PYTHON RECEPTAI

Praktiniai Python kodo pavyzdžiai kiekvienai programavimo dienai

autorius

ŠARŪNAS NAVICKAS

Griaustinis.lt

Kaunas 2021

Turinys

1 Pradmenys **5**
- 1.1 Funkcijos . . . 5
- 1.2 Sveikas pasauli! . . . 5
- 1.3 “main” funkcija . . . 5
  - 1.3.1 Kodo importavimas . . . 6
  - 1.3.2 Standartinės programos imitavimas . . . 8
- 1.4 Darbas su failais . . . 12
  - 1.4.1 Failo skaitymas . . . 12
  - 1.4.2 Rašymas į failą . . . 12
- 1.5 Darbas su direktorijomis . . . 13
- 1.6 Darbas su CSV . . . 13
  - 1.6.1 Skaitymas iš CSV . . . 13
  - 1.6.2 Rašymas į CSV . . . 14

2 Python Aplinka **15**
- 2.1 VirtualEnvironment . . . 15
  - 2.1.1 Kaip tai veikia . . . 15
  - 2.1.2 Sukūrimas . . . 16
  - 2.1.3 Naudojimas . . . 16
  - 2.1.4 Pabėgimas . . . 16
- 2.2 Pipenv . . . 17
  - 2.2.1 Pagrindiniai pliusai . . . 17
  - 2.2.2 Pagrindiniai minusai . . . 17
  - 2.2.3 Sukūrimas . . . 17
  - 2.2.4 Pabėgimas . . . 18
- 2.3 Poetry . . . 18
  - 2.3.1 Pagrindiniai pliusai . . . 18
  - 2.3.2 Pagrindiniai minusai . . . 19
  - 2.3.3 Instaliavimas . . . 19
  - 2.3.4 Sukūrimas . . . 19
  - 2.3.5 Pabėgimas . . . 20

Skyrius 1

Pradmenys

1.1 Funkcijos

1.2 Sveikas pasauli!

Tekstui išvesti galime pasinaudoti funkcija ‘print‘. Paprastuoju atveju, ji išves kas jai duota į ekraną, t.y. parašys ‘Hello world¡

```
1 print("Hello world!")
```

1.3 “main” funkcija

Šis terminas tikriausiai girdėtas tiems, kurie jau turi patirties su tokiomis kalbomis kaip C, C++ ar Java.

Istoriškai susiformavo standartas pagal kurį yra paleidžiamos programos. Trumpai tariant, operacinė sistema (tas pats galioje tiek Windows, tiek Unix) ieško ‘main‘ funkcijos ir ją iškviečia. Ta funkcija priima begalinį (na beveik begalinį) kiekį argumentų ir gražina sveiką skaičių. ‘0‘ visuotinai reiškia, kad programa baigė darbą sėkmingai, visi kiti - klaidos pranešimo kodas.

Įvairios kalbos bandė įvairiai supaprastinti šį procesą, pvz.:

(a) nereikia gražinti skaičiaus, jeigu programa "nenulūžo", tai automatiškai 0, priešingu atveju gražinamas atitinkamas klaidos kodas

(b) jeigu neketinama nieko daryti su perduotais argumentais, galima jų nepaminėti aprašyme

Vienaip ar kitaip, tai neišvengiamas žingsnis.

Tačiau, kalbose kurios yra interpretuojamos (Python, Ruby, JavaScript, Lua ir t.t...), tikroji programa yra interpretatorius ir būtent jis vargsta su ta ‘main‘ funkcija, o mūsų kode to visai nebereikia.

Taigi Python atveju, galima tiesiog rašyti:

```
print("Vienas")
print("Du")
print("Trys")
```

Kodas bus skaitomas nuo viršaus į apačią, iš kairės į dešinę ir atitinkamai vykdomas. Jokio ‘main‘ čia nereikia.

Tačiau, toks supaprastinimas atneša aibę savų problemų, kurias pajuntame peržengę pradedančiųjų lygį.

1.3.1 Kodo importavimas

Tarkime turime tokį kodą:

```
import random


def super_advanced_generator(n):
    return 42


print(
    "O, labas! Atsakymas į rūpimą klausimą yra: {atsakymas}"
    .format(atsakymas=super_advanced_generator(random.randint(0, 100)))
)
```

Ir mes norime panaudoti tą ‘super_advanced_generator‘ kažkur kitur.

Problema: kiekvieną kartą mes gausime bereikalingą tekstą ‘O, labas!<...>‘

Sprendimas: ‘print‘ dalį įkelti į funkciją

```
import random


def super_advanced_generator(n):
    return 42


def main():
    print(
        "O, labas! Atsakymas į rūpimą klausimą yra: {atsakymas}"
        .format(atsakymas=super_advanced_generator(random.randint(0, 100)))
    )
```

Dabar mes galime importuoti be problemų, tačiau, nieks nekviečia mūsų ‘main‘ funkcijos kai tiesiog paleidžiame failą?

Taigi, galiausiai prieiname prie būdo iškviesti ‘main‘ metodą Pythone.

```
import random


def super_advanced_generator(n):
    return 42


def main():
    print(
        "O, labas! Atsakymas į rūpimą klausimą yra: {atsakymas}"
        .format(atsakymas=super_advanced_generator(random.randint(0, 100)))
    )


if __name__ == "__main__":
    main()
```

Ir galiausiai tai turėtų veikti!. Tačiau labai tikėtina, kad niekas čia nėra aišku.

Kintamieji ir metodai su ‘___‘ pradžioje ir pabaigoje (dažnai vadinami “dunder”, kas yra “double under” sutrumpinimas), yra Python “magiški” - juos naudoja pats Python ir dažniausiai tikimasi, kad pats vartotojas prie jų nekiš nagų. Aišku, šiais laikais, nagus kišti tenka ir gan dažnai, orginali intencija kiek išblukus.

Taigi grįžtant prie temos, ‘___name___‘ skirtas gražinti modulio pavadinimą. Dažniausiai, modulio pavadinimas yra failo pavadinimas, tik be ‘py‘ galūnės. Tačiau, pirmasis failas kurį paleidžiame yra ypatingas, jis gauna ‘___main___‘ pavadinimą. Turint tokias žinias, sukonstruojame galutinį kodą.

1.3.2 Standartinės programos imitavimas

Taigi, šis poskyris bus labai mielas širdžiai ir artimas tiems, kurie praleido daug laiko Unix terminale, dažniausiai su Linux, arba *BSD, ir tikriausiai tolimas Windows vartotojams.

Trumpai tariant - kiekviena programa gali priimti N argumentų, tais argumentais mes nurodome kaip programa turės veikti. Dar vienas įdomus bruožas - vienos programos rezultatas gali būti kitos programos argumentas, Unix terminologijoje tai vadinama ‘pipeline‘, arba tiesiog ‘pipe‘.

Pvz.:

```
cat tekstas.txt | head -n 10 | grep 'labas'
```

Kas čia įvyko:

1. su ‘cat‘ komanda nuskaitėme ir išvedėme visą failo ‘tekstas.txt‘ turinį į ekraną

2. su ‘head‘ komanda nuskaitome tik pirmas eilutes, ‘-n 10‘ pasakome, kad norime pirmų 10 eilučių

3. su ‘grep‘ ieškome eilučių turinčių žodį ‘labas‘ savyje

Panašiai galime įsivaizduoti ir savo programėlę.

Bazinį veikimą galime atkartoti štai taip:

```
1  import sys
2
3
4  def main(args):
5      print(args)
6
7
8  if __name__ == "__main__":
9      main(sys.argv)
```

Iškviečiam mūsų programą:

```
> python examples/1_3_2_main.py hello world
> ['examples/1_3_2_main.py', 'hello', 'world']
```

Pirmas argumentas yra pats failas (tokia konvencija), toliau seka perduoti argumentai (atskiriami tarpu), taigi gavome ‘hello‘ ir ‘world‘ kaip argumentus.

Tačiau, jeigu pradėtume kurti kiek labiau sofistikuotą programėlę, kiltų begalės problemų:

(a) Argumentų tvarka gali keistis

(b) Kai kurie argumentai gali būti naudojami pasirinktinai, pvz.: ‘-n 10‘ - mums reikia suprasti imti porą ‘(’-n’, 10)‘ kartu

(c) Kartais nurodome tik argumento pavadinimą neduodami reikšmės (angliškai tai vadinama ‘flag‘, lietuviškai gan šmaikščiai skamba ‘vėlevėlės‘, arba ‘iškelti vėlevėlę‘).

Ir aišku begalės kitų situacijų. Tikrai ne pirmas ir ne paskutinis atvejis, taigi tam yra standartinis sprendimas: ‘ArgParse‘

```
import argparse


def main(args):
    print(args)

    if args.kreipinys:
        print("Sveiki, {kreipinys} {vardas}".format(
            kreipinys=args.kreipinys,
            vardas=args.vardas
        ))
    else:
        print("Sveiki, {}".format(args.vardas))


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("vardas", help="Jūsų vardas")
    parser.add_argument(
        "--kreipinys",
        help="Kaip į Jus kreiptis? (ponas, ponia, panelė?)"
    )
    main(parser.parse_args())
```

Jeigu paleidžiame be jokių argumentų:

```
> python examples/1_3_2_main_w_argparse.py
> usage: 1_3_2_main_w_argparse.py [-h] [--kreipinys
↪  KREIPINYS] vardas
  1_3_2_main_w_argparse.py: error:
  the following arguments are required: vardas
```

Gauname klaidos pranešimą, kad ‘vardas‘ yra būtinas argumentas
Jeigu paprašome pagalbos (dažniausiai tai yra ‘-h‘, arba ‘–help‘)

```
> python examples/1_3_2_main_w_argparse.py -h
> usage: 1_3_2_main_w_argparse.py [-h]
  [--kreipinys KREIPINYS] vardas

  positional arguments:
    vardas                Jūsų vardas

  optional arguments:
    -h, --help            show this help message and
    ↪  exit
    --kreipinys KREIPINYS
      Kaip į Jus kreiptis? (ponas, ponia, panelė?)
```

Ir galiausiai deramai panaudojus:

```
> python examples/1_3_2_main_w_argparse.py "Testas
↪  Testinis"
> Namespace(kreipinys=None, vardas='Testas Testinis')
  Sveiki, Testas Testinis

> python examples/1_3_2_main_w_argparse.py "Testas
↪  Testinis"
  --kreipinys "pone"
> Namespace(kreipinys='pone', vardas='Testas Testinis')
  Sveiki, pone Testas Testinis
```

Yra ir daugiau triukų - nustatyti argumentų tipus (pvz. leisti tik skaičius), reikšmes pagal nutylėjimą ir t.t... Bet čia jau dalis kito recepto.

1.4 Darbas su failais

1.4.1 Failo skaitymas

Paprastas skaitymas atrodo maždaug taip:

```
with open("some_file.txt", "r") as f:
    result = f.read()

print(result)
```

Kelios įdomios vietos:

- ‘with open(...) as f:‘. Alternatyva būtų daryti:

```
f = open("some_file.txt", "r")
result = f.read()
f.close()
```

Esminis skirtumas, kad antruoju atveju turime nepamiršti padaryti ‘f.close()‘. Rodos menka problema. **Tačiau** jeigu skaitant failą įvyks problema - ‘close()‘ bus niekada neiškviestas, arba mes galime elementariai pamiršt iškviesti. Viso to rezultatas gali būti ganėtinai keistas. ‘with open(...)‘ pasirūpina, kad mums nereiktų sukti galvos.

- Ta ‘r‘ raidė. Raidė perduoda operacinei sistemai mūsų ketinimus. Pagrindiniai:

  (a) ‘r‘ - (nuo žodžio read) skaitysime iš failo

  (b) ‘w‘ - (nuo žodžio write) rašysime į failą

  (c) ‘a‘ - (nuo žodžio append) pridėsime prie failo, t.y. su ‘w‘ mes rašome naują turinį, su ‘a‘ mes išlaikome seną ir failo pabaigoje pridedam naujo turinio

  (d) variacijos su ‘b‘ raide: ‘rb‘, ‘wb‘ ir t.t... - (nuo žodžio binary) tie patys rėžimai, tik ne teksto, o binariu rėžimu

1.4.2 Rašymas į failą

Viskas labai panašu kaip ir su skaitymu iš failo:

```
with open("some_file.txt", "w") as f:
    f.write("Daug ilgo teksto")
```

Visi aukščiau aprašyti dalykai galioja ir rašymui, todėl nesikartosiu.

1.5 Darbas su direktorijomis

Kartais mums reikia nuskaityti visus failus iš direktorijos

```
from glob import glob


def read_file(fname):
    with open(fname, "r") as f:
        return f.read()


for fname in glob("dokumentai/*.txt"):
    print(read_file(fname))
```

Naudodami funkciją ‘glob(...)‘ mes pasakome kaip reikia ieškoti. Bazinė sintaksė paprasta - ‘*‘ reiškia, kad ten gali būti bet kas:

‘folderio_pavadinimas1/folderio_pavadinimas2/*‘ galime nurodyti kelią, o jeigu reikia grįžti viena (arba daugiau) direktorija atgal:

‘../folderio_pavadinimas/*.pdf‘, šiuo atveju ‘../‘ reiškia žengti viena direktorija atgal. Galiausiai ‘*.pdf‘, nes dažniausiai mes norime filtruoti pagal failo galūnę. Apie failų skaitymą daugiau rasite 1.4.1 recepte.

1.6 Darbas su CSV

Šis receptas yra glaustai surištas su 1.4, taigi pirma peržvelkite jį.

1.6.1 Skaitymas iš CSV

Tai nėra pats paprasčiausias būdas, bet mano nuomone, pats tinkamiausias:

```
import csv


with open("data.csv", "r") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row)
        print(row["id"])
        print(row["name"])
```

Esminė gera savybė - duomenys iškarto yra suskirstyti pagal stulpelius (pirma eilutė nusako stulpelių pavadinimus). Norimo stuleplio reikšmę galime lengvai gauti pasinaudojus laužtiniais skliaustais ir stulpelio pavadinimu, kaip pvz. ‘row[”name”]‘ kiekvienai eilutei gražintų stuleplio pavadinimu “name” reikšmes

1.6.2 Rašymas į CSV

Rašymas gan panašiai:

```
import csv


with open("data.csv", "w") as f:
    fields = ["id", "name", "comment"]
    writer = csv.DictWriter(f, fieldnames=fields)

    writer.writeheader()

    writer.writerow({
        "id": 1,
        "name": "testas",
        "comment": "Nieko gero"
    })

    writer.writerow({
        "id": 5,
        "name": "testas",
        "comment": "Viso gero"
    })
```

Esminis skirtumas - turime iškarto pasakyti stulpelių pavadinimus ir nepamiršti pirma iškviesti ‘writeheader‘, o tik po to rašyti ‘writerow‘

Skyrius 2

Python Aplinka

2.1 VirtualEnvironment

Kiekvieną, pirmą kartą dirbantį su Python, ši vieta kiek glumins. Taigi kas tai ir kam to reikia?

Pradėkim nuo problemos. Pilnai įmanoma visas pakuotes įrašinėti globaliai, ir iš pirmo žvilgsnio, tai netgi visai nebloga mintis. Ilgainiui atsiranda šios problemos:

(a) Kaip žinoti kokių pakuočių reikia šiam projektui? (Turint omeny, kad vienam kompiuteryje jie gali būti keli)

(b) Ką daryti, jeigu dviem projektams reikia tos pačios pakuotės, bet konkrečių (skirtingų) versijų?

Žinau, kad tai neskamba kaip milžiniškos problemos, bet patikėkit, įmanoma sugaišti valandų valandas prie to.

2.1.1 Kaip tai veikia

"Po kapotu" yra sukuriama virtuali Python instaliacija. Tam, kad sutaupyti laiko ir vietos, failai yra ne kopijuojami, o viskas kas įmanoma, yra padengta 'symbolic links' magija.

'Symbolic link' yra terminas naudojamas Unix sistemose (turintis atitikmenį ir Windows), jo metu yra sukuriami mažyčiai failai kurie tėra tik nuorodos į tikrus. Įdomumas tame, kad operacinei sistemai tiek tikras failas, tiek nuoroda, yra vienas ir tas pats.

Pakuotėms sukuriama atskira direktorija ir ten jos jau normaliai instaliuojamos. Jeigu kuri nors pakuotė jau buvo instaliuota globaliai, vėl bus pagudraujama ir išvengta papildomo instaliavimo.

Operacinei sistemai atrodo, kad kiekvienas projektas turi savo asmeninę Python instaliaciją, jos tarpusavy nesikerta ir veikia nepriklausomai, o realybėje yra panaudota keletas gudrybių kurios leidžia tai atlikti efektyviai.

2.1.2 Sukūrimas

Yra keli būdai sukurti ‘virtualenvironment‘

Pradėsiu nuo pačio ilgiausio (patikimiausio) iki paprasčiausio

```
python3.7 -m venv venv
```

```
python3 -m venv venv
```

Pastaba: ‘venv‘ pakuotė atsirado kartu su Python 3.6 versija. Turint 3.6 ir naujasnę, nieko papildomai įrašinėt nereikia. Kitoms versijoms reikia įrašyti pakuotę ‘virtualenv‘(‘pip install –user virtualenv‘) ir vietoj ‘venv‘ rašyti ‘virtualenv‘.

```
virtualenv venv
```

Pirmuoju atveju užtikrinom konkrečią versiją, antruoju mums tiesiog svarbu, kad tai būtų 3 versija, trečiuoju - leidžiam ir tikimės geriausio. Dažniausiai kyla problemos dėl to, kad turim tiek ‘2.7‘, tiek ‘3.*‘ Python versijas. Naudojant trumpiausią būdą niekad nesam tikri kurią versiją panaudos.

2.1.3 Naudojimas

Linux ir MacOS:

```
source venv/bin/activate
```

Windows:

```
venv\Scripts\activate.bat
```

2.1.4 Pabėgimas

Tiesiog komanda:

```
deactivate
```

2.2 Pipenv

Tai yra alternatyva 2.1 skyriuje aprašytam “VirtualEnvironment”.

2.2.1 Pagrindiniai pliusai

1. Vienodos komandos skirtingoms OS.
2. ‘Pipfile.lock‘ failiukas kuris įšaldo konkrečias versijas 1
3. Visos galimos komandos aiškiai ir paprastai matomos vienoje vietoje:

```
pipenv --help
```

2.2.2 Pagrindiniai minusai

1. “VirtualEnvironment” yra vis tiek sukuriamas, tik šiuo atveju tyliai ir mums apie tai nežinant. Taip, yra standartinės vietos kiekvienoje OS kur jie yra laikomi, tačiau jas rasti nėra taip trivialu. Dalis sudėtingumo nuo mūsų paslėpta, bet tuo pačiu prarandam ir dalį aiškumo.
2. Griežtai įkaltos Python versijos. Jeigu projektas buvo sukurtas su pvz. Python 3.6 versija, tai leidžiant su 3.7 ar 3.8 mums neveiks. Būtų logiška jeigu eitų pasakyti ‘Python >= 3.4‘ ar ‘Python 3.*‘, tačiau taip nėra. Ir autorius griežtai atsisako tai keisti. Gan kontraversiškas sprendimas. Paprasčiausias būdas yra rankomis ištrinti versijos reikalavimą ir tada veiks su visomis, bet dažnai yra poreikis apibrėžti seniausią palaikomą versiją.
3. Tas “Pipfile” ir “Pipfile.lock” mechanizmas dažniausiai veikia, bet retkarčiais pats save įspraudžia į kampą. Tekę ir man turėti atvejų kai tiesiog neįmanoma įrašyti reikiamų bibliotekų, nes pjaunasi versijos tarpusavy. Prie to dirbama, tokių atvejų vis mažėja, bet vis tiek tai lieka validžia kritika.
4. Tam, kad eitų naudotis “pipenv” komanda, turi būti įrašytas pats “pipenv” kaip globali (arba bent jau userio lygyje) pakuotė.

2.2.3 Sukūrimas

Pradedam sukurdami savo projektą (Komanda “–three” liepia sukurti Python 3 versijos projektą):

```
pipenv --three
```

Ir dabar galime instaliuoti pakuotes:

```
pipenv install requests

pipenv install pytest --dev
```

Pirmuoju atveju įrašėme ‘requests‘ biblioteką, antruoju, pasakydami “–dev”, mes nurodome jog šita biblioteka naudinga tik leidžiant development režimu, dažniausiai turint galvoje testavimą.

Norint tiesiog paleisti kodą:

```
pipenv run pytest tests/
```

“pipenv run” galima įsivaizduoti kaip įžengimą į “VirtualEnvironment” trumpam, paleidžiant komandą ir iš ten pabėgant.

Norint užsibūti kiek ilgiau, galima daryti:

```
pipenv shell
```

Šitam papuolame į realų “VirtualEnvironment” ir galioja viskas, kas aprašyti 2.1 skyriuje.

2.2.4 Pabėgimas

Tiesiog:

```
exit
```

2.3 Poetry

Tai yra alternatyva 2.1 skyriuje aprašytam “VirtualEnvironment’ ir glaudžiai siejasi su 2.2, todėl dauguma pliusų ir minusų persidengia. Bet siekiant neblaškyti žmonių atėjusių paskaityti konkrečiai tik apie Poetry, pasikartosiu.

2.3.1 Pagrindiniai pliusai

1. Vienodos komandos skirtingoms OS.
2. ‘poetry.lock‘ failiukas kuris “įšaldo” konkrečias versijas 1
3. Visos galimos komandos aiškiai ir paprastai matomos vienoje vietoje:

```
poetry --help
```

4. Priešingai nei Pipenv 2.2, galima nurodyti reikalavimus Python versijai. Veikia tokios išraiškos kaip "python = $\hat{3}$.6", kas pasako, kad tinka viskos Python versijos nuo 3.6.

5. Sukuriamas "pyproject.toml" failas kuris ne tik aprašo reikalavimus pakuotėms, bet aprašo visą projektą. Ir tai nėra "Poetry" naujądaras, tai yra paties "Python" apibrėžtas naujas būdas aprašyti projektams. Jis visiškai pašalina poreikį senam geram "setup.py".

2.3.2 Pagrindiniai minusai

1. "VirtualEnvironment" yra vis tiek sukuriamas, tik šiuo atveju tyliai ir mums apie tai nežinant. Taip, yra standartinės vietos kiekvienoje OS kur jie yra laikomi, tačiau jas rasti nėra taip trivialu. Dalis sudėtingumo nuo mūsų paslėpta, bet tuo pačiu prarandam ir dalį aiškumo.

2. Tam, kad eitų naudotis "poetry" komanda, turi būti įrašytas pats "poetry" kaip globali (arba bent jau userio lygyje) pakuotė.

2.3.3 Instaliavimas

Naujausią gidą visada galite rasti čia
O šiai dienai tai vyksta maždaug taip:
(Unix OS)

```
curl -sSL
↪ https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py
↪ | python -
```

(Windows OS)

```
(Invoke-WebRequest -Uri
↪ https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py
↪ -UseBasicParsing).Content | python -
```

2.3.4 Sukūrimas

Pradedam su komanda init:

```
poetry init
```

ir čia gausite visą klausimyną: koks projekto pavadinimas? Kokia licenzija? Kas yra autorius? ir t.t... Ko pats nedarau - atsisakau pridėti projektui

reikalingas pakuotes interaktyviai (viena iš paskutinių opcijų). Labai jau painu ir mažai naudos.

Pakuotes pridėti galima:

```
poetry add requests

poetry add --dev pytest
```

Pirmuoju atveju įrašėme "requests" biblioteką, antruoju, pasakydami "–dev", mes nurodome jog šita biblioteka naudinga tik leidžiant development režimu, dažniausiai turint galvoje testavimą.

Norint tiesiog paleisti kodą:

```
poetry run pytest tests/
```

"poetry run" galima įsivaizduoti kaip įžengimą į "VirtualEnvironment" trumpam, paleidžiant komandą ir iš ten pabėgant.

Norint užsibūti kiek ilgiau, galima daryti:

```
poetry shell
```

Šitam papuolame į realų "VirtualEnvironment" ir galioja viskas, kas aprašyti 2.1 skyriuje.

2.3.5 Pabėgimas

Tiesiog:

```
exit
```

1

[1]Tai gan painus dalykas, nes iš pirmo žvilgsnio versijas galima nurodyti 'requirements.txt' faile. Šiuo atveju yra 'Pipfile' failas kuriame galima tiesiog nurodyti reikiamas bibliotekas, arba nurodyti versijos reikalavimus (kaip 'requirements.txt' faile darėm), o 'Pipfile.lock' laiko konkrečias versijas. Skirtumas, kad vienu atveju mes nurodom taisykles pagal kurias reikia parinkti versiją, kitu - pasakom su kokiom konkrečiom versijom tikrai veikia. Painus konceptas, bet naudojamas tiek Ruby, tiek NodeJS, tiek krūvoje kitų kalbų. "Pagavus" idėją, kitur tiesiog automatiškai suprantame prasmę.