# 파일에 데이터 읽고 쓰기

## 열라, 읽으라(쓰라), 그리고 닫으라

- 어플리케이션이 운영체제에게 파일 처리를 API 함수를 통해 의뢰하면, 운영체제가 요청한 업무를 수행해주고 그 결과를 어플리케이션에게 돌려줌.
- 어플리케이션이 파일 처리 업무를 의뢰하는 과정과 각 과정에서 사용되는 파이썬 함수

# 열라, 읽으라(쓰라), 그리고 닫으라

## ❖ 예제 : 11/write.py

- 열기-쓰기-닫기의 순서로 함수 호출 수행
- write.py를 실행하고 나면 test.txt 파일을 열어 내용 확인

```
file = open('test.txt', 'w')
file.write('hello')
file.close()
```

- 실행 결과

```
>write.py
```

## ❖ 예제 : 11/read.py

```
file = open('test.txt', 'r')
str = file.read()
print(str)
file.close()
```

- 실행 결과

```
>read.py
hello
```

# 열라, 읽으라(쓰라), 그리고 닫으라 – 자원 누수 방지를 돕는 with ~ as

- ❖ open() 함수와 함께 with ~ as문을 사용하면 명시적으로 close() 함수를 호출하지 않아도 파일이 항상 닫힘.
- ❖ with ~ as 문을 사용하는 방법은 다음과 같음.

```
with open(파일이름) as 파일객체:
    # 코드블록
    # 이곳에서 읽거나
    # 쓰기를 한 후
    # 그냥 코드을 빠져나가면 됩니다.
```

*"파일객체 = open(파일이름)"* 와 같다고 생각하면 됩니다.

with 문 덕분에 close()를 하지 않아도 됩니다.

- ❖ 예제 : 11/openwith.py

```
with open('test.txt', 'r') as file:
    str = file.read()
    print(str)
#file.close()
```

# 열라, 읽으라(쓰라), 그리고 닫으라 – with문의 비밀 : 컨텍스트 매니저

- **with문은 컨텍스트 매니저(Context Manager)를 제공하는 함수여야 사용 가능**
- **컨텍스트 매니저는 __enter__() 메소드와 __exit__() 메소드를 구현하고 있는 객체**
  - with문은 컨텍스트 매니저를 획득한 후 코드블록의 실행을 시작할 때 컨텍스트 매니저의 __enter__() 메소드를 호출하고, 코드 블록이 끝날 때 __exit__()를 호출

```
with open('test.txt', 'r') as file:
    s = file.read()
    print(s)
```

코드 블록 시작하기 전에 컨텍스트매니저.__enter__() 호출

컨텍스트매니저.__exit__() 호출

# 열라, 읽으라(쓰라), 그리고 닫으라 – with문의 비밀 : 컨텍스트 매니저

## ❖ 예제 : 11/open2.py(컨텍스트 매니저 구현)

```
class open2(object):
    def __init__(self, path):
        print ('initialized')
        self.file = open(path)

    def __enter__(self):
        print ('entered')
        return self.file

    def __exit__(self, ext, exv, trb):
        print ("exited")
        self.file.close()
        return True

with open2("test.txt") as file:
    s = file.read()
    print(s)
```

write.py 예제 프로그램을 통해 생성한 test.txt 파일이 존재해야 합니다.

- 실행 결과

```
>open2.py
initialized
entered
hello
exited
```

# 열라, 읽으라(쓰라), 그리고 닫으라 – with문의 비밀 : 컨텍스트 매니저

## ❖ @contextmanager 데코레이터

- `__call__()` 메소드는 물론이고, 컨텍스트 매니저 규약을 준수하는데 필요한 `__enter__()` 메소드와 `__exit__()` 메소드를 모두 갖추고 있음.

## ❖ 함수를 하나 만들고 이 데코레이터(@contextmanager)로 수식하면 컨텍스트 매니저의 구현이 마무리 됨.

```
from contextlib import contextmanager

@contextmanager
def 함수이름():
    # 자원 획득
    try:
        yield 자원
    finally:
        # 자원 해제
```

contextlib 모듈로부터 contextmanager를 반입

@contextmanager 데코레이터로 함수 수식

yield문을 통해 자원 반환 : with문의 코드블록이 시작될 때 실행됨.

with문의 코드블록이 종료될 때 실행됨.

# 열라, 읽으라(쓰라), 그리고 닫으라 – open() 함수 다시 보기

## ❖ open() 함수의 매개 변수는 모두 8개

- 하나의 필수 매개변수와 일곱 개의 매개변수
- 반환값은 물론 파일 객체.

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,
        newline=None, closefd=True, opener=None)
```

## ❖ file

- 파일의 경로를 나타내는 문자열 또는 이미 생성해놓은 파일 객체의 파일 기술자(`file` 객체의 `fileno`)

# 열라, 읽으라(쓰라), 그리고 닫으라 – open() 함수 다시 보기

## ❖ mode (파일 열기 모드)

| 문자 | 의미 |
|---|---|
| 'r' | 읽기용으로 열기 (기본값) |
| 'w' | 쓰기용으로 열기. 이미 같은 경로에 파일이 존재하면 파일내용을 비움. |
| 'x' | 배타적 생성모드로 열기. 파일이 존재하면 IOError 예외 일으킴. |
| 'a' | 쓰기용으로 열기. 단, 'w'와는 달리 이미 같은 경로에 파일이 존재하는 경우 기존 내용에 덧붙이기를 함. |
| 'b' | 바이너리 모드 |
| 't' | 텍스트 모드(기본값) |
| '+' | 읽기/쓰기용으로 파일 읽기 |

# 열라, 읽으라(쓰라), 그리고 닫으라 – open() 함수 다시 보기

- **buffering**
  - 0을 입력하면 파일 입출력시에 버퍼링을 수행하지 않으며(바이너리 모드에서만 사용 가능), 1을 입력하면 개행 문자(\n)를 만날 때까지 버퍼링을 하는 라인(line) 버퍼링을 수행(텍스트 모드에서만 사용 가능).
  - 임의의 값으로 직접 버퍼의 크기를 지정하고 싶을 때는 이 매개변수에 1보다 큰 수를 입력.
- **encoding**
  - 텍스트 모드에서만 사용. <문자 집합과 인코딩>에서 별도 설명

# 열라, 읽으라(쓰라), 그리고 닫으라 – open() 함수 다시 보기

## ❖ erros

ㅇ 텍스트 모드에서만 사용. 인코딩/디코딩 수행시의 에러 처리 옵션임.

| errors | 의미 |
|---|---|
| 'strict' | 인코딩 에러가 발생할 때 ValueError 예외를 일으킵니다. None과 똑같은 효과를 냅니다. |
| 'ignore' | 이름 그대로 에러를 무시합니다. |
| 'replace' | 기형적인 데이터가 있는 곳에 대체 기호(예를 들어 '?')를 삽입합니다. |
| 'surrogateescape' | U+DC80 ~ U+DCFF 사이에 있는 유니코드 사용자 자유 영역(PUA)의 잘못된 바이트를 코드 포인트(code points)로 나타냅니다. (코드 포인트는 문자를 나타내는 번호입니다. 나중에 한 번 더 설명하겠습니다.) |
| 'xmlcharrefreplace' | 파일에 기록하려는 텍스트 안에 지정된 인코딩(문자를 읽고 쓰는 방식을 정의한 규칙. 나중에 자세히 설명하겠습니다.)에서 지원되지 않는 문자를 &#NNN; 꼴의 XML 문자 참조로 바꿔서 기록하는 옵션입니다. 파일을 쓸 때만 적용됩니다. |
| 'backslashreplace' | 이 옵션도 역시 파일에 텍스트를 기록할 대만 사용됩니다. 현재 인코딩에서 지원되지 않는 문자를 역슬래쉬로 시작되는 이스케이프 시퀀스로 바꿔서 기록합니다. |

# 열라, 읽으라(쓰라), 그리고 닫으라 – open() 함수 다시 보기

❖ **newline**

- 파일을 읽고 쓸 때 줄바꿈을 어떻게 처리할지 결정
- None, '', '\n', '\r', '\r\n' 5가지 중 하나를 입력
- 파일을 읽을 때 :
  - newline이 None으로 설정되어 있으면 '\n', '\r', '\r\n'를 모두 개행문자로 간주하여 이들 개행문자를 '\n'으로 변환하여 읽기 메소드(read(), readline() 등)에게 반환
  - newline이 ''으로 설정되어 있는 경우에는 None으로 설정되어 있는 경우와 동일하게 동작하지만 개행 문자의 변환을 수행하지 않음. '\n'은 '\n'으로, '\r\n'은 '\r\n'으로 그대로 읽어들임.
  - 이 외에 newline에 '\n', '\r', '\r\n' 중 하나를 입력하면 입력한 문자만을 개행문자로 간주.
  - 예) newline을 '\n'으로 설정해 놓으면 '\r', '\r\n'은 개행문자로 취급하지 않음.
- 파일을 쓸 때 :
  - newline이 None이면 '\n', '\r', '\r\n' 등 어떤 개행문자를 파일에 쓰려고 해도 시스템 기본 개행 문자(os.linesep)로 변환되어 기록됨.
  - newline에 '' 또는 '\n'를 지정하는 경우엔 어떤 변환도 수행하지 않음.
  - newline에 '\r', '\r\n' 등이 지정되는 경우에는 모든개행 문자('\n', '\r', '\r\n')가 지정한 개행문자로 변환되어 기록됨.

# 열라, 읽으라(쓰라), 그리고 닫으라 – open() 함수 다시 보기

- **closefd**
  - 첫 번째 매개변수에 파일의 경로가 아닌 파일 기술자가 입력됐을 때만 사용.
  - file 매개변수에 파일 기술자를 입력하고 closed 매개변수에는 False를 입력하면 파일이 닫히더라도 파일 기술자를 계속 열어둔 상태로 유지.
- **opener**
  - 8번째 매개변수 opener는 파일을 여는 함수를 직접 구현하고 싶을 때 이용.
  - opener에 구현한 함수 또는 호출가능 객체(__call__() 메소드를 구현하는 객체)를 넘기면 됨.
  - 이 때 opener에 넘기는 함수/호출가능 객체는 반드시 파일 기술자를 반환해야 함.

# 텍스트 파일 읽기/쓰기 - 문자열을 담은 리스트를 파일에 쓰는 writelines() 메소드

## ❖ 예제 : 11/writelist.py(for와 write() 메소드를 이용)

```
lines = ["we'll find a way we always have - Interstellar\n",
         "I'll find you and I'll kill you - Taken\n",
         "I'll be back - Terminator 2\n"]

with open('movie_quotes.txt', 'w') as file:
    for line in lines:
        file.write(line)
```

- 실행결과

```
>writelist.py

>type movie_quotes.txt
we'll find a way we always have - Interstellar
I'll find you and I'll kill you - Taken
I'll be back - Terminator 2
```

type은 윈도우에서 제공하는 명령어입니다. 리눅스 또는 유닉스에서는 비슷한 기능의 명령어로 cat이 있습니다.

## 텍스트 파일 읽기/쓰기 - 문자열을 담은 리스트를 파일에 쓰는 writelines() 메소드

### ❖ 예제 : 11/writelines.py(writelines() 메소드를 이용)

```
lines = ["we'll find a way we always have - Interstellar\n",
         "I'll find you and I'll kill you - Taken\n",
         "I'll be back - Terminator 2\n"]

with open('movie_quotes.txt', 'w') as file:
    file.writelines(lines)
```

- 실행결과

```
>writelines.py

>type movie_quotes.txt
we'll find a way we always have - Interstellar
I'll find you and I'll kill you - Taken
I'll be back - Terminator 2
```

# 텍스트 파일 읽기/쓰기 - 줄 단위로 텍스트를 읽는 readline()과 readlines() 메소드

## ❖ 예제 : 11/readline.py (readline() 메소드 이용)

```
with open('movie_quotes.txt', 'r') as file:
    line = file.readline()

    while line != '':
        print(line, end='')
        line = file.readline()
```

readline() 메소드는 파일의 끝에 도달하면 ''를 반환합니다. 그런데 실제로 빈 줄을 읽어들였을 때는 어떡하냐고요? 빈 줄을 읽어 들인 경우엔 개행문자를 반환합니다.

- 실행결과

```
>readline.py
we'll find a way we always have - Interstellar
I'll find you and I'll kill you - Taken
I'll be back - Terminator 2
```

## 텍스트 파일 읽기/쓰기 - 줄 단위로 텍스트를 읽는 readline()과 readlines() 메소드

### ❖ 예제 : 11/readlines.py (readlines() 메소드 이용)

```
with open('movie_quotes.txt', 'r') as file:
    lines = file.readlines()
    line = ''
    for line in lines:
        print(line, end='')
```

- 실행결과

```
>readlines.py
we'll find a way we always have - Interstellar
I'll find you and I'll kill you - Taken
I'll be back - Terminator 2
```

# 텍스트 파일 읽기/쓰기 - 문자 집합과 인코딩에 대하여

- **"악보를 컴퓨터가 이해할 수 있는 기호로 표현할 수 있다면 컴퓨터가 음악 분야에서도 사용될 것" – Ada Lovelace**

- **문자 집합(Character Set) – 텍스트를 기호로 표현한 것.**

- **ASCII(American Standard Code for Information Interchange)**
  - 미국 정보 교환 표준 부호
  - 1960년대에 제정된 문자 집합으로, 이후에 개발된 문자 집합들의 토대가 됨.
  - ASCII는 7비트만을 이용하여 음이 아닌 수(0~127)에 문자 집합 내의 문자를 할당. 예) 61에는 '='를, 65에는 'A'를, 97에는 'a'를 할당
  - 52개의 알파벳 대소문자(A~Z, a~Z), 10개의 숫자(0~9), 32개의 특수 문자(!@#$?... 등등), 하나의 공백 문자, 그리고 33개의 출력 불가능한 제어문자, 모두 128개의 문자를 표현.

# 텍스트 파일 읽기/쓰기 - 문자 집합과 인코딩에 대하여

- **ISO/IEC 8859-1**
  - 128(27)개의 문자를 표현하는 ASCII가 7비트만 사용했던 것에 비해 8비트를 사용하여 256(28)개의 문자를 표현
- **ISO/IEC 8859-N**
  - 중앙 유럽어, 남유럽어, 북유럽어, 아랍어 등을 지원
- **DBCS(Double-Byte Character Set)**
  - 2바이트(16비트)를 활용해서 문자 집합을 구성하는 방법.
  - 최대 65,536(2^16)개의 문자를 할당할 수 있으며, 한글(총11,172자), 중국과 일본의 문자를 컴퓨터로 표현 가능.
  - DBCS는 ASCII와의 호환을 유지하기 위해 최상위 비트가 0이면 ASCII, 1이면 DBCS로 인식
  - DBCS에서는 하나의 문자열 안에서도 1바이트로 표현되고 있는 문자와 2바이트로 표현되는 문자를 혼용.
  - 한글 문자 집합 표준으로는 KS X 1001, EUC-KR, CP949 등이 있음.
  - EUC-KR은 한글 표준인 KS X 1001과 ASCII에서 '\'기호를 '₩'로 바꾼 영문자 표준 KS X 1003의 합집합.

# 텍스트 파일 읽기/쓰기 - 문자 집합과 인코딩에 대하여

- 유니코드(Unicode)
- 문자 집합 하나로 모든 문자를 표현할 수 있게 하는 것이 목적
- 초기에는 전세계의 언어별 문자들을 2바이트 안에서 영역을 나눠 할당
- 유니코드에서 문자에 부여되는 번호를 코드 포인트(Code Point)라고 함.
  - 코드포인트는 "U+" 뒤에 2바이트의 수를 16진수로 표현하여 붙여 표시
    예) 'A'의 코드 포인트는 U+0041, '한'의 코드 포인트는 U+D55C, '글'의 코드 포인트는 U+AE00
  - U+0000부터 U+007F까지를 ASCII와 동일하게 맞춰두고 그 뒷번호부터 각국의 문자를 할당.
- 누락된 현대 문자와 기호를 추가적으로 할당하고 고대 문자와 음악 기호 등을 추가하자 코드포인트가 2바이트를 넘어서게 됨.

## ❖ UTF(Unicode Transformation Format)

- 유니코드 변환 인코딩 형식
- UTF-8은 코드포인트의 크기에 따라 1바이트에서부터 4바이트까지 가변폭으로 인코딩하므로 1 바이트로 표현 가능한 U+0000(십진수 0)부터 U+007F(십진수 127)까지는 ASCII와 완벽하게 호환
  - UTF-8 인코딩 방식으로 저장된 문서는 유니코드를 알지 못하는 시스템에서도 사용 가능.
- UTF-8 외에도 UTF-7, UTF-16, UTF-32 인코딩 등이 있음.

# 텍스트 파일 읽기/쓰기 - 문자 집합과 인코딩에 대하여

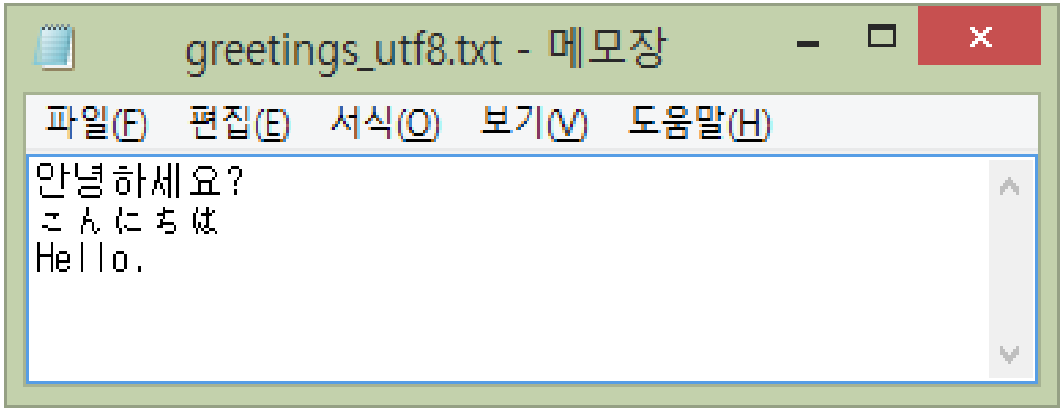
## ❖ 예제 : 11/utf8_write.py

```
lines = ['안녕하세요?\n',
         'こんにちは\n',
         'Hello.\n']

with open('greetings_utf8.txt', 'w', encoding='utf-8') as file:
    for line in lines:
        file.write(line)
```

- 실행결과

```
>utf8_write.py
```



# 텍스트 파일 읽기/쓰기 - 문자 집합과 인코딩에 대하여

## ❖ 예제 : 11/utf_read.py (UTF-8로 텍스트 파일 읽기)

```
with open('greetings_utf8.txt', 'r', encoding='utf-8') as file:
    lines = file.readlines()
    line = ''
    for line in lines:
        print(line, end='')
```

- 실행결과

```
>utf8_read.py
안녕하세요?
こんにちは
Hello
```

# 텍스트 파일 읽기/쓰기 - 문자 집합과 인코딩에 대하여

## ❖ 예제 : 11/ascii_read.py (ASCII로 텍스트 파일 읽기)

```
with open('greetings_utf8.txt', 'r', encoding='ascii',
errors='ignore') as file:
    lines = file.readlines()
    line = ''
    for line in lines:
        print(line, end='')
```

errors 옵션에 'ignore'를 지정해서 해당 인코딩으로 처리가 안되는 문자열이 있을 때는 지나치도록 했습니다.

- 실행결과

```
>ascii_read.py
?

Hello.
```

errors 옵션에 'ignore'를 지정해서 해당 인코딩으로 처리가 안되는 문자열이 있을 때는 무시하도록 했습니다. 그 결과 ASCII와 호환 가능한 'Hello.' 말고는 아무 것도 출력되지 않습니다.

# 바이너리 파일 다루기

## ❖ struct 모듈

- 일반 데이터 형식과 bytes 형식 사이의 변환을 수행하는 함수 정의.

## ❖ 실습 1 (struct 모듈 사용 예)

```
>>> import struct
>>> packed = struct.pack('i', 123)
>>> for b in packed:
    print(b)

123
0
0
0
>>> unpacked = struct.unpack('i', packed)
>>> unpacked
(123,)
>>> type(unpacked)
<class 'tuple'>
```

pack() 함수는 첫 번째 매개변수 'i' 따라 4바이트 크기의 bytes 객체 packed를 준비하고 두 번째 매개변수를 bytes에 복사해 넣습니다.

bytes 객체 packed의 각 바이트에 있는 내용을 출력합니다.

unpack() 함수는 튜플 형식을 반환합니다.

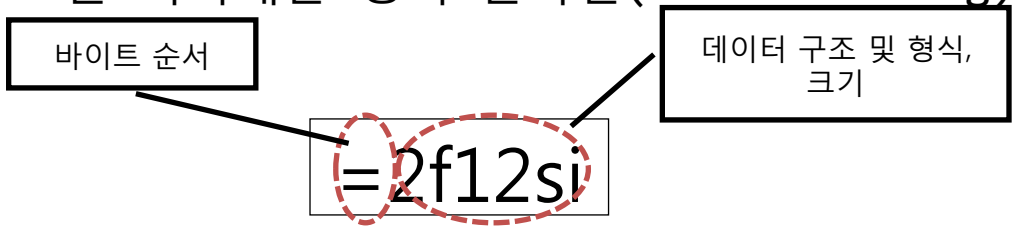
# 바이너리 파일 다루기

## ❖ pack() 함수와 unpack() 함수

```
struct.pack(fmt, v1, v2, ...) # 데이터 ➔ bytes
struct.unpack(fmt, buffer) # bytes ➔ 튜플(데이터)
```

## ❖ pack() 함수와 unpack() 함수의 첫 번째 매개 변수,fmt

- 데이터의 구조를 나타내는 형식 문자열(Format String)




## ❖ fmt에 바이트 순서를 지정하기 위해 사용할 수 있는 문자

| 문자 | 바이트 순서 | 크기 | 바이트 정렬 |
|---|---|---|---|
| @ | 시스템 바이트 순서를 따릅니다. | 시스템을 따릅니다. | 시스템을 따릅니다. |
| = | 시스템 바이트 순서를 따릅니다 | 표준을 따릅니다. | 수행 안 함. |
| < | 리틀 엔디안 | 표준을 따릅니다. | 수행 안 함. |
| > | 빅 엔디안 | 표준을 따릅니다. | 수행 안 함. |
| ! | 네트워크 바이트 순서<br>(빅 엔디안과 동일) | 표준을 따릅니다. | 수행 안 함. |

# 바이너리 파일 다루기

## ❖ 데이터 구조를 정의하는 형식 문자

| 분류 | 문자 | 파이썬 자료형 | C 자료형 | 크기 |
|---|---|---|---|---|
| 정수형 | b | integer | signed char | 1 |
| | B | integer | unsigned char | 1 |
| | ? | bool | _Bool | 1 |
| | h | integer | short | 2 |
| | H | integer | unsigned short | 2 |
| | i | integer | int | 4 |
| | I | integer | unsigned int | 4 |
| | l | integer | long | 4 |
| | L | integer | unsigned long | 4 |
| | q | integer | long long | 8 |
| | Q | integer | unsigned long long | 8 |
| | n | integer | ssize_t | |
| | N | integer | size_t | |
| 부동 소수형 | f | float | float | 4 |
| | d | float | double | 8 |
| bytes | s | bytes | char[] | |
| | p | bytes | char[] | |
| | c | 크기가 1인 bytes형 | char | 1 |
| 기타 | x | no value | 패딩 바이트( 데이터 저장 용도가 아닌 바이트 배열을 맞추기 위한 자리 맞추기 용도) | |
| | P | integer | void * | |

# 바이너리 파일 다루기

## ❖ 실습 1 (부동 소수형 pack/unpack)

```
>>> import struct
>>> packed = struct.pack('f', 123.456)
>>> unpacked = struct.unpack('f', packed)
>>> unpacked
(123.45600128173828,)
```

## ❖ 실습 2 (문자열 pack/unpack)

```
>>> import struct
>>> packed = struct.pack('12s', '대한민국'.encode())
>>> unpacked = struct.unpack('12s', packed)
>>> unpacked[0].decode()
'대한민국'
```

## ❖ 실습 3 (구조체 pack/unpack)

```
>>> import struct
>>> packed = struct.pack('2d2i', *(123.456, 987.765, 123, 456))
>>> unpacked = struct.unpack('2d2i', packed)
>>> unpacked
(123.456, 987.765, 123, 456)
```

# 바이너리 파일 다루기

## ❖ 예제 : 11/binary_write.py

```
import struct

struct_fmt = '=16s2fi' # char[16], float[2], int
city_info = [
    # CITY,          Latitude,   Longitude,  Population
    ('서울'.encode(encoding='utf-8'), 37.566535, 126.977969, 9820000),
    ('뉴욕'.encode(encoding='utf-8'), 40.712784, -74.005941, 8400000),
    ('파리'.encode(encoding='utf-8'), 48.856614, 2.352222,   2210000),
    ('런던'.encode(encoding='utf-8'), 51.507351, -0.127758,  8300000)
    ]

with open('cities.dat', 'wb') as file:
    for city in city_info:
        file.write(struct.pack(struct_fmt, *city))
```

- 실행결과

```
>binary_write.py
```

# 바이너리 파일 다루기

## ❖ 예제 : 11/binary_read.py

```
import struct

struct_fmt = '=16s2fi' # char[16], float[2], int
struct_len = struct.calcsize(struct_fmt)

cities = []
with open('cities.dat', "rb") as file:
    while True:
        buffer = file.read(struct_len)
        if not buffer: break
        city = struct.unpack(struct_fmt, buffer)
        cities.append(city)

    for city in cities:
        name = city[0].decode(encoding='utf-8').replace('\x00', '')
        print('City:{0}, Lat/Long:{1}/{2}, Population:{3}'.format(
              name,
              city[1],
              city[2],
              city[3]))
```

pack() 하는 과정에서 문자를 할당하고 남은 공간에 채워진 ₩x00를 디코딩한 후 빈 문자열로 다시 바꿔 넣습니다

# 바이너리 파일 다루기

- 실행결과

```
>binary_read.py
City:서울, Lat/Long:37.56653594970703/126.97796630859375,
Population:9820000
City:뉴욕, Lat/Long:40.71278381347656/-74.00594329833984,
Population:8400000
City:파리, Lat/Long:48.85661315917969/2.352221965789795,
Population:2210000
City:런던, Lat/Long:51.50735092163086/-0.1277579963207245,
Population:8300000
```