9. МЕРЕЖНІ ДОДАТКИ НА PYTHON.

9.1. Робота із сокетами

Застосовувана в IP-мережах архітектура клієнт-сервер використовує IP-пакети для комунікації між клієнтом та сервером. Клієнт відправляє запит серверу, на який той відповідає. У випадку із TCP/IP між клієнтом та сервером встановлюється з'єднання (звичайно із двостороннім передаванням даних), а у випадку з UDP/IP – клієнт та сервер обмінюються пакетами (дейтаграмами) з негарантованою доставкою.

Кожний мережевий інтерфейс IP-мережі має унікальну в цій мережі адресу (IP-адресу). Спрощено можна вважати, що кожний комп'ютер у мережі Інтернет має власну IP-адресу. При цьому в рамках одного мережевого інтерфейсу може бути кілька мережевих портів. Для встановлення мережевого з'єднання програма клієнта повинна вибрати вільний порт та встановити з'єднання із серверною програмою, що слухає (listen) порт із визначеним номером на віддаленому мережевому інтерфейсі. Пара IP-адреса та порт характеризують сокет (гніздо) – початкову (та кінцеву) точку мережевої комунікації. Для створення з'єднання TCP/IP необхідно два сокети: один на локальній машині, а інший – на віддаленій. Таким чином, кожне мережеве з'єднання має IP-адресу та порт на локальній машині, а також IP-адресу та порт на віддаленій машині.

Модуль *socket* забезпечує можливість працювати із сокетами в Python. Сокети використовують транспортний рівень згідно з семирівневою моделлю OSI (Open Systems Interconnection, взаємодія відкритих систем), тобто належать до нижчого рівня, ніж більшість описуваних у цьому підрозділі протоколів.

Наведемо рівні моделі OSI.

Фізичний – потік бітів, переданих по фізичній лінії. Визначає параметри фізичної лінії.

Канальний (Ethernet, PPP, ATM тощо) кодує та декодує дані у вигляді потоку бітів, справляючись із помилками, що виникають на фізичному рівні в межах фізично єдиної мережі.

Мережевий (IP) маршрутизує інформаційні пакети від вузла до вузла.

Транспортний (TCP, UDP тощо) забезпечує прозоре передавання даних між двома точками з'єднання.

Сеансовий керує сеансом з'єднання між вузлами мережі. Починає, координує та завершує з'єднання.

Представлення забезпечує незалежність даних від форми їхнього подання шляхом перетворення форматів. На цьому рівні може виконуватися прозоре (із погляду вищого рівня) шифрування та дешифрування даних.

Прикладний (HTTP, FTP, SMTP, NNTP, POP3, IMAP тощо) підтримує конкретні мережеві застосування. Протокол залежить від типу сервісу.

Кожний сокет належить до одного з комунікаційних доменів. Модуль *socket* підтримує домени UNIX та Internet. Кожен домен стосується свого набору протоколів та адресації. Цей підрозділ порушуватиме тільки питання домену Internet, а саме протоколи TCP/IP та UDP/IP, тому для вказання комунікаційного домену при створенні сокета вказуватиметься константа *socket.AF_INET*.

Як приклад розглянемо найпростішу клієнт-серверну пару.

Сервер прийматиме рядок та відповідатиме клієнту. Мережевий пристрій іноді називають хостом (host), тому цей термін буде вживатися стосовно комп'ютера, на якому працює мережева програма.

Серверна частина програми:

```
import socket, string

HOST = "127.0.0.1" # localhost
PORT = 33333
srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv.bind((HOST, PORT))

while True:
   print ("Слухаю порт 33333")
   srv.listen(1)
   sock, addr = srv.accept()
   while True:
      pal = sock.recv(1024)
      if not pal:
         break
      print ("Отримано від %s:%s:" % addr, pal)
      lap = 'The message has been processed'
      print ("Відправлено %s:%s:" % addr, lap)
      sock.send(b'ok, request')
   sock.close()
```

Клієнтська частина програми:

```
import socket, os
HOST = "127.0.0.1" # віддалений комп'ютер (localhost)
PORT = 33333# порт на віддаленому комп'ютері
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
sock.connect((HOST, PORT))
sock.send(b'Hello world!!!')
result = sock.recv(1024)
sock.close()
print ("Отримано:", result)
```

Насамперед, потрібно запустити сервер. Сервер відкриває сокет на локальній машині на порту 33333 та за адресою 127.0.0.1. Після цього він слухає (*listen()*) порт. Коли на порту з'являються дані, приймається (*accept()*) вхідне з'єднання. Метод *accept()* повертає пару – Socket-об'єкт та адресу віддаленого комп'ютера, що встановлює з'єднання (пара – IP-адреса, порт на віддаленій машині). Після цього можна застосовувати методи *recv()* та *send()* для спілкування із клієнтом. У *recv()* указується кількість байтів у черговій порції. Від клієнта може прийти й менша кількість даних.

Код програми-клієнта досить очевидний. Метод *connect()* установлює з'єднання з віддаленим хостом (у наведеному прикладі він розташований на тій самій машині). Дані передаються методом *send()* та приймаються методом recv() – аналогічно тому, як це відбувається на сервері.

Модуль *socket* має кілька допоміжних функцій. Зокрема, функції для роботи із системою доменних імен (DNS):

```
>>> import socket
>>> socket.gethostbyname('www.ukr.net')
'212.42.76.253'
>>> socket.gethostbyaddr('212.42.76.253')
('srv253.fwdcdn.com', [], ['212.42.76.253'])
>>> socket.gethostname()
'test-host'
```

Функція *socket.getservbyname()*, яка дозволяє перетворювати найменування інтернет-сервісів у загальноприйняті номери портів:

```
import socket, string

for srv in 'http', 'ftp', 'imap', 'pop3', 'smtp':
    print (socket.getservbyname(srv, 'tcp'), srv)
```

```
80 http
21 ftp
143 imap
```

110 pop3
25 smtp

9.2. Модуль smtplib

Повідомлення електронної пошти в Інтернеті передаються від клієнта до сервера та між серверами в основному за протоколом SMTP (Simple Mail Transfer Protocol, простий протокол передавання пошти). Протоколи SMTP та ESMTP (розширений варіант SMTP) описано в RFC 821 та RFC 1869. Для роботи з SMTP у стандартній бібліотеці модулів є модуль *smtplib*. Для того, щоб почати SMTP-з'єднання із сервером електронної пошти, необхідно на початку створити об'єкт для керування SMTP-сесією за допомогою конструктора класу SMTP:

smtplib.SMTP([host[, port]])

Параметри *host* та *port* визначають адресу та порт SMTP-сервера, через який відправлятиметься пошта. За замовчуванням , port = 25. Якщо host задано, конструктор сам установить з'єднання, інакше доведеться окремо викликати метод *connect()*. Екземпляри класу SMTP мають методи для всіх розповсюджених команд SMTP-протоколу, але для відправлення пошти достатньо виклику конструктора та методів *sendmail()* та *quit()*:

```
# -*- coding: cp1251 -*-
from smtplib import SMTP
fromaddr = "timurdov@ukr.net" # Від кого
toaddr = "antoniy_kovalsky@ukr.net"# Кому
message = """From: Student <%(fromaddr)s>
To: Lecturer <%(toaddr)s>
Subject: From Python course student
MIME-Version: 1.0
Content-Type: text/plain; charset=Windows-1251
Content-Transfer-Encoding: 8bit
Добрий день! Я вивчаю курс з мови Python і
відправляю лист його авторові.
"""
connect = SMTP('mail.ukr.net')
connect.set_debuglevel(1)
connect.sendmail(fromaddr, toaddr, message % vars())
connect.quit()
```

Зазначимо, що *toaddr* у повідомленні (у полі To) та при відправленні можуть не збігатися тому, що параметри (одержувач та відправник) у ході SMTP-сесії

передаються командами SMTP-протоколу. При запуску зазначеного вище прикладу на екрані повинна з'явиться налагоджувальна інформація (адже рівень налагодження вказаний рівним 1)

Під час однієї SMTP-сесії можна надіслати відразу кілька листів поспіль, якщо не викликати quit().

Команди SMTP можна подавати й окремо: для цього в об'єкта-з'єднання є методи *(helo(), ehlo(), expn(), help(), mail(), rcpt(), vrfy(), send(), noop(), data()*), що відповідають однойменним командам SMTP-протоколу.

При роботі з класом smtplib.SMTP можуть генеруватися різні виняткові ситуації. Призначення деяких із них наведено нижче:

- smtplib.SMTPException – базовий клас для всіх виняткових ситуацій модуля;
- smtplib.SMTPServerDisconnected – сервер неочікувано перервав зв'язок (або зв'язок із сервером не було встановлено);
- smtplib.SMTPResponseException – базовий клас для всіх виняткових ситуацій, які мають код відповіді SMTP-сервера;
- smtplib.SMTPSenderRefused – відправника відкинуто;
- smtplib.SMTPRecipientsRefused – сервер відкинув усіх отримувачів;
- smtplib.SMTPDataError – сервер відповів невідомим кодом на повідомлення;
- smtplib.SMTPConnectError – помилка встановлення з'єднання;
- smtplib.SMTPHeloError – сервер не відповів правильно на команду HELO або відкинув її.

9.3. Модуль poplib

Ще один протокол – POP3 (Post Office Protocol, поштовий протокол) служить для прийому пошти з поштової скриньки на сервері (протокол визначено у RFC 1725).

Для роботи з поштовим сервером потрібно встановити з ним з'єднання і, подібно до розглянутого вище прикладу, за допомогою SMTP-команд отримати необхідні повідомлення. Об'єкт з'єднання POP3 можна встановити за допомогою конструктора класу POP3 із модуля *poplib*:

poplib.POP3(host[, port])

де *host* – адреса POP3-сервера, *port* – порт на сервері (за замовчуванням 110), *pop_obj* – об'єкт для керування сеансом роботи з POP3-сервером.

Наступний приклад демонструє основні методи для роботи з POP3-з'єднанням (для того, щоб приклад спрацював коректно, необхідно внести реальні облікові дані користувача на сервері):

```
import poplib, email
# Облікові дані користувача:
SERVER = "pop.server.com"
USERNAME = "user"
USERPASSWORD = "secretword"
p = poplib.POP3(SERVER)
print (p.getwelcome())
# етап ідентифікації
print (p.user(USERNAME))
print (p.pass_(USERPASSWORD))
# етап транзакцій
response, lst, octets = p.list()
print (response)
for msgnum, msgsize in [i.split() for i in lst]:
   print ("Повідомлення %(msgnum)s має довжину% (msgsize)s" % vars())
   print ("UIDL =", p.uidl(int(msgnum)).split()[2])
   if int(msgsize) > 32000:
      (resp, lines, octets) = p.top(msgnum, 0)
   else:
      (resp, lines, octets) = p.retr(msgnum)
   msgtxt = "\n".join(lines)+"\n\n"
   msg = email.message_from_string(msgtxt)
   print ("* Від: %(from)s\n* Кому: %(to)s\n* Тема:%(subject)s\n" % msg)
   # msg містить заголовки повідомлення або все повідомлення (якщо воно невелике)
   # етап відновлення

print (p.quit())
```

Ці й інші методи екземплярів класу POP3 описано в табл. 9.1.

Таблиця 9.1 – екземпляри класу POP3

Метод	Команда POP3	Опис
getwelcome()		Отримує рядок s із вітанням POP3-сервера
user(name)	**USER name**	Надсилає команду USER, вказуючи ім'я користувача name. Повертає рядок із відповіддю сервера

pass_(pwd)	**PASS pwd**	Надсилає пароль користувача в команду PASS. Після цієї команди і до виконання команди QUIT поштова скринька блокується
apop(user, secret)	**APOP user secret**	Виконує ідентифікацію на сервері за APOP
rpop(user)	**RPOP user**	Здійснює ідентифікацію за методом RPOP
stat()	**STAT**	Повертає кортеж з інформацією про поштову скриньку. У ньому m – кількість повідомлень, l – розмір поштової скриньки в байтах
list([num])	**LIST [num]**	Повертає список повідомлень у форматі (resp, ['num octets', ...]), якщо не вказано num, і "+OK num octets", якщо вказано. Список lst складається з рядків у форматі "num octets"
retr(num)	**RETR num**	Завантажує з сервера повідомлення з номером num та повертає кортеж із відповіддю сервера (resp, lst, octets)
dele(num)	**DELE num**	Видаляє повідомлення з номером num
rset()	**RSET**	Скасовує позначки видалення повідомлень
noop()	**NOOP**	Не виконує функцій (підтримує з'єднання)
quit()	**QUIT**	Від'єднує від сервера. Сервер виконує всі необхідні зміни (видаляє повідомлення) та знімає блокування поштової скриньки
top(num, lines)	**TOP num lines**	Команда аналогічна до RETR, але завантажує тільки заголовок та lines рядків тіла повідомлення. Повертає кортеж (resp, lst, octets)
uidl([num])	**UIDL [num]**	Виконує скорочення від "unique-id listing" (список унікальних ідентифікаторів повідомлень). Формат результату: (resp, lst, octets), якщо num не вказано, і "+OK num uniqid", якщо вказано.

		Список lst складається з рядків вигляду "+OK num uniqid"

У цій таблиці *num* означає номер повідомлення (він не змінюється протягом усієї сесії); *resp* – відповідь сервера, повертається для будь-якої команди, починається з "+OK " для успішних операцій (при невдачі генерується виняткова ситуація виняткові ситуації *poplib.proto_error*). Параметр *octets* визначає кількість байтів у прийнятих даних, *uniqid* – ідентифікатор повідомлення, генерується сервером.

Робота з POP3-сервером складається з трьох фаз: ідентифікації, транзакцій та відновлення. На етапі ідентифікації відразу після створення POP3-об'єкта дозволено тільки команди USER, PASS (іноді APOP та RPOP). Після ідентифікації сервер отримує інформацію про користувача і наступає етап транзакцій. Тут доступні інші команди. Етап відновлення викликається командою QUIT, після якої POP3-сервер обновляє поштову скриньку користувача відповідно до поданих команд, а саме – видаляє позначені для видалення повідомлення.

9.4. Модулі для клієнта www

Стандартні засоби мови Python дозволяють отримувати з програми доступ до об'єктів www як у простих випадках, так і за складних обставин, зокрема при необхідності передавати дані форми, ідентифікації, доступу через проксі тощо.

Зазначимо, що при роботі з www використовується здебільшого протокол HTTP, однак www охоплює не тільки HTTP, але й багато інших схем (FTP, HTTPS тощо). Використовувана схема зазвичай вказується на самому початку URL.

9.4.1. Функції для завантаження мережевих об'єктів

Простий випадок отримання веб-об'єкта за відомим URL наведено в такому прикладі:

```
import urllib.request as req
doc = req.urlopen("http://python.org").read()
print (doc[:40])
```

Функція *urllib.urlopen()* створює файлоподібний об'єкт, який можна читати методом *read()*. Інші методи цього об'єкта: *readline(), readlines(), fileno(), close()* працюють як і у звичайного файла, а також є метод *info()*, що повертає message-об'єкт, відповідний отриманим із сервера даним. Цей об'єкт можна використовувати для одержання додаткової інформації:

```
import urllib.request as req
```

```
f = req.urlopen("http://python.org")
print (f.info())
```

Результатом буде:

```
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
x-xss-protection: 1; mode=block
X-Clacks-Overhead: GNU Terry Pratchett
Via: 1.1 varnish
Fastly-Debug-Digest:
a63ab819df3b185a89db37a59e39f0dd85cf8ee71f54bbb42fae41670ae56fd2
Content-Length: 48844
Accept-Ranges: bytes
Date: Sun, 11 Mar 2018 14:02:13 GMT
Via: 1.1 varnish
Age: 1458
Connection: close
X-Served-By: cache-iad2120-IAD, cache-hhn1531-HHN
X-Cache: HIT, HIT
X-Cache-Hits: 3, 16
X-Timer: S1520776934.559399,VS0,VE0
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains
```

За допомогою функції *urllib.urlopen()* можна робити і складніші речі, наприклад, передавати веб-серверу дані форми. Як відомо, дані заповненої веб-форми можуть бути передані на вебсервер із використанням методу GET або методу POST. Метод GET пов'язаний із кодуванням усіх переданих параметрів після знака "?" в URL, а при методі POST дані передаються в тілі HTTP-запиту. Обидва варіанти передавання наведено нижче:

```
import string, urllib.parse  as par
import urllib.request as req
data = {"q": "Python"}
enc_data = par.urlencode(data)
# метод GET
f = req.urlopen("http://searchengine.com/search" + "?" + enc_data)
print (f.read())
# метод POST
```

```
f = req.urlopen("http://searchengine.com/search", str.encode(enc_data))
print (f.read())
```

Функція *urlretrieve()* дозволяє записати вказаний URL мережевий об'єкт у файл. Вона має такі параметри:

urllib.urlretrieve(url[, filename[, reporthook[, data]]]).

Тут url – URL мережевого об'єкта; filename – ім'я локального файла для розміщення об'єкта; reporthook – функція, що буде викликатися для повідомлення про стан завантаження; data – дані для методу POST (якщо він використовується). Функція повертає кортеж (filepath, headers), де filepath – ім'я локального файла, у який завантажено об'єкт; headers – результат методу info() для об'єкта, який було повернуто urlopen().

Для забезпечення інтерактивності функція urllib.urlretrieve() викликає час від часу функцію, указану в reporthook. Цій функції передаються три аргументи: кількість прийнятих блоків, розмір блоку та загальний розмір прийнятого об'єкта в байтах (якщо він невідомий, цей параметр дорівнює –1).

9.5. XML-RPC-сервер

Дотепер високорівневі протоколи розглядалися з погляду клієнта. Не менш просто створювати на Python і їхні серверні частини. Для ілюстрації того, як розробити програму на Python, яка реалізує сервер, обрано протокол XML-RPC. Незважаючи на свою назву, кінцевому користувачу не обов'язково знати XML, тому що її використання приховано від нього. Скорочення RPC (Remote Procedure Call, виклик віддаленої процедури) пояснює суть справи: за допомогою XML-RPC можна викликати процедури на віддаленому хості. Причому за допомогою XML-RPC можна абстрагуватися від конкретної мови програмування за рахунок використання загальноприйнятих типів даних (рядки, числа, логічні значення тощо). У мові Python виклик віддаленої функції за синтаксисом нічим не відрізняється від виклику звичайної функції:

```
import xmlrpc.client as cl
# Встановити з'єднання
req = cl.ServerProxy("http://localhost:8000")
try:
# Викликати віддалену функцію
   print (req.add(1, 3))
except cl.ProtocolError as err:
   print ("ERROR:: %s", err)
```

А ось як виглядає XML-RPC-сервер (для того, щоб перевірити роботу наведеного вище прикладу, необхідно спочатку запустити сервер):

```
from xmlrpc.server import SimpleXMLRPCServer

srv = SimpleXMLRPCServer(("localhost", 8000))# Запустити сервер
srv.register_function(pow)# Зареєструвати функцію
srv.register_function(lambda x,y: x+y, 'add')# І ще одну
srv.serve_forever()# Обслуговувати запити
```

За допомогою XML-RPC (а цей протокол досить "легкий" порівняно з іншими протоколами такого призначення) програми можуть спілкуватися одна з одною на зрозумілій їм мові виклику функцій із параметрами основних загальноприйнятих типів та таких самих значень, що повертаються. Перевагою в Python є зручний синтаксис виклику віддалених функцій.