<웹 프로그래밍>

장고 공식 홈페이지: <https://docs.djangoproject.com/>

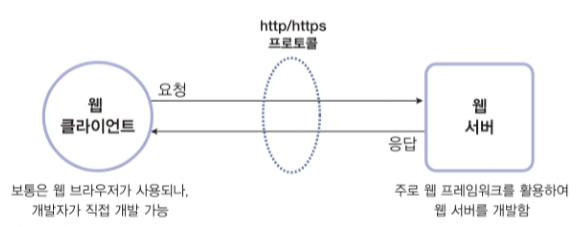
예제 소스 다운로드: [www.hanbit.co.kr/src/10104](http://www.hanbit.co.kr/src/10104)

**1. 웹 프로그래밍의 이해**

웹 프로그램: 기본적으로 클라이언트-서버로 이루어짐

HTTP(S) 프로토콜로 통신하는 클라이언트와 서버를 개발하는 것

웹 클라이언트(웹 브라우저)와 웹 서버(네이버 서버) 중 보통 웹 서버를 개발



다양한 웹 클라이언트

1)웹 브라우저를 사용하여 요청

웹 브라우저의 주소창에 접속하고자 하는 웹 서버의 URL 입력([www.example.com](http://www.example.com))

웹 브라우저는 주소창에 입력된 문장을 해석하여 웹 서버에게 HTTP 요청을 보내는 웹 클라이언트 역할을 수행. 요청을 받은 URL 도메인의 웹 서버는 그 결과를 웹 브라우저로 전송. 웹 브라우저는 전송 받은 결과를 사용자가 볼 수 있도록 HTML 텍스트를 해석하여 화면에 보여줌.

2) 리눅스 curl 명령을 사용하여 요청

리눅스 curl 명령: HTTP/HTTPS/FTP 등 여러 가지의 프로토콜을 사용하여 데이터를 송수신할 수 있는 명령.

$ curl <http://www.example.com>

curl 명령: 인자로 넘어온 URL로 HTTP 요청을 보내는 웹 클라이언트의 역할을 수행함.



웹 브라우저에서 보았던 문장과 동일함. (HTML 형식)

즉 어떤 방법을 사용하던 웹 서버는 동일한 요청을 받을 경우 동일한 응답을 주고 있음.

3) Telnet을 사용하여 요청

리눅스의 telnet 프로그램 사용하여 HTTP 요청 보냄.

[shkim@localhost ch1]$ telnet [www.example.com](http://www.example.com) 80

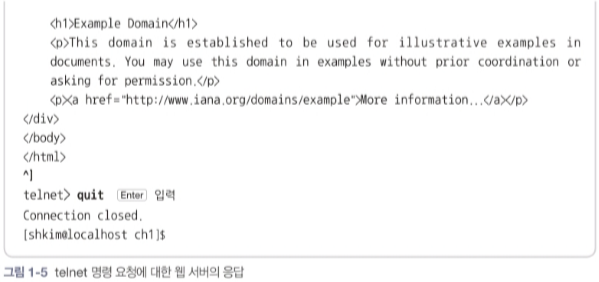
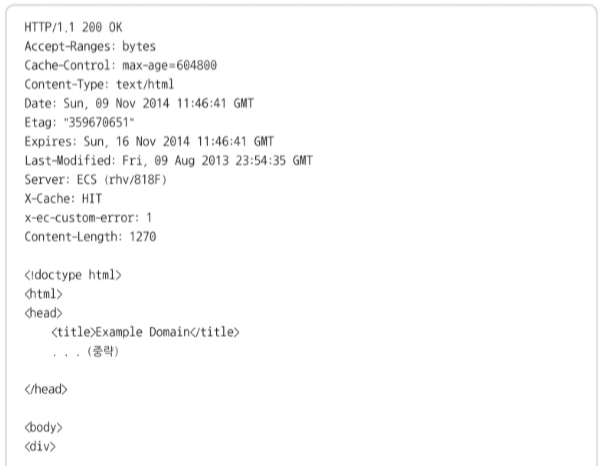
…

GET / HTTP/1.1

Host:www.example.com

telnet 명령은 터미널 창에서 입력하는 내용을 그대로 웹 서버에 전송함.

위 입력 내용은 HTTP 프로토콜의 요청 메시지 규격에 정의된 규칙에 따라 HTTP 요청을 보내는 것으로 telnet 프로그램이 웹 클라이언트의 역할을 수행. 도메인의 웹 서버는 결과 응답.



역시 동일한 출력. (telnet 프로그램 특성상 HTTP 응답 메시지의 헤더가 화면에 출력)

4) 직접 만든 클라이언트로 요청

-notepad 텍스트 에디터

C:\RedBook\ch1>notepad example.py

import urllib.request

print(urllib.request.urlopen(“<http://www.example.com>”).read().decode(‘utf-8’))



HTTP 프로토콜(Hypertext Transfer Protocol)

HTTP는 웹 서버와 웹 클라이언트 사이에서 데이터를 주고받기 위해 사용하는 통신 방식

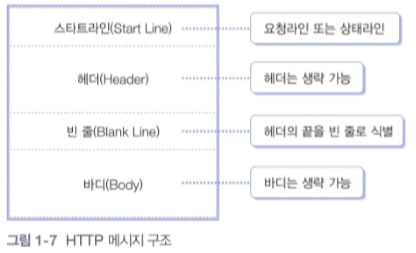
TCP/IP 프로토콜 위에서 동작. 즉, 우리가 웹을 이용하려면 웹 서버와 웹 클라이언트는 각각 TCP/IP 동작에 필수적인 IP 주소를 가져야 함.

HTTP: HTML이나 XML 같은 하이퍼텍스트 뿐 아니라 이미지, 음성, 동영상, 자바스크립트, PDF와 각종 오피스 도큐먼트 파일 등 모든 데이터 전송 가능

우리가 주소창에 주소를 입력하고 엔터를 입력하면 웹 클라이언트와 웹 서버 사이에 HTTP 연결이 맺어지고 서로 요청 메시지와 응답 메시지가 반복적으로 오가면서 웹 사용.

1) HTTP 메시지의 구조

HTTP 메시지: 요청 메시지(클라이언트->서버), 응답 메시지(서버->클라이언트)



스타트라인: 요청라인(요청 메시지일 때), 상태라인(응답 메시지일 때)

헤더: 각 행의 끝에 줄 바꿈 문자인 CRLF가 있음

빈 줄: 헤더와 바디의 구분

헤더와 바디: 생략 가능

바디: 텍스트 뿐 아니라 바이너리 데이터도 들어갈 수 있음

<바디가 없는 요청 메시지의 예>

GET /book/Shakespeare HTTP/1.1

Host: [www.example.com:8080](http://www.example.com:8080)

첫번째 줄: 요청라인

요청 방식, 요청 URL, 프로토콜 버전으로 구성

두번째 줄: 헤더

이름: 값 형식

\*\*Host 항목은 필수로 표시해줘야 함

(요청라인의 URL에 Host를 표시하면 Host 헤더 생략 가능)

GET [http://www.example.com:8080/book/shakespeare HTTP/1.1](http://www.example.com:8080/book/shakespeare%20HTTP/1.1)

<응답 메시지의 예>

HTTP/1.1 200 OK

Content-Type: application/xhtml+xml; charset=utf-8

<html>

…

</html>

첫번째 줄: 상태라인

프로토콜 버전, 상태코드, 상태 텍스트로 구성.

(위 예시에서 200 OK->정상적으로 처리되었음.)

두번째 줄: 헤더

빈 줄

바디: HTML 텍스트 포함

2) HTTP 처리 방식

HTTP 메소드를 통해서 클라이언트가 원하는 처리 방식을 서버에 알려줌.

<HTTP 메소드>

|  |  |  |
| --- | --- | --- |
| 메소드명 | 의미 | CRUD와 매핑되는 역할 |
| GET | 리소스 취득 | Read(조회) |
| POST | 리소스 생성, 리소스 데이터 추가 | Create(생성) |
| PUT | 리소스 변경 | Update(변경) |
| DELETE | 리소스 삭제 | Delete(삭제) |
| HEAD | 리소스의 헤더 취득 |  |
| OPTIONS | 리소스가 서포트하는 메소드 취득 |  |
| TRACE | 루프백 시험에 사용 |  |
| CONNECT | 프록시 동작의 터널 접속으로 변경 |  |

GET 방식: 지정한 URL의 정보를 가져오는 메소드(가장 많이 사용). 웹 브라우저를 이용하여 서버로부터 웹 페이지, 이미지, 동영상 등을 가려오려고 할 때 GET 방식의 요청 사용.

POST 방식: POST의 대표적인 기능은 리소스 생성. 블로그에 글을 등록하는 경우.

PUT 방식: 리소스를 변경하는 데 사용.

예) 블로그에서 글을 업로드한 작성자를 변경하거나 글의 내용을 업데이트

\*리소스를 생성하는 데도 사용 가능.

POST와의 차이점: -POST: 새롭게 생성한 리소스에 대한 URL 결정권이 서버에 있을 때

-PUT: 결정권이 클라이언트에 있을 때

예) 트위터에 글을 포스팅하면 그 글에 대한 URL: 서버에서 결정-POST

위키: 클라이언트가 결정한 타이틀이 그대로 URL이 됨: PUT

BUT 그냥 리소스의 생성은 POST, 리소스의 변경은 PUT

DELETE 방식: 리소스 삭제 메소드

3) GET과 POST 메소드

HTML의 폼에서 지정할 수 있는 메소드가 GET과 POST 밖에 없음

폼에서 사용자가 입력한 데이터들을 서버로 보낼 때

GET: URL 부분의 ? 뒤에 이름=값 쌍으로 이어 붙여 보냄.

GET <http://docs.djangoproject.com/search/?q=forms&release=1> HTTP/1.1

POST: GET에서 URL에 포함시켰던 파라미터들을 아래 예시처럼 요청 메시지의 바디에 넣음

POST <http://docs.djangoproject.com/search/> HTTP/1.1

Content-Type: application/x-www-form-urlencoded

q=forms&release=1

\*\*URL의 길이 제한이 있기 때문에 GET 방식을 이용하면 많은 양의 데이터를 보내기가 어려움

또한 전달되는 사용자의 데이터가 웹 브라우저의 주소창에 노출되므로 보안 상 단점

(네이버-GET 방식 사용됨을 확인할 수 있음)

4) 상태 코드

서버에서의 처리 결과는 응답 메시지의 상태라인에 있는 상태 코드를 보고 파악 가능함.

상태 코드는 세 자리 숫자로 되어 있음.

첫 번째 숫자는 HTTP 응답의 종류를 구분하는 데 사용, 나머지 두 개의 숫자는 세부적인 응답 내용의 구분을 위한 번호.



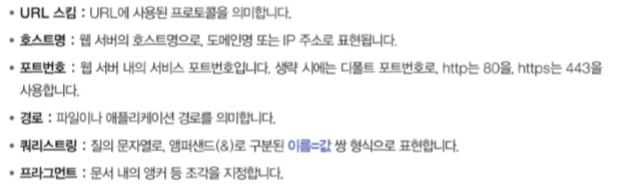
<URL 설계>

디자인: 화면 UI 설계

프로그램 로직: URL 설계

차후에 로직이 변경되더라도 URL 변경은 최소화할 수 있도록 설계!

URL의 구성 



<URL을 바라보는 측면>

URL: 웹 클라이언트에서 호출하므로 애플리케이션에 대한 API라고 할 수 있음.

-RPC/REST로 바라보는 방식

-RPC: 클라이언트가 네트워크상에서 원격에 있는 서버가 제공하는 API 함수를 호출하는 방식

URL 설계와 API 설계를 동일하게 고려하여 URL 경로를 API 함수명으로, 쿼리 파라미터를 함수의 인자로 간주

웹 클라이언트에서 URL을 전송하는 것=웹 서버의 API 함수를 호출한다고 인식

\*URL 경로의 대부분이 동사가 됨

ex) <http://blog.example.com/search?q=test&debug=true>

-REST: 웹 서버에 존재하는 요소들을 모두 리소스라고 정의하고 URL을 통해 웹 서버의 특정

리소스를 표현한다는 개념

리소스는 시간이 지남에 따라 상태가 변할 수 있으므로 서버 간 데이터 교환을 리소스

상태의 교환으로 간주

\*리소스에 대한 조작을 GET, POST, PUT, DELETE 등의 HTTP 메소드로 구분

웹 클라이언트에서 URL을 전송하는 것=웹 서버에 있는 리소스 상태에 대한 데이터를 주고받는 것

ex) <http://blog.example.com/search/test> #GET 메소드 사용

<간편 URL>

쿼리스트링 없이 경로만 가진 간단한 구조의 URL

검색 엔진의 처리를 최적화하기 위해 생겨난 간편한 URL은 URL 입력과 기억이 쉬움



<파이썬의 우아한 URL>

장고에서 사용하는 URL 표현 방식

urlpatterns=[

path(‘articles/2003/’, views.special\_case\_2003),

path(‘articles/<int:year>/’, views.year\_archieve),

path(‘articles/<int:year>/<int:month>/’, views.month\_archieve),

path(‘articles/<int:year>/<int:month>/<slug:slug>/’, views.article\_detail),

]

\*위 표현을 정규표현식을 사용하여 표현할 수도 있음

<웹 애플리케이션 서버>

웹 서버: 웹 클라이언트(웹 브라우저)의 요청을 받아서 처리하는 서버

-웹 서버: 웹 클라이언트의 요청을 받아 요청을 처리하고 그 결과를 웹 클라이언트에게 응답

정적 페이지인 HTML, 이미지, CSS, 자바스크립트 파일을 웹 클라이언트에 제공할 때 웹

서버 사용.

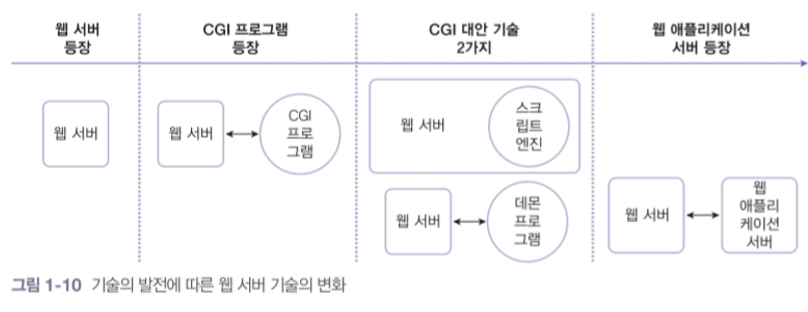
동적 페이지 처리가 필요하다면 웹 애플리케이션 서버에 처리를 넘김.

-프로그램: Apache httpd, Nginx, lighttpd, IIS 등

-웹 애플리케이션 서버: 웹 서버로부터 동적 페이지를 요청 받아서 요청을 처리하고 그 결과를 서버로 반환

동적 페이지 생성을 위한 프로그램 실행과 데이터베이스 연동 기능을 처리

-프로그램: Apache Tomcat, JBoss, WebLogic, WebSphere, Jetty, Jeus, mod\_wsgi, Uwsgi, Gunicorn 등



<정적 페이지 vs 동적 페이지>

정적 페이지: 누가, 언제 요구하더라도 항상 같은 내용을 표시하는 웹 페이지

해당 웹 서비스의 제공자가 사전에 준비하여 서버 측에 배치한 것

동일한 리소스(URL)의 요청-항상 동일한 내용의 페이지 반환

주로 HTML, 자바스크립트, CSS, 이미지만으로 이루어진 페이지 해당

동적 페이지: 동일한 리소스의 요청이라도 누가, 언제, 어떻게 요구했는지에 따라 각각 다른 내용

이 반환되는 페이지

예를 들어 현재 시각을 보여주는 페이지, 온라인 쇼핑 사이트-장바구니 페이지

동적 페이지에는 프로그래밍 코드가 포함되어 있어 페이지 요청 시점에 HTML 문장 을 만들어 냄

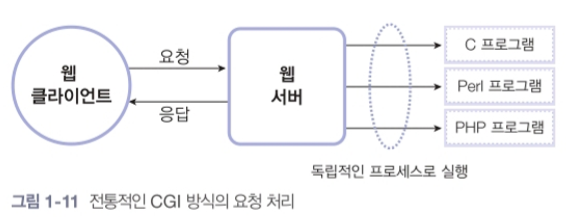
동적 페이지에 대한 요구사항이 생김에 따라 필요한 데이터를 저장하고 꺼내 오는 등의 데이터베이스 처리에 대한 요구가 많아져 웹 서버와는 다른 별도의 프로그램 필요해짐

별도의 프로그램과 웹 서버 사이에 정보를 주고받는 규칙을 정의한 것: CGI 규격

<CGI 방식의 단점>

\*CGI 자체는 정식 프로그래밍 언어나 스크립트가 아님. 정보를 주고받는 규격!

이 규격을 준수하면 어떤 언어를 사용해도 CGI 프로그램 개발 가능



CGI 방식의 근본적인 문제점: 각각의 클라이언트 요청에 대하여 독립적인 별도의 프로세스가 생 성. 요청이 많아질수록 프로세스가 많아지고, 메모리 요구량이 커져

시스템에 많은 부하를 줌.

<CGI 방식의 대안 기술>

1. 별도의 애플리케이션(CGI 프로그램과 같은 역할을 하는 프로그램)을 Perl, PHP 등의 스크립트 언어로 작성, 스크립트를 처리하는 스크립트 엔진을 웹 서버에 내장시켜서 CGI 방식의 단점이었던 별도의 프로세스를 가동시키는 오버헤드를 줄이는 방식.

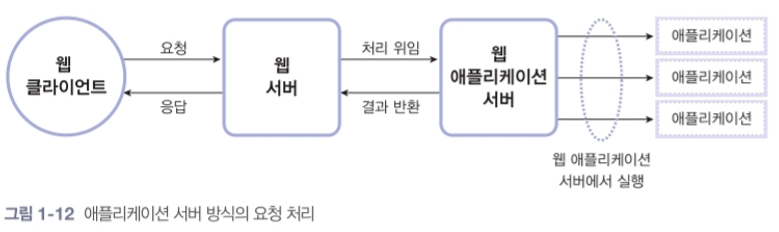
2. 애플리케이션을 처리하는 프로세스를 미리 데몬으로 가동시켜 놓고 웹 서버의 요청을 데몬에서 처리.

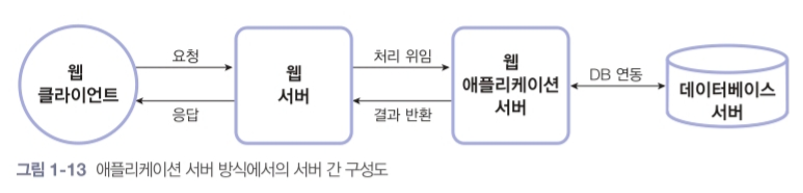
\*파이썬-mod\_wsgi 모듈: 웹 서버 내장 방식, 별도의 데몬 방식으로 실행 가능

현재 가장 많이 사용되고 있는 JSP, ASP 기술에서 애플리케이션 서버 방식을 사용하고 있음

<애플리케이션 서버 방식>

웹 서버가 직접 프로그램을 호출하기보다 웹 애플리케이션 서버를 통해 간접적으로 웹 애플리케이션 프로그램 실행.





웹 서버와 웹 애플리케이션 서버가 분리됨에 따라 서로의 역할도 구분하여 사용하는 것이 좋음

웹 서버는 정적 페이지만 처리하고 웹 애플리케이션 서버는 동적 페이지만 처리하도록.

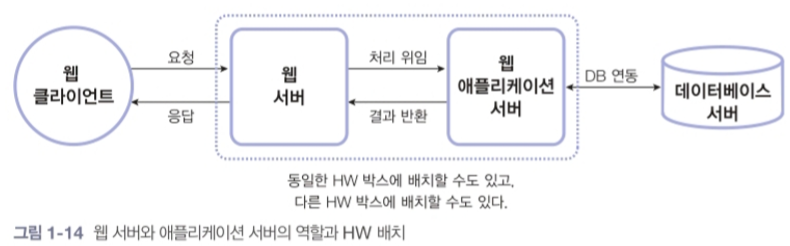
웹 서버: 캐시 기능, 프록시 기능 등의 추가적인 기능 제공

다수의 클라이언트로부터 동시에 요청 받아 처리해야 하므로 동시에 접속을 허가하는 클라이언트 수의 제한 및 처리 프로세스 관리, 요청 및 응답에 관한 로그의 기록, 안정성 확보를 위한 인증 제어 및 암호화 처리 등 다양한 기능

웹 애플리케이션 서버: 웹 서버와의 연동 규격을 잘 따르기만 하면 임의의 언어 플랫폼을 사용해서 애플리케이션 프로그램을 작성하고 실행시킬 수 있기 때문에 기능 더 추가됨

<웹 서버와의 역할 구분>

하드웨어 측면의 용어: 웹 서버 박스, 웹 애플리케이션 서버 박스



HW 박스를 분리하여 구성하면 메모리 효율을 더 높일 수 있음.

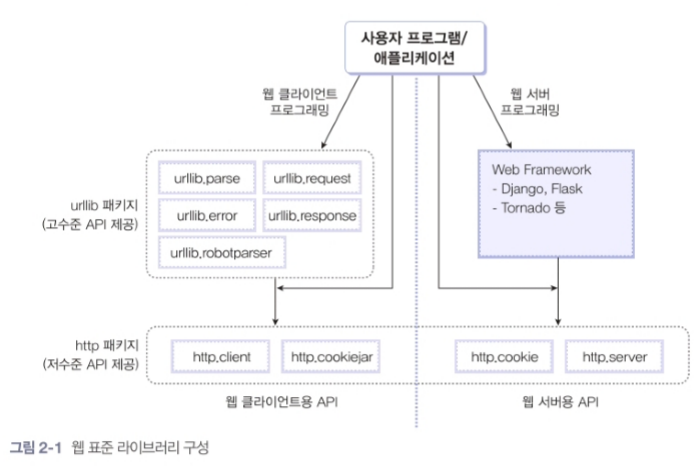
웹 서버 박스와 웹 애플리케이션 서버 박스 간의 메모리 사이즈 비율을 조절.(요청 건수 분석)

**2. 파이썬 웹 표준 라이브러리**

장고: 웹 서버 프레임워크

파이썬을 설치하면 기본적으로 같이 설치되는 표준 라이브러리

<웹 라이브러리 구성>



urllib 패키지에는 웹 클라이언트를 작성하는 데 사용되는 모듈들이 있음.

http 패키지는 크게 서버용과 클라이언트용 라이브러리로 나누어 모듈을 담고 있음.

웹 클라이언트를 개발하는 경우-urllib 패키지 사용

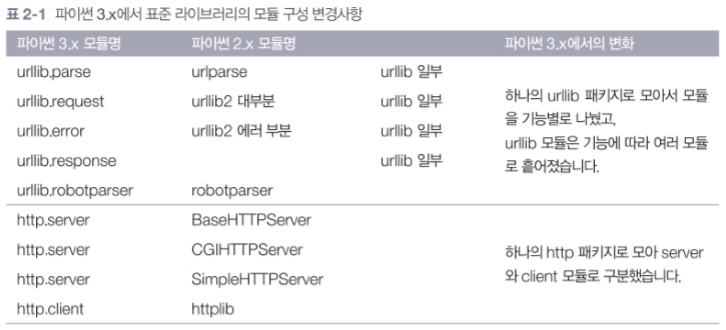
: HTTP 서버뿐만 아니라 FTP 서버 및 로컬 파일 처리

클라이언트에서 공통적으로 필요한 함수와 클래스 등 제공

주로 URL 처리와 서버 액세스 관련 API 제공

웹 서버 프로그래밍: http.cookie 모듈이나 http.server 모듈은 거의 사용할 일이 없음.

웹 프레임워크는 사용자 프로그램과 저수준의 http.server 라이브러리 중간에 위치하여 웹 서버의 애플리케이션 개발을 좀 더 편리하게 해주면서 표준 라이브러리의 기능을 확장해줌



<웹 클라이언트 라이브러리>

웹 브라우저 이외의 웹 클라이언트를 개발하고 사용하는 경우

: 날씨 정보를 제공하는 사이트에서 내가 사는 지역의 날씨 정보를 가져오는 프로그램

트위터 사이트에서 대한민국이라는 단어가 포함된 트윗 메시지를 가져오는 프로그램

웹 사이트의 페이지를 구성하는 HTML 소스를 가져오는 웹 크롤링 프로그램 등

트위터나 구글 같은 인터넷 서비스를 제공하는 회사들은 외부의 프로그램들이 자신의 서비스를 이용할 수 있도록 OpenAPI 제공.

\*\*웹 클라이언트와 서버 간에 HTTP(s) 프로토콜을 사용하여 통신

->프로그램을 HTTP 프로토콜을 사용하여 개발

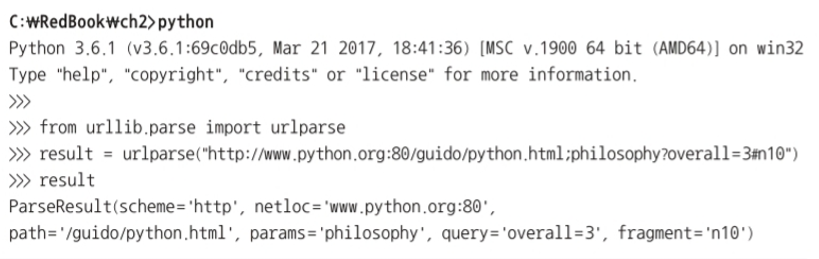
1) urllib.parse 모듈

URL의 분해, 조립, 변경 및 URL 문제 인코딩, 디코딩 처리 함수 제공

기본 함수인 urlparse() 함수

from urllib.parse import urlparse

result=urlparse(<http://www.python.org:80/guido/python.html;philosophy?overall=3#n10>)

urlparse() 함수는 URL을 파싱한 결과로 ParseResult 인스턴스를 반환.

-scheme: URL에 사용된 프로토콜을 으미

-netloc: 네트워크 위치, user:password@host:port 형식으로 표현(HTTP 프로토콜-host:port 형식)

-path: 파일이나 애플리케이션 경로를 의미

-params: 애플리케이션에 전달될 매개변수(현재는 사용 X)

-fragment: 문서 내의 앵커 등 조각을 지정

\*urlparse() 함수 이외에도 urlsplit(), urljoin(), parse\_qs(), quote(), encode() 함수 등이 있음

설명: <https://docs.python.org/3/library/urllib.parse.html#module-urllib.parse>

2) urllib.request 모듈

주어진 URL에서 데이터를 가져오는 기본 기능 제공

기본 함수인 urlopen() 함수

urlopen(url, data=None, [timeout])

-url 인자로 지정한 URL로 연결하고 유사 파일 객체를 반환

-url 인자는 문자열이거나 Request 클래스의 인스턴스가 올 수 있음

-url에 file 스킴을 지정하면 로컬 파일 열기 가능

-디폴트 요청 방식은 GET이고, 웹 서버에 전달할 파라미터가 있으면 질의 문자열을 url 인자에 포함해서 보냄

-요청 방식을 POST로 보내고 싶으면 data 인자에 질의 문자열을 지정해주면 됨

-옵션인 timeout은 응답을 기다리는 타임아웃 시간을 초로 표시

urlopen() 함수 사용 방법

-사용 케이스: URL로 GET/POST 방식의 간단한 요청 처리: urlopen() 함수만으로 가능

PUT, HEAD 메소드 등 헤더 조직이 필요한 경우: Request 클래스를 같이 사용

인증, 쿠키, 프록시 등 복잡한 요청 처리: 인증/쿠키/프록시 해당 핸들러 클래스

같이 사용

from urllib.request import urlopen

f = urlopen(“http://www.example.com”)

print(f.read(500).decode(‘utf-8’)

->웹 브라우저의 주소창에 [www.example.com](http://www.example.com)이라고 입력하는 것과 동일한 데이터를 웹 서버로부터 가져옴. (웹 브라우저는 HTML 형식의 데이터를 해석하여 화면에 보기 좋게 보여주지만 파이썬 프로그램은 HTML 형식의 데이터를 해석하지 않고 그대로 보여줌) 위 프로그램은 HTTP GET 방식을 디폴트로 사용하여 웹 서버에 요청을 보냄.

POST 방식으로 웹 서버에 요청을 보내는 방법

urlopen() 함수를 호출 시 data 인자를 지정해주면 함수는 자동으로 POST 방식으로 요청을 보냄. data 인자는 URL에 허용된 문자열로 인코딩되어야 하고 유니코드(str) 타입이 아니라 바이트 스트링(bytes) 타입이어야 함.



from urllib.request import urlopen

data = ”language=python&framework=django”

f = urlopen(<http://127.0.0.1:8000>, bytes(data, encoding=’utf-8’))

print(f.read(500).decode(‘utf-8’))

\*요청을 보낼 때 요청 헤더를 지정해서 보내고 싶은 경우: URL을 지정하는 방식을 변경

url 인자에 문자열 대신에 Request 객체를 지정

즉 Request 객체를 생성하고 add\_header()로 헤더를 추가하여 웹 서버로 요청을 보내면 됨

[Request 클래스로 요청 헤더 지정]

from urllib.request import urlopen, Request

from urllib.parse import urlencode

url = ‘http://127.0.0.1:8000’

data={

‘name’: ‘김석훈’,

‘email’: ‘shkim@naver.com’

‘url’: ‘http://www.naver.com’,

}

encData=urlencode(data)

postData=bytes(encData, encoding=’utf-8’)

req=Request(url, data=postData)

req.add\_header(‘Content-Type’, ‘application/x-www-form-urlencoded’)

f=urlopen(req)

print(f.info())

print(f.read(500).decode(‘utf-8’))

인증 데이터나 쿠키 데이터 추가하여 요청 보내기

(각 기능에 맞는 핸들러 객체 정의, build\_opener() 함수를 사용하여 오프너에 등록, open() 함수 호출하면 서버로 요청 전송)

[urlopen() 함수-HTTPBasicAuthHandler 클래스로 인증 요청]

from urllib.request import HTTPBasicAuthHandler, build\_opener

auth\_handler = HTTPBasicAuthHandler()

auth\_handler.add\_password(realm=’ksh’, user=’shkim’, passwd=’shkimadmin’, url=’http://127.0.0.1:8000/auth/’)

opener = build\_opener(auth\_handler)

resp = opener.open(‘http://127.0.0.1:8000/auth/’)

print(resp.read().decode(‘utf-8’))

[urlopen() 함수-HTTPCookieProcessor 클래스로 쿠키 데이터를 포함하여 요청]

from urllib.request import Request, HTTPCookieProcessor, build\_opener

url = ‘http://127.0.0.1:8000/cookie/’

#first request (GET) with cookie handler

#쿠키 핸들러 생성, 쿠키 데이터 저장은 디폴트로 CookieJar 객체를 사용함

cookie\_handler = HTTPCookieProcessor()

opener = build\_opener(cookie\_handler)

req=Request(url)

res=opener.open(req)

print(res.info())

print(res.read().decode(‘utf-8’))

#second request(POST)

print(“-----------------------------------------“)

data = “language=python&framework=django”

encData = bytes(data, encoding=’utf-8’)

req=Request(url, encData)

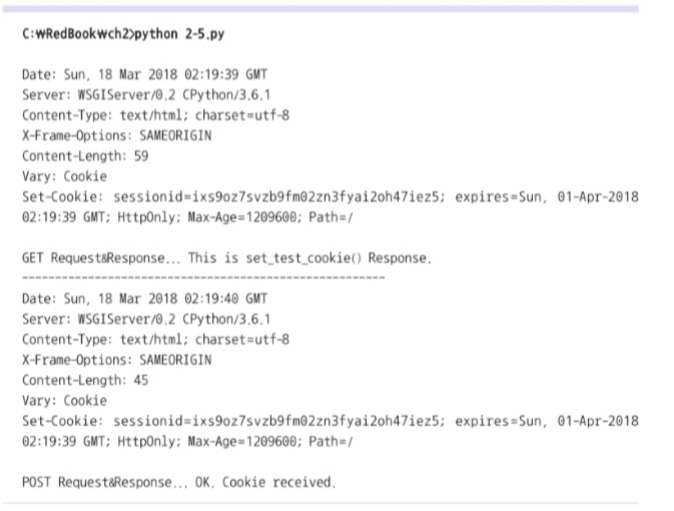
res=opener.open(req)

print(res.info())

print(res.read().decode(‘utf-8’))

->첫 번째 요청에서 쿠키를 담기 위한 준비를 하고 서버로 요청을 보냄.

두 번째 요청에서는 첫 번째 응답에서 받은 쿠키를 헤더에 담아서 요청을 보냄. 만일 두 번째 요청에 쿠키 데이터가 없다면 서버에서 에러로 응답

[urlopen() 함수-ProxyHandler 및 ProxyBasicAuthHandler 클래스로 프록시 처리]

import urllib.request

url = ‘http://www.example.com’

proxyServer = ‘http://www.proxy.com:3128/’

#프록시 서버를 통해 웹 서버로 요청을 보냅니다.

proxy\_handler = urllib.request.ProxyHandler({‘http’: proxyServer})

#프록시 서버 설정을 무시하고 웹 서버로 요청을 보냅니다.

# proxy\_handler = urllib.request.ProxyHandler({})

#프록시 서버에 대한 인증을 처리합니다.

proxy\_auth\_handler = urllib.request.ProxyBasicAuthHandler()

proxy\_auth\_handler.add\_password(‘realm’, ‘host’, ‘username’, ‘password’)

#2개의 핸들러를 오프너에 등록합니다.

opener = urllib.request.build\_opener(proxy\_handler, proxy\_auth\_handler)

#디폴트 오프너로 지정하면, urlopen() 함수로 요청을 보낼 수 있습니다.

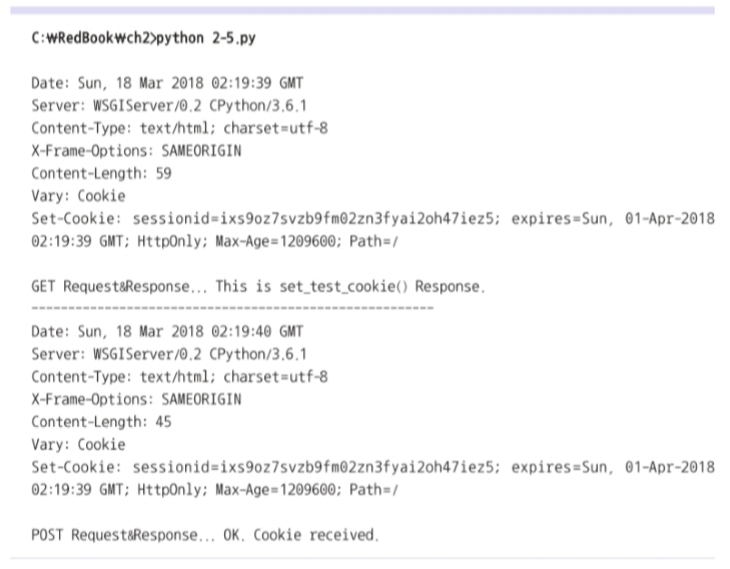
urllib.request.install\_opener(opener)

#opener.open() 대신에 urlopen() 사용

f = urllib.request.urlopen(url)

print(“geturl():”, f.geturl())

print(f.read(300).decode(‘utf-8’))



urlopen() 함수-ProxyHandler 및 ProxyBasicAuthHandler 클래스로 프록시 처리

import urllib.request

url = ‘http://www.example.com’

proxyServer = ‘http://www.proxy.com:3128/’

#프록시 서버를 통해 웹 서버로 요청을 보냅니다.

proxy\_handler = urllib.request.ProxyHandler({‘http’: proxyServer})

#프록시 서버 설정을 무시하고 웹 서버로 요청을 보냅니다.

#proxy\_handler = urllib.request.ProxyHandler({})

#프록시 서버에 대한 인증을 처리합니다.

proxy\_auth\_handler = urllib.request.ProxyBasicAuthHandler()

proxy\_auth\_handler.add\_password(‘realm’, ‘host’, ‘username’, ‘password’)

#2개의 핸들러를 오프너에 등록합니다.

opener = urllib.request.build\_opener(proxy\_handler, proxy\_auth\_handler)

#디폴트 오프너로 지정하면, urlopen() 함수로 요청을 보낼 수 있습니다.

urllib.request.install\_opener(opener)

#opener.open() 대신에 urlopen()을 사용했습니다.

f = urllib.request.urlopen(url)

print(“geturl():”, f.geturl())

print(f.read(300).decode(‘utf-8’))

특정 웹 사이트에서 이미지만을 검색하여 그 리스트를 보여주는 코드

(urlopen() 기능을 주로 사용, 추가적으로 HTMLParser 클래스 사용)

C:\Users\shkim>cd C:\RedBook\ch2

C:\RedBook\ch2>notepad parse\_image.py

from urllib.request import urlopen #필요한 함수 및 클래스 임포트

from html.parser import HTMLParser

class ImageParser(HTMLParser): #상속받는 클래스를 정의하고 필요한 내용 오버라이드

def handle\_starttag(self, tag, attrs): #<img> 태그 찾기 위해 오버라이드

if tag != ‘img’:

return

if not hasattr(self, ‘result’):

self.result = []

for name, value in attrs: #<img src> 속성을 찾으면

if name == ‘src’:

self.result.append(value) #속성값을 self.result 리스트에 추가

def parse\_image(data): #HTML 문장이 주어지면 HTMLParser 클래스를 사용해서 이미지를 찾고

그 리스트 출력

parser = ImageParser()

parser.feed(data) #HTML 문장을 feed() 함수에 주면 파싱하고 그 결과를 parser.result

리스트 에 추가

dataSet = set(x for x in parser.result) #파싱 결과를 set 타입의 dataSet으로 모아줌

return dataSet #dataSet으로 모은 파싱 결과를 리턴

def main(): #프로그램의 시작점, [www.google.co.kr](http://www.google.co.kr) 사이트를 검색하여 이미지 찾는 함수

url = <http://www.google.co.kr>

with urlopen(url) as f: #urlopen() 함수를 사용하여 구글 사이트에 접속한 후 첫 페이지

내용을 가져옴

charset = f.info().get\_param(‘charset’) #사이트에서 가져오는 데이터는 인코딩됨

#인코딩 방식(charset)을 알아내어

data = f.read().decode(charset) #그 방식으로 디코딩해줌

dataset = parse\_image(data) #이미지를 찾기 위해 parse\_image() 함수 호출

print(“\n>>>>>>>>> Fetch Images from”, url)

print(‘\n’.join(sorted(dataset))) #찾은 이미지들을 정리하여 라인별로 출력

if \_\_name\_\_ == ‘\_\_main\_\_’: #프로그램 시작 위해 main 호출

main()

위 프로그램을 실행하기 위해서 다음 명령 입력

C:\RedBook\ch2>python parse\_image.py



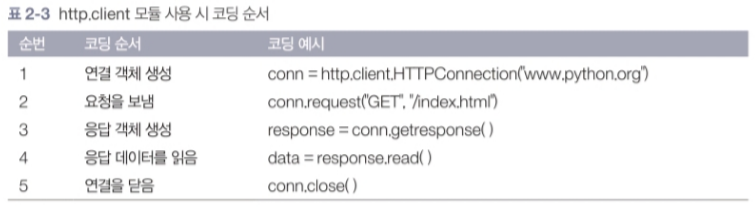
http.client 모듈

대부분의 웹 클라이언트 프로그램은 urllib.request 모듈에 정의된 기능만으로도 작성이 가능하나 GET, POST 이외의 방식으로 요청을 보내거나 요청 헤더와 바디 사이에 타이머를 두어 시간을 지연시키는 등 urllib.request 모듈로는 쉽게 처리할 수 없는 경우 혹은 HTTP 프로토콜 요청에 대한 저수준의 더 세밀한 기능이 필요할 때 http.client 모듈을 사용

urllib.request 모듈도 http.client 모듈에서 제공하는 API를 사용해서 만든 모듈이므로 urllib.request 모듈로 작성한 로직은 http.client 모듈을 사용해도 동일하게 작성 가능

http.client 모듈을 사용하여 웹 클라이언트를 작성할 때는 아래와 같은 순서를 기준으로 삼고 필요에 따라 변경하여 코딩하는 것을 권장

http.client 모듈 사용 시 코딩 순서



http.client 모듈을 사용하여 GET, HEAD, POST, PUT 방식으로 요청을 보내는 방법

1. http.client 모듈을 사용하여 GET 메소드로 요청을 보내는 예제

http.client 모듈 사용-GET 방식 요청

C:\RedBook\ch2>python

from http.client import HTTPConnection

host = ‘www.example.com’ #첫 번째 인자는 url이 아니라 host이므로 http://www.example.com(X)

conn = HTTPConnection(host)

conn.request(‘GET’, ‘/’) #GET 방식임을 명시적으로 표현 request(method, url, body, headers) 형식

r1 = conn.getresponse()

print(r1.status, r1.reason) #응답 결과를 출력 r1.msg 속성에는 응답 헤더 정보가 들어있음

data1 = r1.read() #데이터를 모두 읽어야 다음 request()를 요청할 수 있음

#일부만 읽는 경우는 두번째 요청 시 에러 남

#data1 = r1.read(100)

#두 번째 요청에 대한 테스트

conn.request(‘GET’, ‘/’)

r2 = conn.getresponse()

print(r2.status, r2.reason)

data2 = r2.read()

print(data2.decode())

conn.close()

http.client 모듈 사용-HEAD 방식 요청

from http.client import HTTPConnection

conn = HTTPConnection(‘www.example.com’) #첫 번째 인자는 url이 아니라 host

conn.request(‘HEAD’, ‘/’) #HEAD 방식임을 명시적으로 표현

resp = conn.getresponse()

print(resp.status, resp.reason) #응답 결과 출력

data = resp.read()

print(len(data)) #HEAD 요청에 대한 응답에 헤더는 있지만 바디가 없으므로 data 길이는 0

print(data == b’’)

http.client 모듈 사용-POST 방식 요청

from http.client import HTTPConnection

from urllib.parse import urlencode

host = ‘127.0.0.1:8000’ #첫 번째 인자는 host이므로 <http://127.0.0.1:8000>은 오류

params = urlencode({ #POST 요청으로 보낼 파라미터에 대해 URL 인코딩을 함

‘language’: ‘python’,

‘name’: ‘김석훈’,

‘email’: ‘shkim@naver.com’,

})

headers = { #POST 요청으로 보낼 헤더를 사전 타입으로 지정

‘Content-Type’: ‘application/x-www-form-urlencoded’,

‘Accept’: ‘text/plain’,

}

conn = HTTPConnection(host) #127.0.0.1 사이트의 8000번 포트에 접속 준비

conn.request(‘POST’, ‘’, params, headers) #POST 방식임을 명시적으로 표현

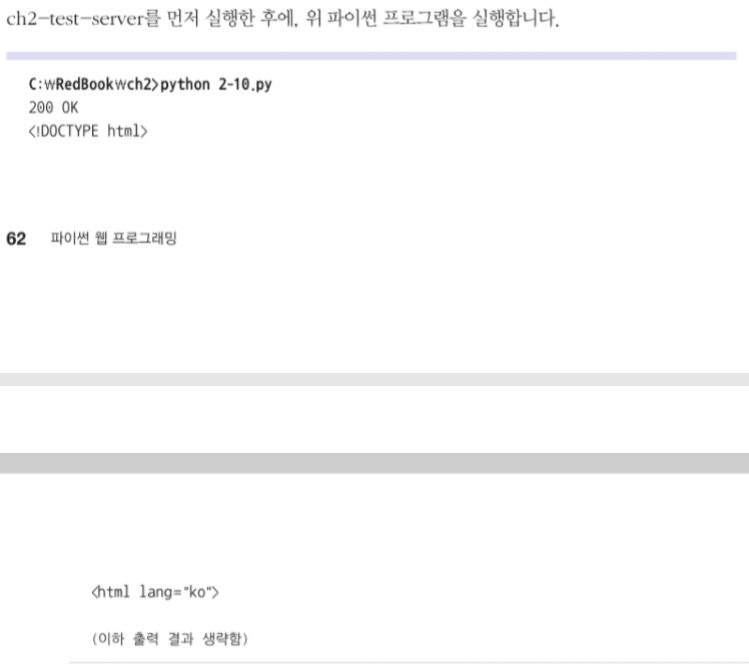
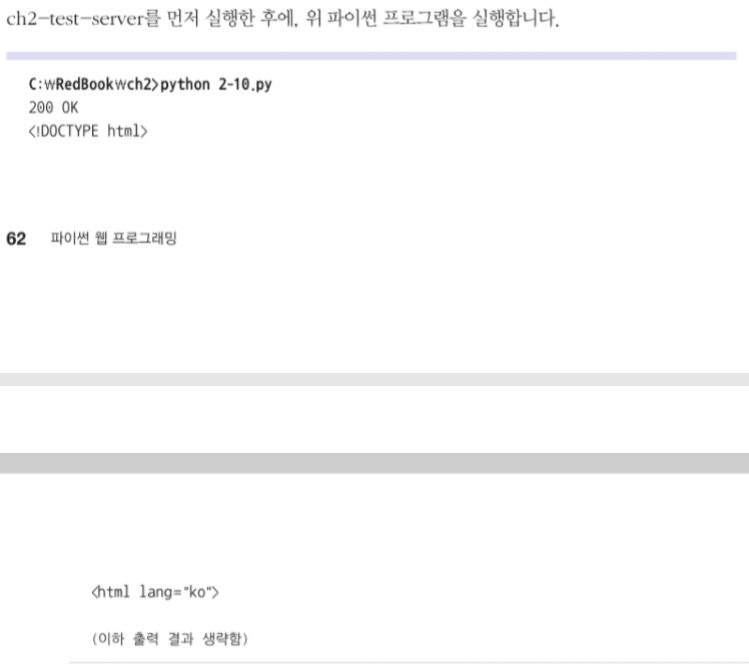
resp = conn.getresponse()

print(resp.status, resp.reason) #서버로부터의 응답 결과 확인. 200 OK가 출력되면 정상 처리

data = resp.read()

print(data.decode(‘utf-8’))

conn.close()



http.client 모듈 사용-PUT 방식 요청

from http.client import HTTPConnection

from urllib.parse import urlencode

host = ‘127.0.0.1:8000’

params = urlencode({

‘language’: ‘python’,

‘name’: ‘김석훈’

‘email’: ‘shkim@naver.com’

})

headers = {

‘Content-Type’: ‘application/x-www-form-urlencoded’,

‘Accept’: ‘text/plain’,

}

conn = HTTPConnection(host)

conn.request(‘PUT’, ‘’, params, headers)

resp = conn.getresponse()

print(resp.status, resp.reason)

data = resp.read(300)

print(data.decode(‘utf-8’))

conn.close()