CHAPTER 11

파일 처리

File Handling

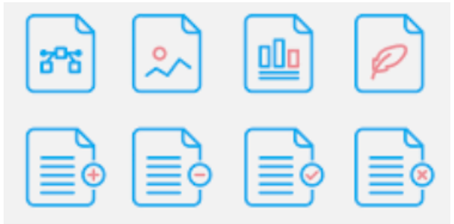
박 진 수 교수
서울대학교•경영대학
jinsoo@snu.ac.kr

학습 목차

- ☑ 파일 종류
- ☑ 파일 쓰기와 읽기

파일 종류

File Type



파일 처리

- 파일
  - 컴퓨터 프로그램이 접근해서 사용할 수 있는 정보들의 모음을 담고 있는 컨테이너
  - 파일은 주로 내구성이 뛰어난 저장소에 저장되기 때문에 파일에 저장한 데이터는 지속성을 가진다
  - 따라서 여러 프로그램이 안정적으로 사용할 수 있다

- 지속성(persistence)
  - 프로그램 실행을 여러번 하는 동안 데이터를 유지하기 위해서는 데이터를 저장해야 한다
  - 데이터는 파일 형식으로 저장되는데 일반적으로 컴퓨터 디스크에 저장
  - 저장한 데이터는 언제든지 가져와서 사용할 수 있다

- 파일 처리
  - 파일 또는 파일에 저장된 데이터를 조작하는 것

파일 종류

- 텍스트 형식(text format)
  - 사람이 읽거나 편집할 수 있으며 툴(tool)이나 텍스트 편집기를 사용하여 간편하게 처리하거나 변경하는 것이 용이
  - 쓰기는 쉽지만 읽기(예, 숫자 및 날짜)는 까다롭다
    - 따라서 파싱(parsing)이 필요

- 마크업 형식(markup format)
  - 사람이 읽거나 편집할 수 있으며, 툴이나 텍스트 편집기를 사용하여 간편하게 처리하거나 변경 가능
  - 태그를 사용하게 때문에 내용이 장황하고 파일의 크기가 텍스트 형식보다는 큰 편
  - 텍스트 데이터 또는 이진 형식 데이터를 불러오거나 내보낼 때 징검다리 역할로 많이 사용

- 이진 형식(binary format)
  - 이진 형식 데이터는 크기가 상대적으로 작기 때문에 빠른 저장과 불러오기가 가능
  - 사람이 읽거나 편집할 수 없기 때문에 파싱(parsing)이 불필요
  - 이진 형식의 정확한 포맷 규칙을 모르면 이진 데이터를 처리할 수 없다

파일 입력 및 출력

- 프로그램이 파일을 사용하기 위한 필수 3 단계
  - 파일 열기
  - 파일 처리
  - 파일 닫기

```
my_file = open('고향의봄.txt', mode='r')
line = my_file.readline()
print(line)
```

```
나의 살던 고향은 꽃피는 산골
```

- 파일 객체(file object)
  - 특정 파일과 연결된 객체
  - 프로그램이 파일 처리와 관련한 모든 작업을 할 수 있는 방법을 제공
  - 파일 객체는 일반적으로 변수를 통해 참조
    - 즉, 불러온 파일을 특정한 이름(변수 이름)으로 부른다는 뜻

파일 쓰기와 읽기

Writing & Reading Files

파일 쓰기와 읽기

작성 방법

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

- 해당 파일(*file*)과 열결된 파일 객체를 반환
  - 만약 파일을 열 수 없으면 OSError가 발생
- *file*은 열려는 파일의 **경로 이름**(현재 작업 폴더의 절대경로 또는 상대경로)과 **파일 이름**을 합한 문자열
- *mode*는 선택 사항으로 열려는 파일의 모드를 지정
- *mode*의 종류는 다음과 같다
  - 기본적으로 읽기와 쓰기 모드('r', 'w', 'a')
  - 'r' : **읽기**용으로 파일을 연다(기본값이 텍스트 읽기 모드이기 때문에 설정하지 않으면 'rt' 모드로 파일을 연다)
  - 'w' : **쓰기**용으로 파일을 연다(만약 파일이 이미 존재하면 **기존의 데이터를 삭제**하고 새로 작성하게 되므로 매우 조심해서 사용)
  - 'x' : 'w'와 같지만 파일이 이미 존재하면 오류가 발생
  - 'a' : 'w'와 같지만 파일이 이미 존재하면 파일의 마지막에 내용을 추가
  - 't' : 텍스트 모드로 파일을 연다
    - ✱ 기본값이기 때문에 텍스트 모드로 파일을 열 때는 't'를 생략해도 된다
    - ✱ 텍스트 모드에서 *encoding*을 지정하지 않으면 파이썬 개발 환경을 설치한 운영체제의 기본 인코딩이 자동 적용
  - 'b' : 이진(binary) 모드로 파일을 연다
    - ✱ 이진 모드로 파일을 열 때는 *encoding*을 지정하지 않는다
    - ✱ 'w+b' 모드로 파일을 열 때 파일이 존재하면 기존의 데이터를 삭제하고 새로 작성한다
    - ✱ 'r+b'모드로 파일을 열 때는 기존의 데이터를 삭제하지 않고 연다
  - '+' : 업데이트(읽고 쓰기)용으로 파일을 연다
- *encoding*은 파이썬이 지원하는 텍스트 인코딩의 이름을 정할 수 있다
  - **텍스트 모드**에서만 사용할 수 있다
  - *encoding*을 지정하지 않으면 파이썬 개발 환경을 설치한 운영체제의 기본 인코딩이 자동 적용
  - *encoding*을 지정하면 파일을 열 때 지정한 인코딩으로 파일을 인코딩하거나 디코딩한다
- 나머지 매개변수 대부분은 기본값을 사용하기 때문에 설명을 생략
  - 구체적인 내용을 알고 싶으면 파이썬 문서를 참고

참고 : 파일 인코딩

- https://en.wikipedia.org/wiki/Code_page
- 디폴트 인코딩(default encoding)
  - Windows : ANSI(예, `cp949`, `cp1252` 등과 같은 윈도우 코드 페이지)
    - '`latin-1`'은 윈도우 코드 페이지와 대부분 호환 가능한 보편적인 인코딩 방식
    - '`cp949`'는 한글을 지원하는 인코딩 방식
    - 더 많은 정보를 위해서는 다음 URL을 참고
      - https://en.wikipedia.org/wiki/Windows_code_page
  - macOS : Unicode
    - '`utf-8`'은 영어 외의 언어를 지원
    - '`cp1252`' ('`latin-1`')는 라틴 알파벳(예, 영어)을 지원
- 국제 표준은 Unicode
  - 최근에는 Unicode를 주된 인코딩 방식으로 사용하는 추세
  - 한글 인코딩을 위해서는 '`utf-8`' 사용을 권장
  - 참고
    - 영어로 쓰인 파일은 대부분 윈도우와 맥 OS 환경에서 잘 작동
    - 한국어 단독이나 한국어와 영어가 함께 쓰인 파일은 윈도우나 맥 OS와 무관하게 utf-8 인코딩 방식을 사용하는 것이 바람직하다

예시 : 파일 인코딩

```
# 윈도우, 맥 OS, 리눅스에서 만든 영어 파일을 읽을 경우 아래 코드를 사용
file = open('input.txt') # 대부분의 경우 작동

# 윈도우에서 만든 한글 파일을 윈도우에서 읽을 경우 아래 코드를 사용
file = open('input.txt') # 대부분의 경우 작동

# 맥 OS, 리눅스에서 만든 한글 파일을 맥 OS, 리눅스에서 읽을 경우 아래 코드를 사용
file = open('input.txt') # 대부분의 경우 작동

# 맥 OS, 리눅스에서 만든 한글 파일을 윈도우에서 읽을 경우 아래 코드를 사용
file = open('input.txt', encoding='utf-8') # 대부분의 경우 작동

# 윈도우에서 만든 한글 파일을 맥 OS, 리눅스에서 읽을 경우 아래 코드를 사용
file = open('input.txt', encoding='cp949')  # 대부분의 경우 작동
file = open('input.txt', encoding='euc-kr') # 'cp949'가 안 되면 이 인코딩을 시도

# 운영체제 기본 값으로 파일을 쓸 경우는 아래 코드를 사용
file = open('output.txt', mode='w') # 또는 간단히 open('output.txt', 'w')

# 모든 언어에서 인코딩 방식으로 UTF-8을 사용하도록 권장하고 있기 때문에 파일 쓰기를 할 때
# 아래 코드처럼 사용할 것을 권장
file = open('output.txt', mode='w', encoding='utf-8')
```

쓰기 모드로 텍스트 파일 열기

아래 내용 그대로 따라하기

```
file = open('고향의봄.txt', mode='w', encoding='utf-8')
print(file)  # 파일 객체의 내용을 알아본다
```

```
<_io.TextIOWrapper name='고향의봄.txt' mode='w' encoding='utf-8'>
```

파일 객체 속성

파일객체.name

파일명을 담고있다

파일객체.mode

파일을 어떤 접근 모드로 열었는지 접근 모드 값을 담고있다

파일객체.encoding

파일의 인코딩 방식을 담고있다

파일객체.closed

파일 스트림이 닫혀 있으면 참(True)을 반환하고
아직 열려있으면 거짓(False)을 반환

```
# 파일 이름을 알려준다.
file.name
```

```
'고향의봄.txt'
```

```
# 파일이 무슨 모드로 열렸는지 알려준다.
file.mode
```

```
'w'
```

```
# 파일의 인코딩 방식을 알려준다.
file.encoding
```

```
'utf-8'
```

```
# 파일이 닫혔으면 참(True)을 반환한다.
file.closed
```

```
False
```

```
# 불러온 파일 객체를 닫는다.
file.close()
```

```
# 파일이 닫혔는지 확인한다.
file.closed
```

```
True
```

파일 닫기

```
파일객체.close()
```

- **파일객체**를 **쓰기 모드**로 열었을 때는
  - 모든 쓰기 작업을 완료한 후에 이 메소드를 **반드시 호출**해야 한다
  - 작업을 완료한 후에 해당 파일의 제어 권한을 다른 프로그램에 양도하는 역할을 하기 때문
  - 만약 작업이 끝난 뒤에 불러온 파일 객체를 닫아주지 않으면 파일 쓰기 작업을 제대로 마무리하지 못하고 종료하는 경우가 가끔 발생한다
  - 따라서 파일 쓰기 작업을 종료할 때 반드시 **파일객체**를 닫아주는 것이 좋다
- **파일객체**를 **읽기 모드**로 열었을 때는
  - 읽기 작업을 완료한 후에 이 메소드를 반드시 호출할 필요는 없다
  - 하지만 만약 같은 프로그램을 실행하는 동안 같은 **파일객체**를 다시 읽으려면 파일을 닫았다가 다시 여는 것이 좋다

텍스트 파일 쓰기

```
파일객체.write(객체)
```

- 전달인자에 명시한 특정 **객체**를 **파일객체**에 쓴 후 쓰여진 문자의 개수를 반환
- 텍스트 모드에서 **객체**는 **문자열**이다
  - 텍스트 파일을 쓸 때는 **문자열**에서 주어진대로 쓰기 때문에 **문자열** 끝에 새줄바꿈('\n')을 자동으로 추가하지 않는다
- **이진 모드**에서 **객체**는 바이트 형식의 텍스트나 객체

파일 쓰기 예시 : 고향의 봄

아래 내용 그대로 따라하기

```
file = open('고향의봄.txt', mode='w', encoding='utf-8')
file.write('나의 살던 고향은 꽃피는 산골')  # 파일에 문자열을 쓴다.
file.close()
```

```
16
```

```
file = open('고향의봄.txt', encoding='utf-8')
file.read()
```

```
'나의 살던 고향은 꽃피는 산골'
```

파일 내용 추가하기 예시 : 고향의 봄

아래 내용 그대로 따라하기

```
# mode='a'는 파일이 이미 존재하는 경우 파일의 마지막에 내용을 추가
file = open('고향의봄.txt', mode='a', encoding='utf-8')
```

```
file.write('\n')  # 앞서 작성한 가사 첫 줄의 마지막에 새줄바꿈을 추가
```

```
1
```

```
file.write('복숭아꽃 살구꽃 아기 진달래\n')
```

```
16
```

```
file.write('울긋불긋 꽃대궐 차리인 동네\n')
```

```
16
```

```
file.write('그 속에서 놀던 때가 그립습니다\n')
```

```
18
```

```
# 2절 가사 쓰기 전에 새줄바꿈
file.write('\n꽃동네 새동네 나의 옛고향\n')
```

```
16
```

```
# 한 번에 가사를 두 줄 쓴다
file.write('파란 들 남쪽에서 바람이 불면\n냇가에 수양버들 춤추는 동네\n')
```

```
33
```

```
# 가사의 마지막 줄에는 새줄바꿈 의도적으로 추가하지 않는다
file.write('그 속에서 놀던 때가 그립습니다')
```

```
17
```

```
file.close()  # 파일에 내용을 다 쓴 후에 파일을 닫는다
```

파일 읽기 : 고향의 봄

```
file = open('고향의봄.txt', mode='r', encoding='utf-8')
print(file.read())  # 텍스트 형식으로 출력
```

```
# mode='r'는 기본값이라 생략해도 된다
file = open('고향의봄.txt', encoding='utf-8')
file.read()  # 대표 형식으로 출력
```

리스트의 객체를 파일로 쓰기 : 새줄바꿈

새줄바꿈으로 각 객체를 분리

```
# 리스트의 각 객체를 새줄바꿈으로 분리해서 한 줄에 하나씩 파일로 쓴 후 저장
L = ['to', 'be', 'or', 'not', 'to', 'be']

file = open('list.txt', mode='w', encoding='utf-8')

for word in L:
    file.write(word + '\n')  # 각 객체 뒤에 새줄바꿈을 추가해서 쓴다
else:
    file.close()
```

```
to
be
or
not
to
be
```

리스트의 객체를 파일로 쓰기 : 추가 모드

역순으로 쉼표와 공백으로 분리해서 파일에 추가

```
# 리스트의 각 객체를 쉼표와 공백으로 분리해서 기존 파일에 추가한 후 저장
L = ['to', 'be', 'or', 'not', 'to', 'be']

file = open('list.txt', mode='a', encoding='utf-8')

# 각 객체를 역순으로 쉼표와 공백으로 분리해서 파일에 추가
file.write(', '.join(reversed(L)))
file.close()
```

```
to
be
or
not
to
be
be, to, not, or, be, to
```

숫자형 데이터 읽기와 쓰기

- 숫자는 반드시 문자열로 변환된 후 파일에 쓰여져야 한다
  - `str` 클래스 생성자를 사용하면 값을 문자열로 변환하는 것이 가능
- 텍스트 파일에서 숫자는 문자열로 읽는다
  - 따라서 수학적 연산을 수행하기 위해서는 반드시 숫자형으로 변환해야 한다
  - `int` 또는 `float` 클래스 생성자를 사용하면 문자열을 수치로 변환하는 것이 가능

숫자를 파일로 쓰기

정수나 실수를 파일로 쓰기

```
# 튜플의 각 숫자(정수 또는 실수)를 새줄바꿈으로 분리해서 한 줄에 하나씩 파일로 쓴 후 저장
numbers = 2, 3.1, -7, -1.4, 5

file = open('numbers.txt', mode='w', encoding='utf-8')

for number in numbers:
    file.write(str(number) + '\n')  # 문자열로 형변환해서 파일로 쓴다
else:
    file.close()
```

```
2
3.1
-7
-1.4
5
```

정리 : 텍스트 파일 쓰기

- open 함수

```
open(파일이름, mode='w', encoding='utf-8')
```

  - open 함수를 사용해서 텍스트 파일(.txt)을 열 수 있다
    - 파일을 저장할 때는 mode를 'w' 또는 'a'로 설정
    - 파일 쓰기는 두 모드가 있다
      - 쓰기 모드 : 'w'
      - 추가 모드 : 'a'
    - 일반적으로 'utf-8' 인코딩을 사용한다

- 파일 쓰기 메소드
  - 파일객체.write()
    - 전달인자에 명시한 특정한 문자열을 파일에 쓴 후 쓰여진 문자/바이트의 수를 반환
    - 줄 바꿈을 하려면 문자열 안에 반드시 새줄바꿈('\n')을 명시해야 한다
  - 파일객체.close()
    - 모든 쓰기 작업이 완료되었음을 확인한 후 해당 파일에 대한 제어 권한을 양도한다는 의미가 있기 때문에 파일 작업의 마무리 때 꼭 필요한 메소드
    - 불러온 파일 객체를 작업이 끝난 뒤 닫아주지 않아 실제 그 프로그램이 파일 생성 작업을 마무리 못하고 종료하는 경우가 종종 발생하기 때문에 작업 종료 전 파일 객체를 반드시 닫아주는 것이 좋다

읽기 모드로 파일 열기

'고향의봄.txt' 파일 열기

```
# 읽기 모드로 파일을 연다
file = open('고향의봄.txt', mode='r', encoding='utf-8')
```

```
file            # 파일 객체의 내용을 대표 형식으로 출력
```

```
print(file)  # 파일 객체의 내용을 텍스트 형식으로 출력
```

파일 읽기 : read 메소드

```
파일객체.read(size=-1)
```

- 텍스트 파일의 **모든 내용을 하나의 문자열**로 반환
- 특징은 다음과 같다
  - 텍스트 파일인 **파일객체**에서 커서의 현재 위치부터 최대 *size* 크기의 문자를 읽은 후, 새줄바꿈(\n)을 포함해서 읽어 들인 문자 모두를 문자열 하나로 반환
  - *size*의 값이 음수거나 지정하지 않으면 파일 끝(end-of-file, EOF)까지 읽어 들인다
  - 파일을 읽을 때마다 *size* 만큼 커서가 이동
  - 커서가 파일의 끝(EOF)에 있으면 빈 문자열('')을 반환

예시 : read 메소드

```
file = open('고향의봄.txt', mode='r', encoding='utf-8')
data = file.read()      # 파일 내용을 읽어 변수에 할당
```

```
type(data)              # read()가 반환한 객체의 자료형을 확인
```

```
data                    # 대표 형식으로 출력
```

```
print(data)             # 텍스트 형식으로 출력
```

예시 : read 메소드

파일 전체를 읽고 난 후 파일의 끝에서 파일을 다시 읽어보자

```
file = open('고향의봄.txt', encoding='utf=8')
```

```
file.read()
```

```
file.read()
```

파일의 일부만 읽어보자

```
file = open('고향의봄.txt', encoding='utf=8')
```

```
print(file.read(9))
```

```
print(file.read())
```

```
file.read()
```

파일 읽기 : readline 메소드

```
파일객체.readline(size=-1)
```

- 텍스트 파일에서 **한 줄의 내용을 하나의 문자열**로 반환
- 특징은 다음과 같다
  - 텍스트 파일인 **파일객체**에서 커서의 현재 위치부터 새줄바꿈('\n')까지 또는 파일의 끝(EOF)까지 읽은 후, 새줄바꿈('\n') 부호를 포함한 텍스트 한 줄을 문자열로 반환
  - 즉, 파일 한 줄의 내용을 문자열 하나로 반환하는 메소드
  - *size*의 값이 주어졌을 때는 최대 *size* 크기의 문자만 읽는다
  - 커서가 파일의 끝(EOF)에 있으면 빈 문자열('')을 반환

예시 : readline 메소드

```
data = open('고향의봄.txt', mode='r', encoding='utf-8').readline()
```

```
type(data)                  # readline 메소드가 반환한 객체의 자료형을 확인
```

```
data                        # 대표 형식으로 출력
```

```
print(data)                 # 텍스트 형식으로 출력
```

예시 : readline 메소드

파일을 한 줄씩 읽어보자

```
file = open('고향의봄.txt', encoding='utf=8')
```

```
file.readline()
```

```
file.readline()
```

```
print(file.readline())
```

파일의 일부만 읽어보자

```
print(file.readline(5))
```

```
for line in file:  # 커서의 현재 위치부터 나머지를 모두 읽어 처리
    print(line, end='')  # print(line.rstrip())와 비슷한 효과
```

```
file.readline()
```

while 문으로 파일 전체 읽기

```
file = open('고향의봄.txt', mode='r', encoding='utf=8')
while True:
    line = file.readline()  # EOF이면 빈 문자열('')을 반환
    if not line:
        break
    print(line.rstrip())
```

```
file.readline()
```

for 문으로 파일 전체 읽기

```
file = open('고향의봄.txt', encoding='utf=8')
for line in file:  # readline()을 호출할 필요가 없다
    print(line.rstrip())  # print(line, end='')와 같다
```

```
file.readline()
```

```
file.closed
```

```
file.close()
file.closed
```

with 문으로 파일 전체 읽기

```
with open('고향의봄.txt', encoding='utf=8') as file:
    for line in file:
        print(line.rstrip())
```

```
file.readline()
```

```
file.closed
```

파일 읽기 : readlines 메소드

```
파일객체.readlines(hint=-1)
```

- 텍스트 파일의 내용을 **한 줄 단위로 모두 읽어**서 **리스트**로 반환
- 특징은 다음과 같다
  - **파일객체**(텍스트 파일)에서 커서의 현재 위치부터 각 줄을 모두 읽은 후, 새줄 바꿈('\n') 부호를 포함한 각 줄을 객체로 갖는 리스트를 반환
  - 즉, 리스트의 개별 객체는 새줄바꿈 부호까지 포함한 텍스트 한 줄의 문자열
  - *hint*는 읽어 들인 줄의 개수를 제어할 때 사용할 수 있다
    - 만약 지금까지 읽어 들인 모든 줄의 총 크기(문자)가 *hint*를 초과하면 더 이상 줄을 읽지 않는다
  - 커서가 파일의 끝(EOF)에 있으면 빈 리스트([])를 반환

예시 : readlines 메소드

```
data = open('고향의봄.txt', mode='r', encoding='utf-8').readlines()
```

```
type(data)
```

```
print(data)
```

readlines vs. for 문

```
file = open('고향의봄.txt', mode='r', encoding='utf=8')
lines = file.readlines()
print(lines)
```

```
['나의 살던 고향은 꽃피는 산골\n', '복숭아꽃 살구꽃 아기 진달래\n', '울긋불긋 꽃대궐 차리인
동네\n', '그 속에서 놀던 때가 그립습니다\n', '\n', '꽃동네 새동네 나의 옛고향\n', '파란 들
남쪽에서 바람이 불면\n', '냇가에 수양버들 춤추는 동네\n', '그 속에서 놀던 때가 그립습니다']
```

=

```
file = open('고향의봄.txt', mode='r', encoding='utf=8')
lines = []
for line in file:
    lines.append(line)
else:
    print(lines)
```

```
['나의 살던 고향은 꽃피는 산골\n', '복숭아꽃 살구꽃 아기 진달래\n', '울긋불긋 꽃대궐 차리인
동네\n', '그 속에서 놀던 때가 그립습니다\n', '\n', '꽃동네 새동네 나의 옛고향\n', '파란 들
남쪽에서 바람이 불면\n', '냇가에 수양버들 춤추는 동네\n', '그 속에서 놀던 때가 그립습니다']
```

두 코드의 차이는?

```
file = open('고향의봄.txt', mode='r', encoding='utf=8')
data = file.readlines()

type(data)  # readlines()가 반환한 객체의 자료형을 확인
print(data)
```

```
file = open('고향의봄.txt', mode='r', encoding='utf-8')
data = file.read().splitlines()  # 참고 : split('\n')

type(data)  # splitlines()가 반환한 객체의 자료형을 확인
print(data)
```

정리 : 텍스트 파일 읽기

- open 함수

```
open(파일이름, mode='r', encoding='utf-8')
```

  - open 함수를 사용해서 텍스트 파일(.txt)을 열 수 있다
    - 파일을 읽을 때는 mode를 'r'로 설정
      - 기본 모드가 읽기 모드이기 때문에 생략 가능
    - 일반적으로 utf-8 인코딩을 사용

- 파일 읽기 메소드
  - 읽기 모드로 파일을 열어서 파일의 내용을 읽을 때 세 가지 파일 메소드 중 하나를 선택해서 사용
    - 파일객체.read()
    - 파일객체.readline()
    - 파일객체.readlines()
    - 어떤 메소드를 사용할지는 상황에 따라 다르다
  - 파일객체.close()
    - 파일을 읽기용으로 불러온 경우 파일객체.close 메소드는 선택 사항(하지만 만약 동일한 프로그램이 실행되는 동안 같은 파일을 다시 읽으려면 파일을 닫았다가 다시 여는 것이 좋다)

데이터 파일 파싱 및 자료형 변환

- 데이터 파일 파싱(parsing data files)
  - 데이터 세트는 주로 2차원의 표 형식으로 이루어져 있다
  - 따라서 각 줄을 파싱해야 한다
  - 이러한 데이터 세트를 파싱하는 방법 중 흔히 사용하는 방법은 각 줄을 분리해서 2-D 리스트(매트릭스 형태)로 저장하는 것이다
  - 예를 들어, 다음과 같은 성적 데이터가 있다면

성명	점수	성적
제이지	97	A+
어피치	80	B+
무지	60	C-
튜브	95	A0

  - 메모장으로 다음과 같이 작성해서 ‘grades.txt'라는 파일로 저장(콤마 뒤에 공백이 없이 바로 숫자나 문자가 온다)

```
제이지,97,A+
어피치,80,B+
무지,60,C-
튜브,95,A0
```

파싱 1단계

- 파일의 각 행을 리스트의 객체로 담기
  - 파일을 읽기 모드로 연다
  - 개별 학생의 성적 정보를 행 단위로 리스트에 담는다

```
data = open('grades.txt', mode='r', encoding='utf-8').read().splitlines()
print(data)
```

파싱 2단계

- 문자열을 리스트로 변환하기
  - 리스트가 담고 있는 각 객체(문자열)를 리스트로 변환
  - 이때, 문자열을 콤마 기준으로 분할한 리스트로 만든 후 리스트에 다시 담는다

```
for index, student in enumerate(data):
    data[index] = student.split(',')
else:
    print(data)
```

파싱 3단계

- 계산에 필요한 데이터를 숫자로 형변환하기
  - 리스트의 두 번째 값을 문자열에서 정수로 변환

```
for student in data:
    student[1] = int(student[1])
else:
    print(data)
```

파싱 4단계

- 평균 점수 구하기
  - 리스트의 두 번째 값들의 합을 구한다
  - 전체 점수의 합을 전체 학생 수로 나눈다

```
total = 0
for student in data:
    total += student[1]        # 각 줄(열)의 두 번째 컬럼 점수의 합을 구한다
else:
    print(total / len(data))   # 평균을 구해 출력
```

참고 : 3단계와 4단계를 한꺼번에 처리

```
total = 0
for student in data:
      total += int(student[1])
else:
     print(total / len(data))
```

참고 : 1 ~ 4 단계 한꺼번에 처리

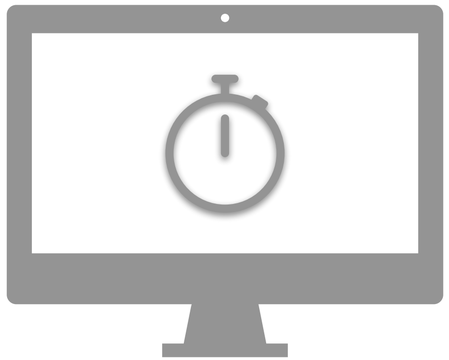
```
data = [int(line.split(',')[1])
            for line in open('grades.txt').read().splitlines()]
print(sum(data) / len(data))
```

CHAPTER 11 실습

파일 처리

Lab Exercises

파일 쓰기



Lab : 리스트 내용을 파일로 저장

- 문자열을 리스트로 변환한 후 리스트의 내용을 파일로 저장하기
  - 다음 문자열을 'pitcher_stats.txt' 파일로 저장
    - `'Kershaw/18/4/2.31,Jansen/5/0/1.32,Wood/16/3/2.72,Hill/12/8/3.32'`
  - 'pitcher_stats.txt' 파일은 다음과 같은 형식으로 저장되어야 한다

```
Kershaw/18/4/2.31
Jansen/5/0/1.32
Wood/16/3/2.72
Hill/12/8/3.32
```

  - 실행 결과 예시

```
Kershaw/18/4/2.31
Jansen/5/0/1.32
Wood/16/3/2.72
Hill/12/8/3.32
```

Lab : 학생 정보를 CSV 파일로 저장

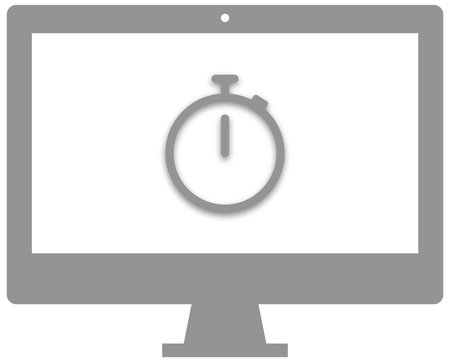
- 학생 정보를 CSV 파일로 저장하기
  - 학생 세 명의 이름, 성별, 나이를 입력받아 CSV(comma separated value) 형식의 파일로 저장하는 프로그램을 구현
  - 저장하는 파일의 이름은 '`students.csv`'로 하며, 파일의 인코딩 형식은 **`utf-8`**로 한다
  - '`students.csv`' 파일은 다음과 같은 형식으로 저장해야 한다
    - 이름,성별,나이
      제이지,M,22
      어피치,F,21
      라이언,M,20
  - 실행 결과 예시

```
학생1의 이름을 입력하세요: 제이지
학생1의 성별을 입력하세요(F/M): M
학생1의 나이를 입력하세요: 22

학생2의 이름을 입력하세요: 어피치
학생2의 성별을 입력하세요(F/M): F
학생2의 나이를 입력하세요: 21

학생3의 이름을 입력하세요: 라이언
학생3의 성별을 입력하세요(F/M): M
학생3의 나이를 입력하세요: 20
```

파일 읽기



Lab : 파일의 각 줄 번호 출력

- 파일을 읽어서 각 줄에 번호를 붙여 출력하기
  - 사용자로부터 텍스트 파일 이름을 입력받아 해당 파일을 읽은 후 각 줄의 줄 번호와 내용을 출력하는 프로그램 작성

  - 실행 결과 예시

```
파일 이름을 입력하세요...:  고향의봄.txt
    1: 나의 살던 고향은 꽃피는 산골
    2: 복숭아꽃 살구꽃 아기 진달래
    3: 울긋불긋 꽃대궐 차리인 동네
    4: 그 속에서 놀던 때가 그립습니다
    5:
    6: 꽃동네 새동네 나의 옛고향
    7: 파란 들 남쪽에서 바람이 불면
    8: 냇가에 수양버들 춤추는 동네
    9: 그 속에서 놀던 때가 그립습니다
```

Lab : 영어를 대문자로 출력

- 파일 안의 영어를 대문자로 변환해서 출력하기
  - 사용자로부터 텍스트 파일 이름을 입력받아 해당 파일을 읽은 후 영어 단어가 있는 경우 모두 대문자로 변환해서 출력
  - 만약 텍스트 파일이 없으면 The Zen of Python(https://www.python.org/dev/peps/pep-0020/#id3)을 방문해서 19문장으로 된 문서를 전체 복사하여 메모장에 붙여놓은 후, 파일 이름을 'zen_of_python.txt'로 해서 이 문서를 사용
  - 실행 결과 예시

```
파일 이름을 입력하세요...: zen_of_python.txt
BEAUTIFUL IS BETTER THAN UGLY.
EXPLICIT IS BETTER THAN IMPLICIT.
SIMPLE IS BETTER THAN COMPLEX.
COMPLEX IS BETTER THAN COMPLICATED.
FLAT IS BETTER THAN NESTED.
SPARSE IS BETTER THAN DENSE.
READABILITY COUNTS.
SPECIAL CASES AREN'T SPECIAL ENOUGH TO BREAK THE RULES.
ALTHOUGH PRACTICALITY BEATS PURITY.
ERRORS SHOULD NEVER PASS SILENTLY.
UNLESS EXPLICITLY SILENCED.
IN THE FACE OF AMBIGUITY, REFUSE THE TEMPTATION TO GUESS.
THERE SHOULD BE ONE-- AND PREFERABLY ONLY ONE --OBVIOUS WAY TO DO IT.
ALTHOUGH THAT WAY MAY NOT BE OBVIOUS AT FIRST UNLESS YOU'RE DUTCH.
NOW IS BETTER THAN NEVER.
ALTHOUGH NEVER IS OFTEN BETTER THAN *RIGHT* NOW.
IF THE IMPLEMENTATION IS HARD TO EXPLAIN, IT'S A BAD IDEA.
IF THE IMPLEMENTATION IS EASY TO EXPLAIN, IT MAY BE A GOOD IDEA.
NAMESPACES ARE ONE HONKING GREAT IDEA -- LET'S DO MORE OF THOSE!
```

Lab : 영어를 대문자로 변환 후 쓰기

- 파일 안의 영어를 대문자로 변환한 후 다른 파일 이름으로 저장하기
  - 바로 앞에서 실습한 'Lab : 영어를 대문자로 출력'과 같은 작업을 실행하지만, 이번에는 결괏값을 출력하지 않고 새로운 텍스트 파일로 생성하는 프로그램을 구현
  - 새로운 파일의 이름은 기존 파일명 앞에 'UPPER'를 추가해서 기존 파일과 같은 디렉토리에 저장
    - 예) 'zen_of_python.txt' —> 'UPPERzen_of_python.txt'
  - 실행 결과 예시

```
파일 이름을 입력하세요...: zen_of_python.txt
'UPPERzen_of_python.txt' 파일을 저장했습니다.
```

Lab : 파일 안의 특정 단어 검색

- 파일 안에 특정 단어가 있는지 검색하기
  - 검색하고자 하는 텍스트 파일 이름과 검색 단어를 입력받아 대상 파일에서 특정 단어가 있으면 몇 번째 줄에 그 단어가 있는지 해당 줄 번호와 그 줄의 내용을 출력하는 프로그램을 구현

  - 실행 결과 예시

```
파일 이름을 입력하세요...: 고향의봄.txt
검색 단어를 입력하세요...: 꽃
    1: 나의 살던 고향은 꽃피는 산골
    2: 복숭아꽃 살구꽃 아기 진달래
    3: 울긋불긋 꽃대궐 차리인 동네
    6: 꽃동네 새동네 나의 옛고향
```

Lab : 텍스트 파일 기본 통계 분석

- 텍스트 파일의 기본 구성을 분석하기
  - 사용자로부터 텍스트 파일 이름을 입력받아 해당 파일을 읽은 후, 그 파일 전체가 몇 줄로 되어 있는지(빈 줄 포함), 몇 개의 단어로 구성되어 있는지, 그리고 몇 개의 글자로 구성되어 있는지 분석해서 그 결괏값을 출력하는 프로그램을 구현
  - 분석할 때 고려할 사항은 다음과 같다
    - **단어**의 구분은 빈 칸을 기준으로 한다
    - **글자**의 수를 계산할 때 부호나 기호를 생략한 알파벳 또는 한글만으로 한다
    - 예) '나의 살던 고향은 꽃피는 산골'의 경우
      - 단어의 개수 : 5
      - 글자의 개수 : 12

💡 `string` 모듈의 `punctuation` 속성 값으로 문장 부호나 기호를 확인할 수 있다

- 사용법 : `string.punctuation`

- 실행 결과 예시

```
파일 이름을 입력하세요...: 고향의봄.txt
=> 고향의봄.txt 파일은 9 줄, 36 단어, 97 문자로 구성되어 있습니다.
```

Lab : CSV 형식의 파일 데이터 구조화

- CSV 형식의 파일 데이터 구조화하기
  - 아래 링크에서 'enrollments.csv' 파일을 내려받은 후 이 파일을 읽어 각 행의 데이터를 하나의 딕셔너리로 구성
    - https://github.com/snu-python/hw-data
  - 전체 데이터는 모든 행이 딕셔너리로 이루어진 리스트로 구조화한다
  - 이 때 첫 번째 행의 모든 값을 딕셔너리의 키로 사용
    - 따라서 첫 번째 행은 전체 데이터에서 제외
  - 처리한 결과 중 첫 두 개만 출력
  - 실행 결과 예시

```
{'account_key': '448', 'status': 'canceled', 'join_date': '2014-11-10', 'cancel_date':
'2015-01-14', 'days_to_cancel': '65', 'is_udacity': 'True', 'is_canceled': 'True'}

{'account_key': '448', 'status': 'canceled', 'join_date': '2014-11-05', 'cancel_date':
'2014-11-10', 'days_to_cancel': '5', 'is_udacity': 'True', 'is_canceled': 'True'}
```

Lab : 텍스트 파일에 있는 각 단어의 출현 횟수를 계산

- 딕셔너리로 각 단어의 출현 횟수 계산하기
  - 사용자로부터 텍스트 파일 이름을 입력받아 그 파일 안에 들어 있는 모든 단어와 각 단어의 출현 횟수를 알파벳(또는 가나다) 순으로 나열해서 출력
  - 영어와 한국어 둘 다 처리할 수 있어야 하며, 프로그램을 간단히 작성하기 위해 한국어인 경우 형태소 분석은 하지 않고 각 단어를 구분짓는 기준으로 띄어쓰기를 사용

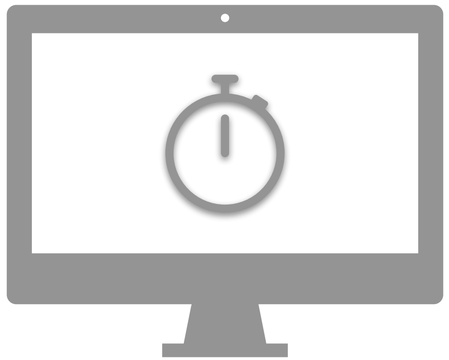
💡 딕셔너리를 사용하면 항목들의 출현 횟수를 저장하기 용이하다

💡 딕셔너리의 `get` 메소드를 사용하면 코드가 간결해진다

  - 실행 결과 예시

```
파일 이름을 입력하세요...: 고향의봄.txt
'고향은' : 1번 출현
'그' : 2번 출현
'그립습니다' : 2번 출현
...
'춤추는' : 1번 출현
'파란' : 1번 출현
```

파일 파싱



Lab : 투수 통계 데이터 파일 파싱

- 투수 통계 데이터 파일 파싱하기
  - 앞서 **'Lab : 리스트 내용을 파일로 저장'**에서 만든 '`pitcher_stats.txt`' 파일로부터 투수 통계 데이터를 읽어와
  - 오른쪽과 같이 텍스트 데이터를 파싱한다
    - `Name`은 문자열로, `W`와 `L`은 정수로, `ERA`은 실수로 처리한다

Name	W	L	ERA
Kershaw	18	4	2.31
Jansen	5	0	1.32
Wood	16	3	2.27
Hill	12	8	3.32

- 텍스트 파일의 단락은 슬래시 '`/`'로 구분(delimited)하고 있다
  - '`pitcher_stats.txt`' 파일은 다음과 같은 형식으로 저장되어 있다

```
Kershaw/18/4/2.31
Jansen/5/0/1.32
Wood/16/3/2.72
Hill/12/8/3.32
```

  - 실행 결과 예시

```
[['Kershaw', 18, 4, 2.31],
 ['Jansen', 5, 0, 1.32],
 ['Wood', 16, 3, 2.72],
 ['Hill', 12, 8, 3.32]]
```