Distribucija Python aplikacija

Filip, Novački

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Filip Novački

DISTRIBUCIJA PYTHON APLIKACIJA

ZAVRŠNI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Novački

Matični broj: 44531/15–R

Studij: Informacijski sustavi

DISTRIBUCIJA PYTHON APLIKACIJA

ZAVRŠNI RAD

Mentor :

Doc. dr. sc. Marcel Maretić

Varaždin, rujan 2019.

Filip Novački

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Rad opisuje problematiku distribucije gotovih aplikacija napisanih u programskom jeziku `Python`. Obrađeni su danas aktualni kanale i oblici distribucije raznih oblika aplikacije (aplikacije u konzoli, aplikacije sa grafičkim sučeljem, web aplikacije i dr.) do krajnjeg korisnika. Poseban naglasak stavit će se na `GNU/Linux` operativne sustave (debian, arch). U radu je izrađen prototip jednostavne aplikacije i primjerom pokazani različiti načini distribucije.

Ključne riječi: Python; distribucija; pip; virtualenvvenv; setup; pypi; pypa

Sadržaj

Poglavlje 1

Uvod

`Python` je programski jezik koji je uveo mnogo noviteta u svijet programiranja i s vremenom je prebrodio sve nedostatke koje imao i etablirao se u svijetu strojnog učenja, znanosti, edukacije, weba i mnogo drugdje. Zbog toga što je `Python` interpreterski jezik, jako ga je dobro prihvatila zajednica otvorenog kôoda jer se sve što se izvršava može i pročitati. Ta fleksibilnost je dovela i do toga se mnogi članovi zajednica okušavaju u automatiziranju brojnih procesa što se tiče održavanja `Pythona`, pa tako i instalacija i distribucija paketa.

S obzirom da u početcima nije bilo standarda koji će to kontrolirati, nastala je situacija koja podsjeća na džunglu - isprepleteno, divlje, neorganizirano itd. Imajući to u vidu, stvorena je `PyPA` - radna skupina koja se brine o razvijanju `Pythona` te ponajviše distribuciji softvera. Ovaj će rad pokriti metode instalacije i distribucije paketa u `Pythonu` te pokazati koje su sve mogućnosti paketa koji služe za tu namjenu.

Svi opisani procesi testirani su i izvedeni na `Linux/GNU` operacijskom sustavu distribucije `Arch` verzije 5.0.0. i za `Python` verzije 3.7. Unatoč tome, svi će primjeri vrlo vjerojatno raditi i na svim ostalim `Linux/GNU` operacijskim sustavima, a nespecifične `bash` naredbe radit će i na drugim operacijskim sustavima (npr. `Windows`).

O terminologiji

Izraz paket (en. *package*) inače se koristi kao termin koji označava jednu biblioteku ili sličnu jedinicu. U kontekstu distribucije aplikacija bi odgovarajući termin bio *distribucija* koja podrazumijeva skup paketa, no s obzirom da se koristeći izraz *distribucija* često misli na distribucije `Linux/GNU` operacijskih sustava, adekvatan izraz koji se koristi u tu svrhu ipak će biti paket. [17]

Osnovni pojmovi

Za razumijevanje rada potebno je ukratko objasniti osnovne pojmove [16] koji će se spominjati u radu kako bi čitatelju bilo lakše pratiti tekst. Važno je napomenuti kako je mnogo nazivlja osmišljeno kao referenca na *Monty Python* seriju koja se prikazivala na britanskoj televiziji `BBC` od 1969. uz prekide do 2014.

`PyPA` ili `Python Packaging Authority` - radna skupina koja se brine o projektima koji služe za distribuciju softvera (`pip`, `setuptools` i općenito način distribu-

cije). Sami sebe ponekad zovu i *Ministry of Installation*, prema skici iz *Montyja Pythona The Ministry of Silly Walks*

`PyPI` je repozitorij softvera za projekte pisane u `Pythonu`, služi svim programerima za distribuciju te korisnicima za pretraživanje i preuzimanje paketa. Često se pod `PyPI` kaže i `The Cheeseshop` prema istoimenoj skici iz serijala *Montyja Pythona*

`PEP` ili `Python Enchancements Proposals` - dokument koji daje zajednici uvid u nove funkcionalnosti `Pythona`, mehanizme procesa, tehničke specifikacije mogućnosti i sl. Podijeljen je na brojeve i svaki broj detaljno opisuje jednu temu

`pip` je instalater `Python` paketa s `PyPI` ili nekih drugih repozitorija ili lokalno iz `wheel` datoteka

`egg` je format za distribuciju `Python paketa`, često se kaže da je `egg` za `Python` što je `jar` za `Javu`. Danas se više ne održava te ga mijenjaju `wheel` datoteke

`wheel` je također format za distribuciju `Python` paketa, mijenja `egg` format, a ima brojne prednosti kao što su brzina, sigurnost, olakšano razvijanje, konzistentnije kroz više platformi itd.

Poglavlje 2

Alati i tehnike

Kako bismo mogli dublje ući u tematiku distribucije `Python` aplikacija prvo se moramo upoznati s osnovnim alatima koji će nam koristiti u tu svrhu, a za razumijevanje teksta moramo definirati i neke osnovne pojmove.

2.1 Paket `setuptools` (i njegov prethodnik `distutils`)

Paket `distutils` služi za razvijanje i instalaciju dodatnih modula u `Python`. On podržava module koji su programirani u `Pythonu`, `C`-u te oni koji su kombinacija obojega. U praksi se `distutils` ne koristi sam po sebi, već se koristi `setuptools` - poboljšana verzija paketa `distutils`.

`Setuptools` ima mnogo mogućnosti koje olakšavaju proces razvijanja i distribucije `Python` aplikacija, jedna od glavnih je ta što može automatski pronaći sve druge biblioteke koje su potrebne za ispravno izvršavanje aplikacije. Ta se funkcionalnost izvršava koristeći `EasyInstall`, a ima i mnogo opcija za preuzimanje paketa kao što su korištenje `FTP`-a, `HTTP`-a... Ta je funkcionalnost dodatna olakšana time što `setuptools` sam zna prepoznati koji su paketi potrebni za izvršavanje, odnosno nije ih potrebno ručno popisivati u datoteci `setup.py`.

Paket `setuptools` koristan je i kod distribucije aplikacija na različitim operacijskim sustavima. Taj problem uglavnom nastaje kod operacijskog sustava Windows jer u njemu ekstenzije datoteka imaju semantičku važnost i datoteka se ne može ispravno izvršiti ako nema ispravnu ekstenziju (npr. `.exe` i sl.). [1]

Pokretanjem Pythona nad datotekom koja u sebi ima `setup()` metodu će se automatski napraviti distribucija koja će uključiti sve pakete koji žive u `Python` skriptama koje žive u tom direktoriju.

Kad `pip` naiđe na neku `setup.py` datoteku, on ju uvijek pokreće pomoću `setuptools` alata. Naravno, kad projekt kojim se upravlja krene rasti i postane nešto više od samo skripte `setup.py`, praktično bi bilo i uključiti neka dodatna svojstva koja nam pruža `setuptools`.

Pomoću paketa `setuptools` možemo i stvoriti `wheel` datoteke koje nam služe za *offline* instalaciju. To nam može biti osobito korisno ukoliko želimo distribuirati paket preko drugih kanala osim repozitorija `PyPI`.

2.2 Paket `pip`

Paket `pip` alat je koji služi za instalaciju drugih `Python` paketa. U dokumentaciji autori upozoravaju da instalacijom `Pythona` sa službenih *web* stranica `pip` se instalira sam zajedno s `Pythonom`, a da ukoliko smo koristili repozitorije nekih Linux distribucija, moguće je da ćemo ga morati ručno instalirati. Unatoč tome, praksa je pokazala kako je `pip` obično instaliran i kad se koriste razne `Linux` distribucije. Nakon instalacije, `pip` se može pozvati iz konzole pomoću istoimene naredbe. Na primjer, paket `SciPy` instalira se moću naredbe:

```
$ pip install scipy
```

Moramo obratiti pozornost na to da je za instalaciju paketa potrebno imati administratorske ovlasti. To se događa iz razloga što je `Python` obično instaliran na lokaciji `/usr/bin/` u kojeg ne smije bilo tko pisati. Detaljnije o toj temi bit će opisano u poglavlju Lokalne i izolirane kopije `Pythona`. Ukoliko `pip` nije dodan na putanju operacijskog sustava, može se i pozvati preko `Pythona` naredbom:

```
$ python -m pip
```

Osim ručno, `pip` može instalirati i pakete koji su zapisani u datoteci što će nam biti osobito zanimljivo u kontekstu ovoga rada. Datoteka s popisom paketa se postavlja kao argument i suštinski se ne razlikuju od ručnog ispisivanja svih potrebnih paketa. Naravno, koristimo računala i cilj nam je automatizirati što je toga više moguće.

Ukoliko želimo instalirati neke pakete *offline*, možemo pokrenuti `pip install` naredbu nad `wheel` datotekom. `Wheel` datoteka u sebi sadrži *samo sebe*, odnosno ne sadrži module koji se uvjetuju kod instalacije tog `wheel` modula. Kako to ne bi bio velik problem, preuzimanjem paketa korištenjem `pip download` preuzimaju se svi paketi (tražen paket i njegovi uvjeti) u `.whl` formatu tako da se kasnije mogu instalirati pojedinačno.[20]

2.2.1 Datoteka `requirements.txt`

Kad razvijamo neku aplikaciju, korištenjem naredbe `freeze` možemo sve potrebne pakete zapisati u datoteku te kasnije lako iskoristiti tu datoteku za instalaciju.

```
$ pip freeze > requirements.txt
$ cat requirements.txt
docutils==0.11
Jinja2==2.7.2
MarkupSafe==0.19
Pygments==1.6
Sphinx==1.2.2
```

Instalacija iz datoteke funkcionira na sličan kao i pojedinačnih paketa. Jedina je razlika što `pip` mora znati da kao argument prima datoteku, a ne imena paketa. U datoteci `requirements.txt` možemo specificirati i operatore za verzije paketa koje želimo instalirati.

```
$ pip install -r requirements.txt
```

Datoteka `requirements.txt` je vrlo fleksibilna i podržava mnogo opcija. Jedna od važnijih je specificiranje verzija potrebnih za pakete. Operatori za verzije su uglavnom univerzalni, odnosno prepoznatljivi bez puno kompliciranja:

== - potrebna je točna verzija koja je napisana

<= - potrebna je verzija manja od zadane

>= - potrebna je verzija veća od zadane

~= - potrebna je verzija blizu zadane (npr. ako je verzija ~= 2.3.5, onda su prihvatljive verzije `2.3.X`, gdje je svejedno koji je `X`

!= - ne smije biti zadana verzija

Paket možemo specificirati pomoću nekoliko operatora, te ih odvojiti zarezima. ti će se operatori tumačiti inkluzivno, npr.

```
urllib3>=1.25.3, <=0.9, *=1.1 --hash=sha256:b246607a25ac(...)
        024ac94435cabe6e18d1
```

Ukoliko želimo neke posebne opcije kod instalacije, možemo ih dodati nakon verzije, npr. ako ne želimo kompajliranje paketa, odnosno želimo li dodati zastavicu `--no-compile` potrebno je dodati `--install-option="--no-compile"` na kraj reda u datoteci `requirements.txt`. [1]

```
urllib3>=1.25.3, <=0.9, *=1.1
        --hash=sha256:b246607a25ac(...)024ac94435cabe6e18d1
        --install-option="--no-compile"
```

U datoteku `requirements.txt` možemo dodati referencu na neku drugu datoteku s uvjetima tako da jednostavno napišemo `-r other_requirements.txt`.

Osim instalacije, `pip` možemo koristiti i za deinstalaciju (većine) paketa. Slično kao i `install` naredba, `uninstall` može deinstalirati pojedinačne pakete ili one koje su zapisani u datoteci koju dobijemo kao rezultat naredbe `freeze`.[20]

2.2.2 Preuzimanje i *hash* sume

Preuzimanje podataka koristeći internet oduvijek je predstavljalo određenu prijetnju, a zbog mrežne infrastrukture i arhitekture lako može doći do malih promjena u datotekama te neželjenih nuspojava. Kako bi se osiguralo da su svi preuzeti paketi dobro preuzeti, te da nema nikakvih interferencija u datotekama, `pip` ima naredbu `hash` koja računa *hash* sumu preuzete datoteke. Bez dodatnih opcija koristi se algoritam SHA-256, a podržani su i SHA-384 te SHA-512. Naredba `hash` ima dobru sinergiju s `download` naredbom.

Paket `requests` preuzima se naredbom:

[1] Zapis za jedan paket bi trebao biti u jednom redu, ali je zbog preglednosti u pisanom formatu odvojen novim redom

```
$ pip download requests
```

Rezultat preuzimanja su sve `wheel` datoteke koje su potrebne za rad paketa `requests` (uključujući i one koje paket `requests` uvjetuje za rad):

```
$ ls -s
total 556K
156K certifi-2019.6.16-py2.py3-none-any.whl
132K chardet-3.0.4-py2.py3-none-any.whl
 60K idna-2.8-py2.py3-none-any.whl
 60K requests-2.22.0-py2.py3-none-any.whl
148K urllib3-1.25.3-py2.py3-none-any.whl
```

Kako bi se provjerila autentičnost sadržaja tih datoteka možemo pokretati `hash` funkciju nad njima.

```
$ pip hash urllib3-1.25.3-py2.py3-none-any.whl
urllib3-1.25.3-py2.py3-none-any.whl:
--hash=sha256:b246607a25ac80bedac05c6f282e(...)024ac94435cabe6e18d1
```

Zanimljivo je što istu provjeru možemo dobiti i koristeći `Linux` naredbe:

```
$ sha256sum < urllib3-1.25.3-py2.py3-none-any.whl
b246607a25ac80bedac05c6f282e(...)024ac94435cabe6e18d1  -
```

no taj način provjere nam nije osobito praktičan jer `pip` formatira zapis način da ga možemo upotrijebiti u datoteci `requirements.txt`.

Kako bismo ustanovili autentičnost paketa, *hash* sumu moramo usporediti s očekivanom sumom paketa. Tu sumu možemo vidjeti na *web* stranicama repozitorija paketa, npr. `https://pypi.org/` ili možemo iskoristiti zastavicu `--require-hashes`. Kroz nedavnu prošlost bilo je i nekoliko pokušaja integriranja *hash* suma kroz naredbu `pip freeze` na način da se odmah izračuna *hash* suma tih paketa, no to se i dalje nije ostvarilo. Međutim, Erik Rose je osmislio paket `peep` koji uključuje tu funkcionalnost. Cijeli sustav funkcionirao je tako da se u određenom trenutku kad aplikacija radi izračuna *hash* suma svih paketa te da se dodaju ispred paketa kojeg je potrebno instalirati. Takva datoteka bi izgledala:

```
# sha256: L9XU_-gfdi3So-WEctaQoNu6N2Z3ZQYAOu4-16qor-8
Flask==0.9

# sha256: qF4YU3XbdcEJ-Z7N49VUFfA15waKgiUs9PFsZnrDj0k
Jinja2==2.6
```

Potencijalni nedostatak je što taj način provjere nije automatiziran, već bi korisnik sam trebao izračunati *hash* sume i potom ih dodati u datoteku ispred naziva paketa za instalirati.

Paket `peep` se više ne održava jer je ta funkcionalnost ugrađena u `pip` od verzije 8, no i dalje ne podržava više automatizacije od ovoga što je pružao `peep`.

Danas bi se *hash* sume izračunavale na opis način, samo što ne možemo izravno izravno proslijediti rezultat naredbe `hash` u `requirements.txt`, već moramo formatirati i onda pripasati ispravan *hash* određenom paketu. Formatirati zapis se može tako

da se uzme samo drugi red rezultata naredbe `pip hash` pa bi se onda izvršila malo izmijenjena naredba:

```
$ pip hash urllib3-1.25.3-py2.py3-none-any.whl | sed -n '2{p;q}'
--hash=sha256:b246607a25ac80bedac05c6f282e(...)024ac94435cabe6e18d1
```

Taj rezultat sad je samo potrebno dodati u datoteku `requirements.txt` iza odgovarajućeg paketa. Konkretno, za `urllib3` zapis bi izgledao ovako:

```
urllib3==1.25.3 --hash=sha256:b246607a25ac(...)024ac94435cabe6e18d1
```

S ovako postavljenom datotekom `requirements.txt` instalacija svih potrebnih paketa obavila bi se naredbom: [20]

```
$ pip install --require-hashes -r requirements.txt
```

2.2.3 Razine instalacije paketa

`Pip` podržava rad nad `Pythonom` nad nekoliko shema:

- sistemski
- korisnički i
- lokalni.

Sistemski je onaj koji uglavnom dolazi predinstaliran s većinom distribucija Linuxa i nalazi se na putanji `/usr/bin/`. Korisnici koji nemaju administratorske postavke ne mogu u njega ništa zapisivati, a to se niti ne preporuča iz sigurnosnih razloga. Kad se korisnik ne nalazi u virtualnom okruženju, `Python` kojeg operacijski sustav koristi je sistemski, unatoč tome što bi on zapravo trebao biti korisnički. To je poznata *greška* u radu `Pythona`, no predložene izmjene još od 2008. nisu bile prihvaćene.

Korisnička distribucija `Pythona` se uglavnom nalazi na putanji `~/.local/` [20]

2.3 Konfiguracijske datoteke

Konfiguracijske datoteke su jednostavni način pomoću kojeg se zapisuju sve informacije koju su vezane za projekte, pakete itd. Sve konfiguracijske datoteke su u `plain text` formatu tako da su čitljive iz svih tekstualnih uređivača te se mogu na jednostavan način verzionirati.

2.3.1 `setup.py`

Datoteka `setup.py` sadrži sve podatke koji su važni za distribuciju paketa. U suštini `setup.py` je `Python` skripta koja se može pokrenuti na način kao i sve ostale `Python` skripte, ali ima i poseban značaj. Kod instalacije paketa, `setuptools` ili `pip` traže datoteku `setup.py` te prema njenom sadržaju rade instalaciju. [28]

Kako bi setup.py datoteka mogla instalirati neki paket, potrebno je u njoj pozvati `setup()` metodu. Najjednostavnije rješenje koje će instalirati neki paket izgleda ovako:

```
from setuptools import setup
setup()
```

Paket `pip` još ne može iskoristiti ovakvu skriptu, ali ju možemo pozvati pomoću naredbe

```
$ python -m setup.py install
```

Paket se uspješno instalirao, instalacijske datoteke su uspješno stvorene, samo što nam ovakav paket i ne može puno koristiti:

```
$ pip freeze
UNKNOWN==0.0.0
```

Taj paket je dobio generičko ime i verziju, a u instalaciju nije uključen niti jedan paket. Kako bi aplikacija imala bar nekog smisla, morali bismo joj dati neko ime, verziju i reći joj koje pakete treba instalirati kako bi nešto radila.

```
(...)
setup(name="Hello World",
version="0.2.1",
packages=["HelloWorld"]
)
```

Sada već možemo koristiti i `pip` za instalaciju. Izvršavamo naredbu `pip install ./HelloWorld` i nakon nekoliko trenutaka paket je instaliran. Sad je vrijeme da uđemo malo dublje u naš uradak.

Funkcija `setup()` dio je `setuptools` paketa stoga ga je potrebno na početku uvesti u skriptu. Već predstavljeni argumenti su `name` koji prima objekte tipa `string` i označava ime paketa. taj atribut je *case sensitive*, no to samo znači da će se razlika između malih i velikih slova pospremiti na taj način. Općenito kod instaliranja i deinstaliranja paketa velika i mala slova nemaju veze. Razmaci su dozvoljeni u nazivu, no automatski će se pretvoriti u povlake(`-`) nakon instalacije.[9]

Ukoliko želimo stvoriti `wheel` datoteku, potrebno je dodati argument kod pozivanja `setup.py` skripte `bdist_wheel`: [7]

```
python -m setup sdist bdist_wheel
```

Takav se paket može instalirati koristeći `pip`:

```
pip install Hello-World-0.2.1-py3-none-any.whl
```

Verzije paketa

Atribut `version` označava verziju paketa. Uobičajen format verzija nekog softvera je `X.Y.Z`, gdje su slova `X`, `Y` i `Z` neki brojevi. Broj 0 nikad se ne koristi osim na mjestu `X` jer nema značaj, a kod usporedbe verzija se verzije uvijek implicitno *šire* do tri razine verzije tako da su zapravo `X.Y` i `X.Y.0` iste verzije. Općenito se kaže da verzije `X.Y.Z` spadaju u epohu verzije `X.Y` za bilo koji `Z`. Također postoji i praksa verzioniranja po pravilu `Godina.Mjesec`, no te ne vidi često u `Python` svijetu.

Osim glavnih verzija, postoje i `pre-release` verzije koje još nisu spremne za izdavanje i distribuciju. U svijetu su poznate pod nazivima *alpha*, *beta* i *release candidate* (točno tim redom se označava i njihov stadij razvoja: *alpha* je rani stadij, a `release candidate` je gotovo spreman za izdavanje). Te se verzije označavaju tako da se umjesto zadnje točke ubaci slovo (`a` za *alpha*, `b` za *beta* te `c` ili `rc` za *release-candidate*). Takav zapis izgleda `X.YaZ` ili konkretno `3.3a2`. Kad faze razvoja završe, cijeli taj nastavak nestaje i nastaje verzija `3.3`.

Također postoje i *post-release* verzije (zakrpe) koje se izdaju ubrzo nakon izdavanja, a imaju vrlo male promjene koje ne utječu na rad softvera, ako što su tipfeleri u bilješkama o promjenama softvera (en. *release-notes*). One se formiraju na sličan način kao i *pre-release* verzije, jedino se umjesto jednog slova umjesto točke dodaje `post` nakon točke zajedno s oznakom verzije, npr. `3.3.post1`.

Moguće je i kombinirati verzije *pre-release* i *post-release* na način `X.Yb2.post1` što bi označavalo prvu zakrpu druge *beta* verzije verzije `X.Y`, no to preporuča izbjegavati jer je izrazito nečitko za ljude. [6]

Uključeni paketi

Kako bi `pip` znao koje je pakete potrebno uključiti u instalaciju, moramo mu sugerirati kako će se te mape zvati. To se predaje kao argument pod nazivom `packages` u obliku `string` liste. Kako bi neki paket bio paket, a ne obična mapa, potrebno je u toj mapi imati datoteku `__init__.py`. Ukoliko se naši paketi ne nalaze u podmapama od lokacije gdje je skripta `setup.py`, potrebno je reći gdje se mogu paketi pronaći. U tu svrhu se može zadati rječnik `package_dir`. Dakle, ukoliko imamo paket `foo` u mapi `src`, to možemo definirati ovako:

```
# (...)
setup(
        package_dir={'':'src'},
        packages=['foo'],
)
```

Takvim postavkama obećajemo alatima paketa `distutils` da će moći pronaći datoteku `src/foo/__init__.py`. Sve mape koje su navedene u `package_dir` funkcioniraju rekurzivno, dakle svi paketi unutar tih mapa bit će pregledani. Važno je naglasiti da unatoč tome što će sve mape biti pregledane, pakete moramo navesti u `packages`.

Kao rješenje problema nerekurzivnog instaliranja paketa unutar podmapa postoji funkcija `find_packages` koja će automatski pregledati sve podmape vratiti listu paketa koje pronađe. Ta funkcija može primiti i argument koji će joj reći iz koje mape početi tražiti pakete. Ukoliko imamo paket u mapi `foo`, kao argument pod nazivom `packages` predat ćemo `find_packages('foo')`. Isto tako u `find_packages` možemo predati argument `exclude` ukoliko određene mape ne želimo uključiti u naš paket, npr. `find_packages(exclude=['docs', 'tests'])`.

Funkciju `find_packages` možemo koristiti i bez korištenja `package_dir`. U takvom scenariju možemo funkciji `find_packages` *pomoći* u pronalasku pravih paketa koristeći se argumentom `where`. Ukoliko nam je paket `foo` smješten u mapi `folder-foo`, možemo pozvati `find_packages('./folder-foo')`. S obzirom da funkcija `find_packages` vraća listu, lako možemo nekoliko funkcija (odnosno rezultata funkcija) *zbrojiti* operatorom `+`. [9]

Opis paketa

U datoteci `setup.py` definiraju se i opisi paketa. Ti opisi olakšavaju idntificiranje paketa u repozitorijima `Python` paketa te pomažu pretraživačima u njihovom pronalasku.

Opisivanje se vrši argumentima:

- `description`
- `long_description`
- `long_description_content_type` te
- `keywords`.

Argument `description` je kratki opis paketa koji sadrži manje od dvjesto znakova. Poanta ovog paketa je da na jednostavan i sažet način objasni čemu paket služi i za koja je namijenjen.

Argument `long_description` opširan je opis paketa koji je u `rST` formatu (en. *reStructuredText*). Taj opis sličan je onome kojeg pronalazimo na stranicama repozitorija (općenito repozitorija, poput `GitHub` repozitorija). Taj opis opisuje cijeli projekt u detalje, a praksa je do sada pokazala da su to mjesta u koja se pišu način instalacije, osnovno pokretanje i korištenje, primjeri rada, obavijesti o novim izdanim verzijama i sl.

Argument `long_description_content_type` opisuje format u kojem je predan argument `long_description` ukoliko želimo neki drugi osim `rST` formata. Ova dva argumenta pružaju dobru sinergiju s vanjskim datotekama u kojima se nalazi opis, često nazvan `README`. Koristeći funkciju `read()` za čitanje datoteka možemo smanjiti zatrpanost `setup.py` skripte i povećati njenu čitkost. [9]

Ostalo

Funkcija `setup.py` može primiti brojne argumente i ovaj rad neće pokriti sve, no valja obratiti pozornost na još nekoliko njih.

`Entry_points` su ulazne točke programa i one name olakšavaju ulaz u program kroz određene funkcije, metode, klase itd. Ulazne su točke osobito korisne ukoliko paket nije namijenjen samo za korištenje u sklopu drugih aplikacija, već kao samostalna aplikacija.

Osim grupa koje korisnik definira, postoje i dvije specijalne grupe ulaznih točaka koje imaju već predviđene funkcionalnosti. One su `gui_scripts` i `console_scripts`. Razlika između njih dviju postoje samo za korisnike `Windows` operacijskih sustava stoga se na to ovdje nećemo previše obazirati. Ove specijalne grupe posebne su po tome što se prilikom instalacije one same dodaju u putanju te se tako mogu bez problema pokrenuti iz konzole. Velika prednost tog pristupa je da se korisnik aplikacije ne mora brinuti ni o čemu osim o samom korištenu aplikacije.

Klasifikatori (en. *classifiers*) su predefinirani stringovi koji služe za kategoriziranje aplikacije i njihovo jednostavnije pronalaženje u repozitorijima `Python` paketa. Pomoću klasifikatora mogu se definirati stadij razvoja, ciljana publika, tema, licence, korištena verzija jezika, okruženje u kojem se koristi itd. Popis svih klasifikatora nalazi se na stranicama PyPI-ja.

Licenca označava razinu prava pod kojima se paket smije koristiti, mijenjati, distribuirati itd. Ovo polje je vrlo važno definirati pogotovo ukoliko se paket razvija za zajednicu otvorenog koda kako bi drugi korisnici mogli koristiti i unaprijeđivati paket. [9]

2.3.2 setup.cfg

Kod prilagođavanja aplikacije za instalaciju neke je mogućnosti teško implementirati pomoću datoteke `setup.py` ili to nije poželjno jer se tijekom instalacije žele napraviti neke vlastite preinake. U tim se slučajevima ispunjava datoteka `setup.cfg`. [10]

U toj su datoteci sadržane sve opcije koje bi se inače proslijedile kao argumenti prilikom instalacije. Tako sadržaj datoteke `setup.cfg`

```
[build]
build-base=blib
force=1
```

znači da se kod pokretanja naredbe `build` implicitno pokreće:

```
$ python setup.py build --build-base=blib --force
```

U tu datoteku možemo dodati koliko god želimo opcija i pokriti jako puno slučajeva korištenja. Na primjer ukoliko želimo paket instalirati kao `egg`, a treba nam `data_files` is `setup.py`, moramo koristiti `pip` ili možemo koristiti

```
$ python setup.py install --old-and-unmanagable
```

ili jednostavno dodati u `setup.cfg`: [13]

```
[build]
build-base=blib
force=1
[install]
old-and-unmanagable=1
```

2.3.3 MANIFEST.in

Datoteka `MANIFEST.in` služi za dodavanje drugih datoteka u distribuciju za vrijeme instalacije. Kod distribuiranja inače su uključene datoteke:

- sve `Python` skripte naznačene pomoću `py_modules` i `packages`
- sve `C` datoteke u `ext_modules` i `libraries`
- sve skripte naznačene pomoću `scripts`
- sve `README` datoteke s uobičajenim ekstenzijama (`.txt`, `.rst`, ništa)
- `setup.py`
- `setup.cfg`

- `package_data` i `data_files`

U pozadini se kod instalacije generira datoteka `MANIFEST` (primijetimo da nema ekstenziju) u kojoj se nalazi popis svih datoteka koje će biti distribuirane. Ta se datoteka automatski generira tako da se čita sadržaj iz `MANIFEST.in` ako ona postoji te iz `setup.py`. Mi možemo i sami napisati `MANIFEST` datoteku, no onda moramo paziti da uključimo sve potrebne datoteke. [11]

Ova se datoteka popunjava tako da se u jedan red stavi ime jedne datoteke, a ispred nje riječ `include`. Tako će jedna datoteka izgledati ovako:

```
include data/results.cvs
include data/options.h
include opt_args.dat
```

2.3.4 `README`

`README` datoteka zapravo nije ništa neviđeno u svijetu programiranja - to je isti onaj format koji se pronalazi u gotovo svim opisima softvera, dakle na repozitorijima, kod raznog softvera, raznih multimedijalnih sadržaja itd. Ta se datoteka može ispuniti nekim od `MarkUp` jezika kako bi se mogao rezultat lijepo formatirati i prikazivati na raznim uređajima i stranicama.

2.4 Lokalne i izolirane kopije `Pythona`

Paketi `virtualenv` i `venv` su paketi koji služi za upravljanje izoliranim kopijama `Pythona`. U većini slučajeva `Python` se instalira na mjestu `/usr/bin/python`. `Python` instaliran na toj putanji dostupan je korisnicima računala na čitanje (dakle, i korištenje), ali ne i za zapisivanje. U suštini to znači da svaka instalacija dodatnih paketa očekuje od korisnika da ima administratorske ovlasti, odnosno da je `superuser`, odnosno da ispred `pip install scipy` napiše `sudo` (kratko od *superuser do*). To može predstavljati sigurnosni problem kojih je bilo nekoliko do sada u `pip` i `PyPI` povijesti.

Osim sigurnosti, problem rukovanja nad sistemskom kopijom `Pythona` je i ta što se instalacijom ili deinstalacijom određenih paketa može *potrgati* velik broj aplikacija koje ovise o `Pythonu`, a može se i dogoditi da se poremeti rad sustava za upravljanje paketima u sklopu operacijskog sustava (`pacman`, `yay` i drugi na `Arch` distribucijama, `apt` na `Debian` distribucijama itd.)

Također, sigurnosne je probleme moguće izbjeći korištenjem korisničkog `Pythona` kojem je moguće pristupiti bez administratorskih ovlasti tako da se doda zastavica `--user` kod instalacije paketa.

Kod razvoja `Python` paketa često će nam u interesu biti da korisnik koji preuzima naš paket dobije i odgovarajuću verziju programskog jezika zajedno s paketom kako bi se izbjegle razlike između korisničkog i razvijateljevog `Pythona` u verzijama. Ponekad su te razlike malene (npr. `Python 3.7` i `Python 3.6`), a ponekad velike (npr. `Python 3 i Python 2`), no koliko god one bile velike ili malene, ne želimo kredibilitet naše aplikacije ostaviti na sreći, odnosno očekivanju da korisnik ima potrebnu verziju `Pythona`. [21]

2.4.1 `virtualenv`

`virtualenv` tome doskače tako da kreira izoliranu kopiju Pythona koja se koristi isključivo za projekt za koji je napravljena. Obvezan argument naredbi `virtualenv` je naziv virtualnog okruženja. [2] U primjeru će se napraviti okruženje pod nazivom `myEnv`.

```
$ virtualenv myEnv
Using base prefix '/usr'
New python executable in /home/username/zavrsni/primjer1/myEnv/bin/python
Installing setuptools, pip, wheel...
done.
```

Rezultat ove naredbe stvara strukturu mapa i datoteka nalik na:

```
$ tree -La 3
.
└── myEnv
    ├── bin
    │   ├── activate
    │   ├── activate.csh
    │   ├── activate.fish
    │   ├── activate.ps1
    │   ├── activate_this.py
    │   ├── activate.xsh
    │   ├── easy_install
    │   ├── easy_install-3.7
    │   ├── pip
    │   ├── pip3
    │   ├── pip3.7
    │   ├── python
    │   ├── python3 -> python
    │   ├── python3.7 -> python
    │   ├── python-config
    │   └── wheel
    ├── include
    │   └── python3.7m -> /usr/include/python3.7m
    └── lib
        └── python3.7

6 directories, 16 files
```

Primjećujemo da su svi instalirani paketi na putanji `myEnv/bin/`. Sad kad smo napravili izoliranu kopiju `Pythona`, sad smo ju spremni koristiti. Međutim, nailazimo na problem - i dalje kad bismo koristili `Python` koristili bismo sistemsku distribuciju, a ne onu koju smo upravo napravili. To možemo provjeriti naredbom `which` u `POSIX-compliant` ljuskama - ona nam govori gdje se nalazi naredba koju ćemo izvršavati.

```
$ which python
/usr/bin/python
$ which pip
/usr/bin/pip
```

Tome možemo doskočiti na dva načina:

- tako da svaki put umjesto `python` pišemo `myEnv/bin/python` te umjesto `pip` pišemo `myEnv/bin/pip`
- tako da kažemo operacijskom sustavu da imamo kopiju `Pythona` koju bismo koristili umjesto sistemske verzije

Naravno da prva verzija podrazumijeva mnogo tipkanja i da nije toliko elegantno rješenje. Kako bismo rekli operacijskom sustavu da koristimo kopiju `Pythona` moramo izvršiti `source` naredbu.

```
$ source myEnv/bin/activate
(myEnv) $ which python
/home/username/zavrsni/primjer1/myEnv/bin/python
```

Kad god od sada pokrenemo naredbu `Python`, pokrenut ćemo ga sa zadane lokacije. Kad završimo s radom na tom paketu, potrebno je reći operacijskom sustavu da se korištenjem naredbe `python` ponovno pokreće sistemska distribucija, a ne kopija koju smo stvorili u svrhu projekta.

```
(myEnv) $ deactivate
$ which python
/usr/bin/python
```

Kod korištenja `virtualenva` nailazimo na još jedan problem - veličina distribucije.

```
$ du -sh env/
14M        env/
$
```

Ovaj nam podatak govori kako je samo *goli* `Python` veliki čak 14MB i da to nije uvijek praktično postavljati na poslužitelj za verzioniranje. Dobra praksa preporuča stavljanje cijele `env/` mape u `.gitignore`, a onda svaki korisnik nakon kloniranja repozitorija koristi distribuciju `pythona` kojeg ima na računalu, bila to `python` kopija koja je napravljena za potrebe tog repozitorija ili sistemska distribucija.

2.4.2 `venv`

Razlike između `venva` i `virtualenva`

`venv` je sličan alat `virtualenvu`, samo što `venv` dolazi kao sastavni dio `Pythona`, a `virtualenv` je neslužben alat koji se koristio zbog njegovih funkcionalnosti. `Pyvenv` i `venv` su jako bliski alati, samo što se razlikuje način njihova korištenja.

```
$ pyvenv
WARNING: the pyenv script is deprecated in favour of `python3.7 -m venv`
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT]
            ENV_DIR [ENV_DIR ...]
venv: error: the following arguments are required: ENV_DIR

$ python -m venv
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT]
            ENV_DIR [ENV_DIR ...]
venv: error: the following arguments are required: ENV_DIR
```

Prema poruci koju nam daju alati, `pyvenv` se više ne preporuča koristiti. Umjesto njega bi se trebao koristiti `python -m venv`. `pyvenv` se preporučao koristiti u verzijama `pythona` 3.3 i 3.4, a u 3.5 se preporučuje `venv`. Od verzije 3.6 `pyvenv` izlazi iz upotrebe.

Korištenje `venva`

Većih razlika sa strane korisnika i nema previše između `venva` i `virtualenva`. Sintaksa je gotovo identična.

```
$ python -m venv myEnv
$ tree -L 3
.
└── myEnv
    ├── bin
    │   ├── activate
    │   ├── activate.csh
    │   ├── activate.fish
    │   ├── easy_install
    │   ├── easy_install-3.7
    │   ├── pip
    │   ├── pip3
    │   ├── pip3.7
    │   ├── python -> /usr/bin/python
    │   └── python3 -> python
    ├── include
    ├── lib
    │   └── python3.7
    ├── lib64 -> lib
    └── pyvenv.cfg

6 directories, 11 files
$ source env/bin/activate
(myEnv) $ which python
/home/username/zavrsni/primjer3/myEnv/bin/python
```

```
(myEnv) $ deactivate
$ which python
/usr/bin/python
```

Kreiranje distribucije bez `Pipa`

Osim kreiranja cijele distribucije, `venv` nam omogućuje i da `Pip` ne instaliramo u sklopu kopije, već da koristimo sistemski. Glavna motivacija za taj pristup je da je `pip` projekt odvojen od samog `Pythona` i da ima drugačiji razvojni ciklus i da ga je dobro držati odvojenog od programskog jezika.

```
$ python -m venv --without-pip myEnv
$ tree -L 3
.
└── myEnv
    ├── bin
    │   ├── activate
    │   ├── activate.csh
    │   ├── activate.fish
    │   ├── python -> /usr/bin/python
    │   └── python3 -> python
    ├── include
    ├── lib
    │   └── python3.7
    ├── lib64 -> lib
    └── pyvenv.cfg

6 directories, 6 files
```

Sad kad imamo kopiju bez `pipa` nismo u mogućnosti instalirati nikakve dodatne pakete. Međutim, u tome postoji i jedna velika prednost:

```
$ du -sh myEnv/
40K     myEnv/
```

Bez ostalih alata, ova kopija `Pythona` velika je samo `40KB`. Ukoliko nam ipak zatreba `pip`, možemo ga dodati u distribuciju pomoću naredbe `$ python -m ensurepip`.

Korištenje *symlinkova*

Symlinkovi (kratko od *symbolic link* su tehnologija koja se intenzivno koristi na Linux operacijskim sustavima, a oni služe za pristupanje datotekama koje su na nekom drugom mjestu bez potrebe za kopiranjem cijele datoteke.

Kod kreiranja kopija distribucije *symlinkovi* se koriste kako bi se izbjeglo kopiranje svih datoteka u novi folder. Dakle, kao što smo mogli vidjeti u rezultatima naredbe `tree` u gornjim primjerima `python -> /usr/bin/python` znači da je `python` zapravo *symlink* na sistemski `python`.

Ovdje se spominje problematika *symlinkova* iz razloga što oni nisu (dobro) implementirani na svim operacijskim sustavima. Microsoftova serija operacijskih sustava

Windows ima slabu podršku za *symlinkove* od Windowsa verzije 2000 do 7 u vidu prečaca (*shortcuts*). [26]

Zbog toga postoji opcija u sklopu `venva` da se cijela distribucija stvori uz pomoć *symlinkova* te opcija da se *symlinkovi* ni pod koju cijenu ne koriste. U tu svrhu kao argument je potrebno dodati ili `--symlinks` ili `--copies`.

2.5 Znanstveni paketi

Paketi koji su namijenjeni za znanstvene svrhe uglavnom imaju vrlo složene ovisnosti o drugim modulima i *binaryjima* koji se ne mogu jednostavno instalirati pomoću `Pipa`. Osim ovisnosti, taj se softver često mora izvoditi na nestandardnom hardveru i softveru što dublje komplicira instalaciju. Kako bi se znanstveni paketi instalirali postoji nekoliko metoda. [18]

2.5.1 Kompajliranje iz izvornog kôda

Kompajliranje iz izvornog koda je relativno napredna metoda jer su znanstveni paketi napisani uglavnom u `C`-u i u `Fortranu` kako bi imali dobre performanse kod matematički zahtjevnih operacija.

Ova metoda nije preporučena svim korisnicima jer povezivanje `Fortrana` i `C`-a te njihovih kompajlera je dosta zahtjevan zadatak te se koristi samo u vrlo specifičnim situacijama.

2.5.2 Paketi iz pojedinih distribucija Linuxa

Često razne distribucije u svojim repozitorijima imaju pakete koji su predkompajlirani. U repozitorijima se često mogu naći pod imenom `python-imepaketa` te se instalirati naredbama `sudo apt-get install python-imepaketa` ili `sudo pacman -S python-imepaketa` i slične varijacije.

Potencijalan problem za ovakav način distribucije paketa je taj što korisnici ovise o veličini zajednice i popularnosti paketa. Kako bi se neki paket pojavio na Debianovim repozitorijima, Archevim repozitorijima, `AUR`-u itd. netko ga treba kompajlirati i postaviti na te repozitorije. Uvidom u repozitorije Arch Linuxa na dan pisanja ovog poglavlja (19.08.2019) paket `Scipy` je u repozitoriju na zadnjoj verziji koja je izdana, dok paket `Basemap` nije, već kaska jednu verziju. Prethodna je verzija na repozitorije dospjela točno pet dana nakon što je izašla na stranicama `MatPlotLiba`, a trenutna verzija kasni dvanaest dana. Na taj problem upozoravaju i na stranicama dokumentacije naglašavajući da verzije mogu biti stare i po nekoliko mjeseci.

Drugi je nedostatak na taj način distribuiranih paketa što se jedino mogu instalirati u sistemskoj verziji `pythona`, odnosno ne mogu se staviti u virtualno okruženje. To se može zaobići tako da se distribucijama na računalu omogući pristup sistemskom `pythonu`.

2.5.3 Instalacija `SciPy`-ja

Instalacija gotovih distribucija je najjednostavnija opcija od svih navedenih ovdje jer se u nekoliko koraka automatski instaliraju svi preduvjeti, svi paketi, a ponekad čak i

IDE koji omogućuje olakšano korištenje. Od njih je vjerojatno najpoznatija Anaconda.

Anaconda je distribucija pythona koja je namijenjena primarno za znanstvene svrhe (strojno učenje, duboko učenje, umjetnu inteligenciju itd.) te sadrži velik broj znanstvenih paketa. [19] Anacondu je preporučena čak i u Python dokumentaciji. [12]

Osim Anaconde postoji i još nekoliko distribucija kao što su Enthought Canopy, Python(x,y), WinPython itd. Suštinski su svi oni poprilično slični, a detalji koji ih razlikuju međusobno su uglavnom operacijski sustavi na kojima su podržani, verzije Pythona koje koriste itd. [8]

Paket SciPy se preporuča instalirati i pomoću Pipa. Pod tim se naglašava da se uglavnom radi o osnovnim paketima koji nemaju druge preduvjete te su široko rasprostranjeni. Skupinu paketa koji će biti vrlo vjerojatno ažurni gotovo stalno može se instalirati naredbom

```
$ python -m pip install --user numpy scipy matplotlib ipython
        jupyter pandas sympy nose.
```

2.6 *Hosting* paketa

Kako bi paketi bili dostupni korisnicima koji su povezani internetom, oni se moraju nalaziti na nekom poslužitelju. Ovisno o potrebama korisnika razvijeno je nekoliko različitih poslužitelja i svaki od njih ima svoje prednosti za pojedine korisničke slučajeve. [22]

2.6.1 The Cheeseshop

Jedan od takvih poslužitelja je i The Cheeseshop ili PyPI koji je uglavnom usredotočen na pakete otvorenog kôda. U ovom radu The Cheeseshop nije ništa novo - koristeći pip se automatski preuzimaju paketi s tog repozitorija.

Svi korisnici mogu pretraživati sve pakete i čitati sav kôd koji se nalazi u svakom paketu što je dobro za cijelu zajednicu Pythona koja se bavi razvijanjem *softvera* što sa sobom donosi jednu razinu sigurnosti (barem veću od paketa sa zatvorenim kodom, o problemima više u poglavlju o sigurnosti).

2.6.2 Conda

Conda je repozitorij koja rješava probleme vrlo složenih paketa namijenjenih za znanosti koji se često moraju kompajlirati i teško ih je *natjerati* na rad na raznom softveru i hardveru. Također je mnogo paketa na conda repozitoriju prilagođeno zajednici koja koristi Windows operacijske sustave.

Takvim se pristupom olakšalo korištenje paketa onim ljudima koji nisu skloni razvijanju softvera i možda ne razumiju dublje kako funkcionira Python, kompajliranje raznih programskih jezika i sl.

2.6.3 Vlastiti HTTP server

Kreiranje vlastitog HTTP servera dobra je opcija kad neko poduzeće (pogotovo ako je većeg obujma) ima vlastite biblioteke koje koristi za razvijanje svog softvera i ne želi ga dijeliti s drugima.

`HTTP` server jednostavno se stvara na portu 9000 pomoću naredbe:

```
$ python -m SimpleHTTPServer 9000
```

Nakon što se server pokrene, paketi se s njega instaliraju naredbom:

```
$ pip install --extra-index-url http://127.0.0.1:9000 ImePaketa
```

2.6.4 Distribucija pomoću sustava za verzioniranje

Aplikacije je moguće i instalirati s poslužitelja za verzioniranje kao što su `git`, `svn` i dr. na način da se jednostavno naredbi `pip install` doda putanja do projekta s protokolom preko kojeg želimo preuzeti.

```
git+git://github.com/filipnovacki/linearni-splajn
```

Ovakve je projekte moguće instalirati i u `editable` modu, odnosno da se po učinjenoj promjeni u kodu ne mora ponovno instalirati aplikacija, već da su promjene odmah aktivne. Preuzeti se može koristeći razne protokole, kao što su `HTTP`, `HTTPS`, `SSH` itd. Čak je moguće i iza znaka @ označiti `commit` ili granu s koje želimo preuzeti paket.

Poglavlje 3

Primjena distribuiranog softvera

Softver koji se priprema za distribuciju koristeći opisane alate i tehnike može se koristiti u nekoliko svrha. Za početak je važno razumijevati razliku između modula (en. *module*) i aplikacija.

3.1 Moduli

Moduli, često nazvani bibliotekama, su skupine paketa. Paketi su sve mape koje u sebi imaju datoteku `__init__.py` pa su često i moduli zapravo paketi, no moduli nose i šire značenje kao organizacijska jedinica. Moduli i paket su najviše okrenuti drugim programerima kako bi im te biblioteke pomogle u razvijanju aplikacija. Kod programiranja se moduli pozivaju naredbama poput `import MyPackage` ili `from MyPackage import Function`. [22]

3.2 Aplikacije

Aplikacije su okrenute krajnjim korisnicima koji ne moraju biti programeri i obično u sebi nose kompletne funkcionalnosti. Kroz primjere u nadolazećim poglavljima vidjet ćemo kako je primjena `Pythona` vrlo široka - koristi se u mnoge svrhe kao što su serveri, klijenti, igre itd.

3.2.1 Jednostavne aplikacije

Kad govorimo o jednostanvim aplikacijama, uglavnom mislimo na aplikacije koje se sastoje od nekoliko skripti, vrlo malo implementiranih funkcionalnosti te primjeni na točno ono za što su namijenjene. Takve se aplikacije slažu s filozofijom `Linux/GNU` aplikacija: "Alati su jednostavni programi, uglavnom napravljeni u neku specifičnu svrhu te se ponekad nazivaju naredbama" te omogućavaju njihovu dobru međusobnu povezanost. [25]

Primjer jednostavne aplikacije koji se često javlja u literaturi je `Howdoi`. [15] Ta aplikacija ima jednu funkcionalnost koja automatizirano pronalazi rješenje na jednostavne probleme koje programer može imati, a to je riješeno kroz konzolno sučelje:

```
$ howdoi import package in python
import FooPackage.foo
```

Unatoč tome što su ovdje programeri ciljana publika, ova se aplikacija ipak smatra korisničkom jer je njeno sučelje prilagođeno na način da programerske vještine nisu potrebne za uspješno korištenje aplikacije.

Distribucija ovakvih aplikacija može se napraviti korištenjem već opisanih metoda, od kojih će ovdje biti istaknute samo ključne. Unutar `setup.py` datoteke su navedeni svi podatci o autoru aplikacije i aplikaciji (ime, e-mail, naziv aplikacije, opisi itd.).

Argument `long_description` popunjen je na način da je stvorena funkcija `read` koja prima argumente koji su nazivi datoteka, čita ih te vraća sadržaj tih datoteka uzastopno.

Argument `install_requires` popunjen je nazivima modula koji su potrebni za izvršavanje aplikacije. Kao dodatak je napravljena funkcija koja dohvaća verziju `Pythona` koja se pokreće te dodaje uvjet za modul `Argparse`.

Ulazna točka je dodana jednostavnim unosom:

```
entry_points={
        'console_scripts': [
            'howdoi = howdoi.howdoi:command_line_runner',
        ]
    },
```

Datoteka `MANIFEST.in` nosi popis ostalih datoteka koje je autor htio uključiti u aplikaciju, a to su `CHANGES.txt` koja nosi popis svih promjena u razvoju i `fastentrypoints.py` koja služi za poboljšanje performansi ulaznih točaka (riječ je o detaljima koji su izvan dosega ovog rada). Popis promjena je ovdje dodan jer je autor zamislio to kao odvojenu datoteku od `README` datoteke te se ona doda pomoću već opisane funkcije u `long_description`.

3.2.2 Složene aplikacije

Složene aplikacije su drugačije po tome što nose puno više kompleksnosti te zahtijevaju drugačiji pristup problemu distribucije. Neke od poznatijih aplikacija pisane u `Pythonu` su `Dropbox`, igra `EVE online`, `Mercurial` (sustav za kontrolu verzioniranja) itd. Za takvu distribuciju namijenjen je tzv. zamrzavanje (en. *freezing*, različito od `pip` naredbe `freeze`). One se distribuiraju na način da korisnik uopće ne mora razmišljati o `Pythonu`, štoviše, ne mora niti znati da je aplikacija implementirana u tom programskom jeziku. `Python` interpreter distribuira se s aplikacijom te će uvijek raditi na svim računalima.

Ono što je potrebno za dobro korisničko iskustvo je aplikacija koja se pokreće kao `executable`, a ne kao skripta koja se mora pokrenuti pomoću nekog interpretera. Zamrzavanjem je to omogućeno uz nedostatak da će cijela aplikacija biti veća za 2-12MB. Taj nedostatak i ne mora biti jako naglašen ukoliko je sama aplikacija značajno veća od desetak MB.

Za zamrzavanje postoji nekoliko alata koji se razlikuju po raznim parametrima kao što su podržani operacijski sustavi, podrška za razne grafičke biblioteke, licence itd. Ovdje neće biti obrađeni svi, već samo `pyInstaller` i `cx_Freeze` koji podržavaju `Linux/GNU` operacijske sustave te podržavaju `Python 3`, odnosno još se uvijek održavaju.

Alat `pyInstaller` podržava nekoliko grafičkih paketa kao što su `pygame`, `PyGTK`, `PyQT4` i `pyQT5`, `PyOpenGL` i druge. To čini `pyInstaller` dobrim odabirom za zamrzavanje raznih igara koje su pisane u `Pythonu`.

Koristeći `pyInstaller` zadržavaju se licence koju su korištene u svim bibliotekama za tu aplikaciju, ali se i dopušta korisnicima da aplikacije koje naprave ne budu besplatne, odnosno da se njihovo korištenje naplaćuje. Također omogućeno je i sakrivanje kôda što je olakotno za poduzeća koja imaju namjeru naplatiti kôd koji napišu.

Korištenje `pyInstaller` alata jednostavno je - nakon instalacije (`$ pip install pyinstaller`) `pyinstaller` je spreman na korištenje. Od skripte `skripta.py` *executable* se radi:

```
$ pyinstaller skripta.py
```

Nakon toga nastaje mapa `build` uz druge datoteke i mape u kojoj se nalaze sve datoteke koje se moraju distribuirati. U procesu nastaje i datoteka sa `.spec` nastavkom u kojoj se mogu popisati datoteke koje treba dodati (slično datoteci `MANIFEST.in`). Dokumentacija alata `pyInstaller` ima popis tzv. recepata koji olakšavaju programerima pisanje `.spec` datoteka tako da primijene već postojeći predložak.

Osim alata `pyInstaller` postoji i `cx_Freeze` koji stvara izvršne datoteke integracijom unutar `setup.py` skripte. Tamo se mogu dodati opcije koje želimo te na taj način *obogatiti* instalaciju. Za izvršnu datoteku za računala s `Linux/GNU` operacijskim sustavom koristimo naredbu `$ python setup.py bdist_rpm`. [22]

Oba se alata mogu koristiti za distribuciju aplikacija na više platformi, konkretno koristeći `Linux/GNU` pomoću `pyInstaller` alata možemo napraviti `Windows` aplikaciju koristeći `Wine` (sloj za kompatibilnost `Windows` aplikacija na `Linux/GNU operacijskim sustavima`. Međutim, to se ne preporuča raditi s `cx_Freeze` jer može doći do problema s nekim datotekama.

3.2.3 Aplikacije `plug-in` arhitekture

`Plug-in` arhitektura aplikacije je arhitektura u kojoj glavna aplikacija sadrži samo osnovne funkcionalnosti, a dodatne funkcionalnosti se mogu dodati u aplikaciju pomoću dodataka (tzv. `plug-in`). Takve aplikacije ima smisla kreirati ukoliko je riječ o vrlo velikim aplikacijama ili ako je određenim korisnicima potreban samo dio funkcionalnosti za njeno korištenje te nema potrebe za instalacijom velike količine softvera. Kao primjer takvih aplikacija možemo uzeti mnoge ERP sustave koji se čak i naplaćuju ovisno o tome koliko funkcionalnosti se isporučuje korisnicima.

Sa strane *developera* to znači da svaki put kad se instalira neki dodatak glavna aplikacija mora na neki način znati da se dodatak instalirao te da se iz glavne aplikacije može pristupiti svim dodatcima.

U kontekstu ovog rada obratit ćemo pozornost na primjenu alata koje smo do sada obradili u svrhu stvaranja takve arhitekture.

Kako bismo ostvarili željenu arhitekturu za *plug-in* aplikaciju trebamo:

- glavnu aplikaciju imati spremnu za pokretanje iz konzole
- ulazne točke dodataka aplikaciji imati spremne u distribuciji `Pythona`

- automatski povezati glavnu aplikaciju s dodatcima

Preporučeno je imenovati ključeve tako da se prvo stavi ime paketa te se nakon točke specifičnije označi o čemu se zapravo radi. Na primjer:

```
setup(
    entry_points={
        'baseApp.commands':[
            'help=manual.manual:main',
            'config=configuration.configuration:main',
        ],
    },
)
```

Ova `setup` funkcija nam govori da je ulazna točka `help` u nekoj metodi, funkciji i sl. koja se zove `main` u datoteci `manual.py` unutar paketa `manual` te analogno i za *config*.

Dohvaćanje ulaznih točaka moguće je provjeriti alatima iz paketa `pkg_resources` koji dolazi s osnovnim paketom `Pythona`. `Iter_entry_points` iterira po svim ulaznim točkama i daje nam njihove reference kako bismo im mogli pristupiti iz koda.

```
Python 3.7.2 (default, Jan 10 2019, 23:51:51)
[GCC 8.2.1 20181127] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pkg_resources import iter_entry_points
>>> for e in iter_entry_points('baseapp.commands'):
...      print(repr(e))
...
EntryPoint.parse('help=helpmanual.manual:main')
EntryPoint.parse('config=configuration.configuration:main')
```

Za ostvarivanje *plug-in* arhitekture smo riješili prve dvije točke s početka poglavlja. Na kraju nam ostaje povezati cijelu aplikaciju u jedan sustav koji će izvršavati željene funkcionalnosti. Završni korak je glavnu aplikaciju naučiti kako se čitaju ulazne točke aplikacije. Kako bismo to ostvarili, modul `pkg_resources` nam pruža nekoliko funkcija za tu svrhu. /citesnek

U aplikaciji moramo sve ulazne točke pospremiti u neku strukturu, npr. rječnik:

```
from pkg_resources import
        load_entry_point,
        get_entry_point,
        iter_entry_points
# ep (kao entry points)
ep = {}
for entry_point in iter_entry_points('baseApp.commands'):
         ep[entry_point.name] = entry_point.load()
```

Sada u rječniku `ep` imamo pospremljene sve ulazne točke koje jednostavno možemo pokrenuti preko ključa koji je ujedno i naziv ulazne točke. Dalje se prema potrebi može napraviti čitanje argumenata kako bi se pokrenula određena ulazna točka, ili se može napraviti njihovo uzastopno čitanje, ili se može napraviti neka treća varijante pozivanja dodataka aplikaciji ovisno o potrebama aplikacije.

Ovakav pristup dobar je za velike projekte i kad ne možemo kao programeri znati koje sve pakete ima korisnik instaliran prije nego što se kod počne izvršavati.[27]

Ukoliko razvijamo aplikaciju koja sama po sebi ima definirano više ulaznih točaka, onda njih pojedinačno možemo pokrenuti izvršavanjem:

```
load_entry_point('plugin', 'baseApp.commands', 'plug1')
```

Na sličan se način mogu i pregledati informacije o ulaznim točkama bez njihovog izvršavanja:

```
>>> print(get_entry_point('plugin', 'baseApp.commands', 'plug1'))
'plug1 = plugin.app:plug1'
```

Objekt tipa `EntryPoint` kojeg vraća funkcija `get_entry_point` daje nam određenu fleksibilnost zbog svojstava koji su definirani za njega. Naziv ulazne točke može se dobiti pomoću svojstva `name`, naziv distribucije pomoću svojstva `dist`, naziv paketa pomoću `module_name`, argumenti pomoću svojstva `extras` itd.

Ovakvim pristupom dobivamo arhitekturu koja je modularna i proširiva na jednostavan način. Aplikaciju možemo realizirati do kraja dodatnim programiranjem koje je izvan opsega ovog rada, a finalni rezultat može rezultirati korisničkim sučeljem koje možda i prepoznajemo iz drugih aplikacija:

```
These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

(...)
```

Unatoč tome što `git` nije modularna aplikacija u smislu objašnjenom u ovom poglavlju, vrhunski je primjer kako aplikacija može izgledati u konačnici, gdje `clone` i `init` mogu predstavljati opcije osnovnog modula, a `add`, `mv`, `reset` itd. opcije uključenih dodatnih modula.

Poglavlje 4

Sigurnosni aspekt distribuiranja aplikacija

Kao i sve što se nalazi na internetu i dostupno je javnoj masi, `Python` i njegove aplikacije također su potencijalni izvor prijetnji unatoč tome što sve pakete pregledavaju mnogi. Osim pod maliciozni kod, viruse i sl. izvor prijetnji može biti i nestručno rukovanje paketima te može narušiti integritet sustava instaliranog na računalu.

4.1 Instalacija paketa u sistemskom `Pythonu`

Kao što je već opisano kao motivacija za korištenje virtualnih okruženja u `Pythonu`, instalacija paketa u sistemski `Python` nije preporučeno. Glavni razlog tome je što bi se o sistemskim paketima trebao brinuti sistemski *package manager*. Oni su različiti ovisno o kojoj se distribuciji `Linux/GNU` operacijskog sustava radi.

Korištenjem paketa `pip` za instalaciju drugih paketa može dovesti do nesinkroniziranosti stvarnih instaliranih paketa i onoga što sustavski *package manager* misli da je instalirano. Slikovito možemo reći da radimo operacijskom sustavu iza leđa. [24]

Osim loše sinkronizacije, potencijalan problem može predstavljati i nehotično brisanje paketa koji su esencijalni za rad sustava. Mnogo je korisničkih aplikacija pisano u `Pythonu` (`ranger`, `rtv`, `pulsemixer` itd.) te njihova deinstalacija može prouzročiti nemogućonst obavljanja određenih zadataka. Osim njih, moguće je i deinstalirati pakete koji su potrebni za rad aplikacija što se može, ali i ne mora moći jednostavno riješiti.

4.2 Maliciozni kod

Osim korisnika koji neoprezno rukuju svojim sustavom prijetnje su moguće i od nestašnih programera koji postavljaju svoje aplikacije na službene repozitorije. Tako je Slovački nacionalni centar za digitalnu sigurnost u 2017. pronašao dvanaest paketa koji su sadržavali maliciozni kod. Pristup koji je korišten je neinformiranost korisnika `Pythona` jer su maliciozni paketi imali slična imena onim *pravim* paketima (`urlib` i `urlib3` mijenjaju `urllib3`, `setup-tools` mijenja `setuptools` i sl., poznato pod nazivom *typo-packages*). [4]

Ti su paketi maknuti s repozitorija, no teško je procijeniti koliko se riješio problem jer su ti paketi dospjeli u neke aplikacije. Poduzeća koja su implementirala *krive* pakete u svojim aplikacijama pozvana su na pregledavanje svojeg koda, no i dalje nema garancije da su paketi u potpunosti istrijebljeni iz uporabe.

Zla namjera pronađena je tako što su korisnici `Pythona` epohe 3 primijetili kako im paket ne radi kako je predviđeno i kod otklanjanja grešaka (en. *debugging*) su vidjeli da se podaci šalju na servere u Kini.

Mogućnost ovakvog prodora djelomično je omogućeno i time što je `pip` dobio podršku za provjeravanje *hash* sumi paketa tek nedavno prije tog incidenta te korisnici još nisu u potpunosti implementirali te funkcionalnosti. Drugi je razlog što `PyPI` ne posjeduje nikakav sustav kontrole kvalitete i sigurnosti koda. [5]

Zanimljivo je što su sve klase i moduli nazvani identično kao i oni iz pravih paketa stoga deinstalacija malicioznih i instalacija ispravnih paketa jednostavno rješava problem bez potrebe za popravljanjem softvera.

4.3 Instalacija preko `setup.py` datoteke

U 2018. je na princip *typo-packages* otkriveno još nekoliko paketa koji su iskoristili korisnika koji je pozvao instalaciju paketa uz pomoć `sudo` naredbe. [23] Na taj način je instaliran paket pregledavao *clipboard* korisnika, tražio sadržaj koji nalikuje na adrese `Bitcoin` novčanika te ih, ukoliko se radilo o odredišnoj adresi, izmijenio u adresu novčanika nestašnog programera. [3]

Paket koji se infiltrirao kao paket `colorama` računajući na britansko pisanje riječi značenja *boja* sa *u* zvao se `colourama`. 54 korisnika je preuzelo paket, no nijedan transfer novaca se nije dogodio. [14]

Poglavlje 5

Zaključak

Unatoč kaosu koji je nastao kroz godine zbog raznih projekata koji su na različite načine pokušali riješiti problem distribucije, radna skupa `PyPA` uspjela je organizirati različite načine u jedan preporučeni način koji se pokazao kao dobar za sve primjere s kojima se susrećemo koristeći `Python`. Danas su svi putevi koje jedna aplikacija mora proći već mnogim stopama utabani te je time i postignuta (relativna) sigurnost distribucije te je zajednica koja koristi opisane alate postala široka.

Sve to u konačnici rezultira i olakšanim korištenjem `Pythona` u razne svrhe izvan okvira programiranja kao takvog, kao što su igre (`EVE online`, `Sid Meier's Civlization IV`), aplikacije (`Dropbox`), web stranice (ili njihovi dijelovi)(`Facebook`, `YouTube`, `Reddit` itd.) te poprilično jednostavan način distribucije tih aplikacija i kôda.

Bibliografija

[1] Python Packaging Authority. Building and distributing packages with setuptools. https://setuptools.readthedocs.io/en/latest/setuptools.html, 2019. Pristupljeno 19.08.2019.

[2] Ian Bicking. User guide. https://docs.python.org/3.7/distutils/sourcedist.html, 2018. Pristupljeno 02.09.2019.

[3] Danny Bradburry. Snakes in the grass! malicious code slithers into python pypi repository. https://nakedsecurity.sophos.com/2018/10/30/snakes-in-the-grass-malicious-code-slithers-into-python-pypi-repository/, 2018. Pristupljeno 12.09.2019.

[4] Nacionalni centar za digitalnu sigurnost u Slovačkoj. skcsirt-sa-20170909-pypi. https://www.nbu.gov.sk/skcsirt-sa-20170909-pypi/, 2017. Pristupljeno 12.09.2019.

[5] Catalin Cimpanu. Ten malicious libraries found on pypi - python package index. https://www.bleepingcomputer.com/news/security/ten-malicious-libraries-found-on-pypi-python-package-index/, 2018. Pristupljeno 12.09.2019.

[6] Nick Coghlan and Donald Stufft. Pep 440 – version identification and dependency specification. https://www.python.org/dev/peps/pep-0440/, 2013. Pristupljeno 02.09.2019.

[7] Charlie Denton. Python wheels. https://pythonwheels.com/, 2019. Pristupljeno 10.09.2019.

[8] SciPy developers. Installing packages. https://www.scipy.org/install.html, 2019. Pristupljeno 19.08.2019.

[9] Python dokumentacija. 2. writing the setup script. https://docs.python.org/3.7/distutils/setupscript.html, 2019. Pristupljeno 05.09.2019.

[10] Python dokumentacija. 3. writing the setup configuration file. https://docs.python.org/3.7/distutils/configfile.html, 2019. Pristupljeno 19.08.2019.

[11] Python dokumentacija. Creating a source distribution. https://docs.python.org/3.7/distutils/sourcedist.html, 2019. Pristupljeno 02.09.2019.

[12] Python dokumentacija. Installing python modules. https://docs.python.org/3.7/installing/index.html, 2019. Pristupljeno 19.08.2019.

[13] Python dokumentacija. Installing python modules (legacy version). `https://docs.python.org/3.7/install/index.html`, 2019. Pristupljeno 19.08.2019.

[14] Alisa Esage. 12 malicious libraries found in python pypi. `https://www.securitynewspaper.com/2018/10/31/12-malicious-libraries-found-in-python-pypi/`, 2018. Pristupljeno 12.09.2019.

[15] Benjamin Gleitzman. Howdoi. `https://github.com/gleitz/howdoi`, 2019. Pristupljeno 15.09.2019.

[16] Python Packaging User Guide. Glossary. `https://packaging.python.org/glossary/#term-python-packaging-authority-pypa`, 2019. Pristupljeno 12.09.2019.

[17] Python Packaging User Guide. Installing packages. `https://packaging.python.org/tutorials/installing-packages`, 2019. Pristupljeno 19.08.2019.

[18] Python Packaging User Guide. Installing scientific packages. `https://packaging.python.org/guides/installing-scientific-packages`, 2019. Pristupljeno 19.08.2019.

[19] Anaconda Inc. Anaconda distribution. `https://www.anaconda.com/distribution/`, 2019. Pristupljeno 19.08.2019.

[20] pip documentation. User guide. `https://pip.pypa.io/en/stable/user_guide/`, 2019. Pristupljeno 12.09.2019.

[21] Real Python. Python virtual environment - a primer. `https://realpython.com/python-virtual-environments-a-primer/`, 2019. Pristupljeno 02.09.2019.

[22] Kenneth Reitz and Tanya Schlusser. *The Hitchhiker's Guide to Python*. Packt Publishing Ltd., Birmingham, UK, 2016.

[23] Antonio Zekić Robert Perica. Suppy chain malware - detecting malware in package manager repositories. `https://blog.reversinglabs.com/blog/suppy-chain-malware-detecting-malware-in-package-manager-repositories`, 2019. Pristupljeno 12.09.2019.

[24] Donald Stufft. Default to –user. `https://github.com/pypa/pip/issues/1668`, 2014. Pristupljeno 12.09.2019.

[25] GNU/Linux Command-Line Tools Summary. Chapter 3. the unix tools philosophy. `http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/c1089.htm`. Pristupljeno 15.09.2019.

[26] Wikipedia. Symbolic link. `https://en.wikipedia.org/wiki/Symbolic_link`, 2019. Pristupljeno 02.08.2019.

[27] Phoenix Zerin. Demystefying setuptools entry points. `https://www.youtube.com/watch?v=0W0k6zP_Lto`, 2017. Pristupljeno 27.08.2019.

[28] Tarek Ziadé and Michal Jaworski. *Expert Python programming: learn best practices to designing, coding, and distributing your Python software, third edition.* Packt Publishing Ltd., Birmingham, UK, 2008.