POGLAVLJE 11

Mrežno i veb programiranje

Ovo poglavlje pokriva razne teme vezane za korišćenje Pythona u mrežnim i distribuiranim aplikacijama. S jedne strane, govorimo o korišćenju Pythona kao klijenta za pristup raznim servisima, a s druge, o implementiranju mrežnih servisa u vidu servera pomoću Pythona. Predstavljamo i uobičajene tehnike za pisanje koda za rad ili komuniciranje sa interpreterima.

11.1. Interakcija sa HTTP servisima iz pozicije klijenta

Problem

Treba da pristupate raznim servisima putem HTTP protokola kao klijent. Na primer, da preuzimate podatke ili da radite sa API interfejsom REST.

Rešenje

U jednostavnim slučajevima, obično je najlakše primeniti modul `urllib.request`. Na primer, da biste poslali jednostavan HTTP zahtev `GET` udaljenom servisu, postupićete na sledeći način:

```
from urllib import request, parse

# Pristupa se osnovnoj URL adresi
url = 'http://httpbin.org/get'

# Rečnik parametara upita (ako ih ima)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Kodira znakovni niz upita
querystring = parse.urlencode(parms)

# Pravi GET zahtev i čita odgovor
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

Ukoliko je potrebno da pošaljete parametre upita u telu zahteva metodom `POST`, kodiraćete ih i poslati kao opcioni argument funkcije `urlopen()` na sledeći način:

```
from urllib import request, parse

# Pristupa se osnovnoj URL adresi
url = 'http://httpbin.org/post'

# Rečnik parametara upita (ako ih ima)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}

# Kodira znakovni niz upita
querystring = parse.urlencode(parms)

# Pravi POST zahtev i čita odgovor
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

Ako u zahtev treba da uvrstite neka namenska HTTP zaglavlja, kao što je promena polja korisničkog agenta, napravite rečnik s njihovom vrednošću, potom i instancu tipa `Request` i prosledite je funkciji `urlopen()` na sledeći način:

```
from urllib import request, parse
...

# Dodatna zaglavlja
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

req = request.Request(url, querystring.encode('ascii'), headers=headers)

# Pravi zahtev i čita odgovor
u = request.urlopen(req)
resp = u.read()
```

Ukoliko je interakcija komplikovanija, verovatno bi trebalo da razmislite o biblioteci `requests` (*http://pypi.python.org/pypi/requests*). Na primer, evo koda ekvivalentnog prethodnim operacijama, ali s bibliotekom `requests`:

```
import requests

# Pristupa se osnovnoj URL adresi
url = 'http://httpbin.org/post'

# Rečnik parametara upita (ako ih ima)
parms = {
   'name1' : 'value1',
   'name2' : 'value2'
}
```

```
# Dodatna zaglavlja
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

resp = requests.post(url, data=parms, headers=headers)

# Dekodirani tekst koji zahtev vraća
text = resp.text
```

Važna karakteristika biblioteke `requests` jeste način na koji vraća sadržaj dobijenog odgovora na zahtev. Kao što je prikazano, atribut `resp.text` sadrži dekodirani Unicode tekst zahteva. Međutim, ako pristupite atributu `resp.content`, umesto toga dobijate binarni sadržaj. S druge strane, vrednost u atributu `resp.json` jeste sadržaj odgovora u formatu JSON.

Evo kako biste upotrebili `requests` da napravite `HEAD` zahtev i izdvojite nekoliko polja podataka zaglavlja iz zahteva:

```
import requests

resp = requests.head('http://www.python.org/index.html')

status = resp.status_code
last_modified = resp.headers['last-modified']
content_type = resp.headers['content-type']
content_length = resp.headers['content-length']
```

U narednom primeru, biblioteka `requests` se koristi za prijavljivanje na portal Python Package Index putem osnovne provere identiteta:

```
import requests

resp = requests.get('http://pypi.python.org/pypi?:action=login',
                    auth=('user','password'))
```

Evo primera u kome se pomoću biblioteke `requests` prosleđuju HTTP kolačići od jednog do drugog zahteva:

```
import requests

# Prvi zahtev
resp1 = requests.get(url)
...
# Drugi zahtevi dobijeni na prve zahteve
resp2 = requests.get(url, cookies=resp1.cookies)
```

Na kraju, evo primera korišćenja biblioteke `requests` za otpremanje sadržaja:

```
import requests
url = 'http://httpbin.org/post'
files = { 'file': ('data.csv', open('data.csv', 'rb')) }

r = requests.post(url, files=files)
```

Objašnjenje

Ugrađeni modul `urllib` obično je dovoljan za jednostavan kôd HTTP klijenta. Međutim, za bilo šta van jednostavnih `GET` ili `POST` zahteva, ne možete računati na njegovu funkcionalnost. Tu na scenu stupaju namenski moduli kao što je `requests`.

Recimo, ako odlučite da se dosledno držite standardne biblioteke, izbegavajući biblioteke kakva je `requests`, možda ćete morati da implementirate kôd pomoću modula niskog nivoa `http.client`. Na primer, evo kako se izvršava `HEAD` zahtev:

```
from http.client import HTTPConnection
from urllib import parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print('Status', resp.status)
for name, value in resp.getheaders():
    print(name, value)
```

Isto tako, ako treba da pišete kôd s namenskim klasama, proverom identiteta, kolačićima i drugim detaljima, modul `urllib` je nepodesan i preopširan. Kao primer, navešćemo jednostavan segment koda za proveru identiteta prilikom upisa na portal Python Package Index:

```
import urllib.request

auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi','http://pypi.python.org','username','password')
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?:action=login')
u = opener.open(r)
resp = u.read()

# Odavde možete da pristupite stranicama koristeći opener
...
```

Iskreno govoreći, sve ovo je mnogo lakše uraditi pomoću biblioteke `requests`.

Testiranje koda HTTP klijenta u razvojnoj fazi često može biti obeshrabrujuće zbog raznih nezgodnih detalja o kojima morate da vodite računa (na primer, o kolačićima, proveri identiteta, zaglavljima, kodiranju itd.). U tome bi vam mogao pomoći servis httpbin (*http://httpbin.org*). Ova veb lokacija prima zahteve, a potom vam vraća informacije u obliku JSON odgovora. Evo interaktivnog primera:

```
>>> import requests
>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...     headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type': '',
'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
```

```
'keep-alive', 'Host': 'httpbin.org', 'Accept': '*/*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>
```

Često je bolje raditi s veb lokacijom *httpbin.org* nego eksperimentisati s pravom veb prezentacijom, naročito ako postoji rizik da vam ukinu nalog posle tri neuspešna pokušaja prijavljivanja (tj., ne pokušavajte da naučite kako se piše HTTP klijent za proveru identiteta tako što ćete se prijavljivati na portal vaše banke).

Iako ovde ne razgovaramo o tome, modul `requests` ima podršku za brojne naprednije protokole HTTP klijenata, kao što je OAuth. Dokumentacija za modul `requests` (*http://docs.python-requests.org*) izuzetna je (i, iskreno govoreći, bolja od bilo čega što bi moglo da stane u ovako ograničen prostor). Tamo ćete naći podrobnije informacije.

11.2. Pravljenje TCP servera

Problem

Hoćete da implementirate server koji komunicira s klijentima putem TCP Internet protokola.

Rešenje

Biblioteka `socketserver` omogućava da lako napravite TCP server. Na primer, evo jednostavnog eho-servera (engl. *echo server*):

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

U ovom kodu definisali smo specijalnu upravljačku klasu koja implementira metodu `handle()` za servisiranje klijentskih veza. Atribut `request` predstavlja odgovarajući klijentski priključak, dok `client_address` sadrži adresu klijenta. Da bismo testirali server, pokrenućemo ga, a potom otvoriti poseban Pythonov proces koji se povezuje s njim:

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
```

```
>>> s.recv(8192)
b'Hello'
>>>
```

Često je lakše definisati malo drugačiju vrstu upravljačke klase. U narednom primeru koristimo osnovnu klasu `StreamRequestHandler` da priključku pridružimo interfejs nalik datoteci:

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # self.rfile je objekat nalik datoteci za čitanje
        for line in self.rfile:
            # self.wfile je objekat nalik datoteci za upisivanje
            self.wfile.write(line)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

Objašnjenje

Pomoću modula `socketserver`, relativno je lako napraviti jednostavne TCP servere. Međutim, trebalo bi da znate da su serveri podrazumevano jednonitni i da u jednom trenutku mogu da opslužuju samo jednog klijenta. Ako hoćete da radite s više klijenata, umesto toga napravite instancu tipa `ForkingTCPServer` ili `ThreadingTCPServer`. Na primer:

```
from socketserver import ThreadingTCPServer
...
if __name__ == '__main__':
    serv = ThreadingTCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

Ovakvi serveri, koji rade s više klijenata ili procesa, prave nov proces ili nit za svaku vezu s klijentom. Ne postoji ograničenje u broju dozvoljenih klijenata, tako da bi zlonamerni haker mogao pokrenuti veliki broj veza u isto vreme, s namerom da onesposobi vaš server.

Ukoliko se brinete da bi se to moglo desiti, možete da napravite unapred alocirano skladište radnih niti ili procesa. U tu svrhu, napravili biste instancu normalnog servera bez niti, ali biste potom pokrenuli metodu `serve_forever()` nad skladištem niti. Na primer:

```
...
if __name__ == '__main__':
    from threading import Thread
    NWORKERS = 16
    serv = TCPServer(('', 20000), EchoHandler)
    for n in range(NWORKERS):
        t = Thread(target=serv.serve_forever)
        t.daemon = True
        t.start()
    serv.serve_forever()
```

U normalnim okolnostima, `TCPServer` povezuje i aktivira utičnicu po instanciranju. Međutim, ponekad ćete hteti da podesite datu utičnicu. Da biste to uradili, prosledili argument `bind_and_activate=False` na sledeći način:

```
if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler, bind_and_activate=False)
    # Podešava razne opcije priključka
    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    # Povezuje i aktivira
    serv.server_bind()
    serv.server_activate()
    serv.serve_forever()
```

Prikazana opcija utičnice zapravo je uobičajena postavka koja omogućava serveru da se ponovo poveže s prethodno korišćenim brojem priključaka (engl. *port*). U toj meri je uobičajena, da predstavlja promenljivu klase koja se može zadati za klasu `TCPServer`. Podesićete je pre instanciranja servera, kao što je prikazano u ovom primeru:

```
...
if __name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

U ovom rešenju prikazane su dve upravljačke osnovne klase (`BaseRequestHandler` i `StreamRequestHandler`). Klasa `StreamRequestHandler` malo je prilagodljivija i podržava neke opcije koje se mogu aktivirati putem specifikacije dodatnih promenljivih klase. Na primer:

```
import socket

class EchoHandler(StreamRequestHandler):
    # Opciona podešavanja (podrazumevana u ovom primeru)
    timeout = 5                     # Tajm-aut za sve operacije utičnice
    rbufsize = -1                   # Čita veličinu bafera
    wbufsize = 0                    # Upisuje veličinu bafera
    disable_nagle_algorithm = False  # Zadaje opciju TCP_NODELAY utičnice
    def handle(self):
        print('Got connection from', self.client_address)
        try:
            for line in self.rfile:
                # self.wfile je objekat nalik datoteci za pisanje
                self.wfile.write(line)
        except socket.timeout:
            print('Timed out!')
```

Na kraju, valja primetiti da je većina Pythonovih mrežnih modula visokog nivoa (na primer, HTTP, XML-RPC itd.) napravljena povrh funkcionalnosti `socketserver`. Ipak, servere je jednostavno i direktno implementirati pomoću biblioteke `socket`.

Evo jednostavnog primera direktnog programiranja servera pomoću ove biblioteke:

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_handler(address, client_sock):
    print('Got connection from {}'.format(address))
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)
    client_sock.close()

def echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if __name__ == '__main__':
    echo_server(('', 20000))
```

11.3. Pravljenje UDP servera

Problem

Hoćete da implementirate server koji komunicira s klijentima putem UDP Internet protokola.

Rešenje

Kao i TCP servere, UDP servere ćete lako napraviti pomoću biblioteke `socketserver`. Evo primera jednostavnog servera za vreme:

```
from socketserver import BaseRequestHandler, UDPServer
import time

class TimeHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # Dobavlja poruku i utičnicu klijenta
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    serv = UDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

I u ovom slučaju, definišete posebnu upravljačku klasu koja implementira metodu `handle()` za servisiranje klijentskih veza. Atribut `request` je n-torka koja sadrži dolazeći datagram i prateći objekat utičnice za server. `client_address` sadrži adresu klijenta.

Ako želite da testirate server, pokrenite ga, a potom otvorite zaseban Pythonov proces koji mu šalje poruke:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

Objašnjenje

Tipičan UDP server prima datagram (poruku) zajedno sa adresom klijenta. Ako server hoće da odgovori, vratiće datagram klijentu. Datagrame bi trebalo da šaljete metodama `sendto()` i `recvfrom()` utičnice. Iako bi mogle da se upotrebe i tradicionalne metode `send()` i `recv()`, prethodne dve metode su uobičajene kada je reč o UDP komunikaciji.

S obzirom da u osnovi nema veze, UDP server se često mnogo lakše piše nego TCP server. Međutim, UDP server je i nepouzdan (na primer, ne uspostavlja se „veza“ i poruke se mogu zagubiti). Zato je na vama da odredite šta ćete sa izgubljenim porukama. Ta tema prevazilazi opseg ove knjige, ali se pouzdanost obično postiže brojevima sekvenci, ponovnim pokušajima, tajm-autom i drugim mehanizmima, ukoliko je to važno za vašu aplikaciju. UDP se često koristi u slučajevima za koje ne postoje strogi zahtevi za pouzdanu isporuku poruka. Na primer, u aplikacijama koje se izvršavaju u realnom vremenu, poput reprodukovanja multimedijskih sadržaja u realnom vremenu i igrica, kada nema mogućnosti da se vratite unazad i povratite izgubljeni paket (program ga naprosto preskače i nastavlja napred).

Klasa `UDPServer` je jednonitna, što znači da se u jednom trenutku može obrađivati samo jedan zahtev. U praksi, to je manje važno za UDP veze nego za TCP veze. Ipak, ako vas zanimaju istovremene operacije, umesto toga instanciraćete objekat tipa `ForkingUDPServer` ili `ThreadingUDPServer`:

```
from socketserver import ThreadingUDPServer
...
if __name__ == '__main__':
    serv = ThreadingUDPServer(('',20000), TimeHandler)
    serv.serve_forever()
```

Direktno implementiranje UDP servera pomoću utičnica takođe nije teško uraditi. Evo primera:

```
from socket import socket, AF_INET, SOCK_DGRAM
import time

def time_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
```