list`, :class:`str`, :class:`tuple` ir :class:`unicode`. Atkreipkite dėmesį į tai, kad :class:`dict` taip pat palaiko :meth:`__getitem__` ir :meth:`__len__`, bet yra atvaizdis, o ne seka, nes paieškai naudojami :term:`nekintami <nekintamas>` raktai, o ne sveikieji skaičiai.

specialus metodas

Metodas, kurį iškviečia Python'as, kad atliktų tam tikro tipo operaci-

ją (pvz.: sudėtį). Tokie metodai turi vardus prasidedančius ir pasibaigiančius dvigubu pabraukimo brūkšniu.

sveikųjų skaičių dalyba (Ang. integer division)

Matematinė dalyba numetant liekaną. Pvz.: reiškinys ``11/4`` apskaičiuojamas kaip ``2``, kai tuo tarpu realiųjų skaičių dalyboje būtų grąžinta ``2.75``. Taip pat dar vadinama *grindų dalyba*. Kai dalinami du sveikieji skaičiai rezultatas visada bus sveikas skaičius (kuriam bus pritaikyta *grindų* funkcija). Tačiau, jei vienas iš skaičių yra kitokio tipo (pvz.: :class:`float`), tai rezultatas bus keičiamas (žiūrime :term:`keitimas`) į bendrą tipą. Pvz.: jei sveikas skaičius dalinamas iš realiojo skaičiaus rezultatas bus realusis skaičius, tikriausiai su dešimtaine liekana. Sveikųjų skaičių dalyba gali būti nurodoma priverstinai naudojant ``//`` operatorių vietoje ``/`` operatoriaus. Taip pat žiūrėkite :term:`__future__`.

šiukšlių surinkimas (Ang. garbage collection)

Atminties atlaisvinimo procesas kai ji nebenaudojama. Python'as atlieka šiukšlių surinkimą naudodamas rodyklių skaičiavimą ir ciklišką šiukšlių surinkiklį, kuris sugeba aptikti ir nutraukti rodyklių ciklus.

tipas

Python'o objekto tipas nustato koks tai yra objektas. Kiekvienas objektas turi tipą. Objekto tipas pasiekiamas kaip :attr:`__class__` atributas arba gali būti nustatytas naudojant ``type(obj)``.

vardinis argumentas (Ang. keyword argument)

Argumentas prieš kurį parašoma ``variable_name=`` kvietime. Kintamojo vardas nurodo lokalų kintamąjį funkcijoje, kuriai perduodama reikšmė. ``**`` naudojamas vardinių argumentų priėmimui arba perdavimui. Daugiau :term:`argumentas`.

vardinis kortežas (Ang. named tuple)

Bet kuri į kortežą panaši klasė, kurios indeksuojami elementai gali būti pasiekti naudojant vardinius atributus. Pvz.: :func:`time.localtime` sugrąžina į kortežą panašų objektą, kur *year* (metai) gali būti pasiekti arba naudojant indeksą ``t[0]`` arba vardinį atributą ``t.tm_year``).

Vardinis kortežas gali būti standartinis tipas (pvz.: :class:`time.struct_time`) arba jis gali būti sukurtas naudojant įprastus klasės apibrėžimus. Visas savybes turintį vardinį kortežą galima sukurti naudojantis :func:`collections.namedtuple` funkcija. Šis sprendimas automatiškai suteikia papildomas galimybes, pvz.: save-dokumentuojančią išraišką kaip ``Employee(name='Jonas', title='programuotojas')``.

vardų erdvė (Ang. namespace)

Vieta, kur laikomi kintamieji. Vardų erdvės yra įgyvendintos kaip žodynai. Egzistuoja lokali, globali ir įtaisytoji erdvės bei vidinė erdvė objektuose (arba metoduose). Vardų erdvės suteikia moduliškumą, kuris padeda išvengti vardų konflikto. Pavyzdžiui funkcijos :func:`__builtin__.open` ir :func:`os.open` gali būti atskirtos pagal vardų

erdvę. Vardų erdvės taip pat padeda skaitomumui ir priežiūrai aiškiai parodydamos kuriam moduliui priklauso funkcija. Pavyzdžiui, užrašai :func:`random.seed` ir :func:`itertools.izip` aiškiai parodo, kad šios funkcijos yra įgyvendintos :mod:`random` ir :mod:`itertools` moduliuose atitinkamai.

virtualioji mašina

Kompiuteris apibrėžtas programine įranga. Python'o virtualioji mašina vykdo :term:`baitinį kodą <baitinis kodas>`, sugeneruotą baitinio kodo kompiliatoriaus.

žodynas (Ang. dictionary)

Asociatyvus masyvas, kur raktai yra atvaizduoti į reikšmes. Klasės :class:`dict` naudojimas labai panašus į klasės :class:`list` naudojimą, bet raktais gali būti bet kokie objektai turintys :meth:`__hash__` funkciją, ne tik sveiki skaičiai.