1-1. 데이터분석과 프로그래밍

- 수작업이 불가능한 방대한 규모의 데이터 처리 가능
  - 데이터의 양이 많으면 수정하는데 시간이 오래 걸리거나, 오류가 발생하기 쉬움
  - 수집, 수백, 수천개의 파일을 처리해야 하는 경우도 있음
- 작업의 자동화 가능
- 데이터 매니지먼트 ➔ ETL(Extract Transform Load)
  - 고객이나 공급자로부터 데이터 수집, 보관이 필요한 데이터 추출
  - 분석을 위해 데이터 형태나 자료형 변환
  - 데이터베이스 등 저장소에 저장

1-2. 파이썬 라이브러리

- 표준 라이브러리만 가지고 다양한 포맷의 파일을 읽을 수 있음
  - 텍스트
  - CSV
  - JSON
  - HTML
  - XML
- xlrd, xlwt: 엑셀 통합 문서를 파싱하고 작성하는 함수 제공
- mysqlclient: MySQL 데이터베이스에 연결하고 쿼리를 실행하는 함수 제공
- Pandas: 다양한 형식의 파일을 읽고, 관리하고, 필터링 및 변환, 데이터 병합 기초 통계 계산, 시각화 등 데이터 분석에 특화된 함수 제공
- statsmodels: 선형회귀모형, 일반화선형모형, 분류 모형 등 다양한 통계 모형을 계산하는 함수 제공
- 사이킷런: 회귀, 분류, 군집화 등 머신러닝과 데이터 전처리, 차원 축소, 교차검증 등 통계분석용 함수 제공

1-3. 파이썬 패턴

- 질문1) 문자열 내에 등장하는 패턴의 횟수 구하기

```
import re

string = "The quick brown fox jumps over the lazy dog."
string_list = string.split()
pattern = re.compile(r"The", re.I)
count = 0

for word in string_list:
    if pattern.search(word):
        count += 1
print("Output #38: {0:d}".format(count))
```

demo_02.py

1-3. 파이썬 패턴

- (?P<이름>)
  - 일치한 문자열을 나중에 프로그램에서 재 사용 가능

```
string = "The quick brown fox jumps over the lazy dog."
string_list = string.split()
pattern = re.compile(r"(?P<my_matching_word>The)", re.I)
print("Output #39:")
for word in string_list:
    if pattern.search(word):
        print("{:s}".format(pattern.search(word).group('my_matching_word')))
```

demo_02.py

1-4. 파이썬 날짜

```
# Print today's date, as well as the year, month, and day elements
today = date.today()
print("Output #41: today: {0!s}".format(today))
print("Output #42: {0!s}".format(today.year))
print("Output #43: {0!s}".format(today.month))
print("Output #44: {0!s}".format(today.day))
current_datetime = datetime.today()
print("Output #45: {0!s}".format(current_datetime))

# Calculate a new date using a timedelta
one_day = timedelta(days=-1)
yesterday = today + one_day
print("Output #46: yesterday: {0!s}".format(yesterday))
eight_hours = timedelta(hours=-8)
print("Output #47: {0!s} {1!s}".format(eight_hours.days, eight_hours.seconds))
```

demo_03.py

1-4. 파이썬 날짜

~ 이전코드

```
# Calculate the amount of time between two dates and grab the first element, the number of days
date_diff = today - yesterday
print("Output #48: {0!s}".format(date_diff))
print("Output #49: {0!s}".format(str(date_diff).split()[0]))

# Create a string with a specific format from a date object
print("Output #50: {:s}".format(today.strftime('%m/%d/%Y')))
print("Output #51: {:s}".format(today.strftime('%b %d, %Y')))
print("Output #52: {:s}".format(today.strftime('%Y-%m-%d')))
print("Output #53: {:s}".format(today.strftime('%B %d, %Y')))
```

demo_03.py

1-4. 파이썬 날짜

~ 이전코드

```
# Create a datetime object with a specific format
# from a string representing a date
date1 = today.strftime('%m/%d/%Y')
date2 = today.strftime('%b %d, %Y')
date3 = today.strftime('%Y-%m-%d')
date4 = today.strftime('%B %d, %Y')

# Two datetime objects and two date objects
# based on the four strings that have different date formats
print("Output #54: {!s}".format(datetime.strptime(date1, '%m/%d/%Y')))
print("Output #55: {!s}".format(datetime.strptime(date2, '%b %d, %Y')))

# Show the date portion only
print("Output #56: {!s}".format(datetime.date(datetime.strptime(date3, '%Y-%m-%d'))))
print("Output #57: {!s}".format(datetime.date(datetime.strptime(date4, '%B %d, %Y'))))
```

demo_03.py

1-5. 파이썬 리스트

a_list = [1, 2, 3]

- 리스트 분할

  a_list[0:2]

  a_list[:2]

  a_list[1:3]

  a_list[1:]

- 리스트 복사

  a_new_list = a_list[:]

- in, not in

  a = 2 in a_list

  b = 6 not in a_list

demo_04.py

1-5. 파이썬 리스트

- append()
  - 리스트의 마지막에 원소 추가
- remove()
  - 리스트 내 특정 원소 제거
- pop()
  - 리스트의 마지막 원소 제거
- reverse()
  - 리스트 반전
- sort()
  - 인플레이스(in-place)로 리스트 정렬
- sorted()
  - key와 결합하여 각 리스트에 있는 특정 인덱스에 있는 값에 따라 리스트 정렬
- itemgetter()
  - 리스트 내 다양한 위치의 인덱스 값에 따라 리스트를 정렬

demo_05.py

1-5. 파이썬 튜플

- 변경하지 못하는 리스트
  - 변경 관련 함수가 존재하지 않음
- 튜플 풀기
  - my_tuple = ('x', 'y', 'z')
  - one, two, three = my_tuple
  - var1, var2 = var2, var1
- 튜플 ←→ 리스트
  - tuple()
  - list()

demo_06.py

1-5. 파이썬 딕셔너리

- 연관 배열 (associative array)
  - 키-값 저장소 (Key-value store)
  - Hash
- 리스트는 인덱스로 개별 값에 접근하나, 딕셔너리는 키를 이용하여 접근
- 딕셔너리에는 정렬 방법이 내장되어 있지 않음
- 딕셔너리의 경우, 필요에 따라 새로운 위치(키)를 생성 가능
- 딕셔너리는 새로운 아이템을 삽입하는 경우 인덱스 값들 재할당할 필요가 없기 때문에, 값을 추가하거나 검색할 경우 응답 속도가 빠르다
- 많은 데이터 처리해야 하는 경우 좋음

demo_07.py

1-6. 파이썬 간결한 for문

- list, set, dictionary 축약
- list 축약
  - row_to_keep = [row for row in my_data if row[2] > 5]
- set 축약
  - set_of_tuples = { x for x in my_data}
- dictionary 축약
  - my_results = {key : value for key, value in my_dictionary.items() if value > 10}

demo_08.py

1-7. 파이썬 텍스트 파일 읽기

- with 구문을 사용하여 자동으로 파일 객체 닫기

```
print("Output #142:")
with open(input_file, 'r', newline='') as filereader:
    for row in filereader:
        print("{}".format(row.strip()))
```

- glob을 이용해 다수의 텍스트 파일 읽기

```
for input_file in glob.glob(os.path.join(inputPath,'*.txt')):
    with open(input_file, 'r', newline='') as filereader:
        for row in filereader:
            print("{}".format(row.strip()))
```

demo_10.py

1-7. 파이썬 CSV 파일 쓰기(테스트)

```
output_file = 'write_to_file.txt'
my_numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
max_index = len(my_numbers)
filewriter = open(output_file, 'a')
for index_value in range(len(my_numbers)):
    if index_value < (max_index-1):
        filewriter.write(str(my_numbers[index_value])+',')
    else:
        filewriter.write(str(my_numbers[index_value])+'\n')
filewriter.close()
print("Output #145: Output appended to file")
```

demo_11.py

2-1. CSV 파일 읽고/쓰기(기본 파이썬 코드)

```
with open(input_file, 'r', newline='') as filereader:
    with open(output_file, 'w', newline='') as filewriter:
        header = filereader.readline()
        header = header.strip()
        header_list = header.split(',')
        print(header_list)
        filewriter.write(','.join(map(str,header_list))+'\n')
        for row in filereader:
            row = row.strip()
            row_list = row.split(',')
            print(row_list)
            filewriter.write(','.join(map(str,row_list))+'\n')
```

basic_demo12.py

Pandas

- 통계 분석을 위한 R의 Dataframe 데이터 타입과 같은 Pandas dataframe을 사용
- Pandas DataFrame
  - 테이블 형식의 데이터 (tabular, rectangular grid 등으로 불림)를 다룰 때 사용
  - Column, Row(데이터), Index
  - Numpy의 ndarray, Pandas의 DataFrame, Series, Python의 dictionary, list 등으로 부터 생성 가능

```
df = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]))
print(df.shape)
print(len(df.index))
print(list(df.columns))
```

```
(2, 3)
2
[0, 1, 2]
```

basic_demo12.py

Pandas

- Dataframe에서 특정 컬럼이나 로우 선택
  - iloc, loc, ix

```
df = pd.DataFrame({"A": [1,4,7], "B":[2,5,8], "C":[3,6,9]})

print(df.iloc[0])
print(df.loc[0])
print(df.ix[0])

print(df.loc[:, 'A'])
print(df['A'])

print(df.ix[0]['A'])
print(df.ix[0]['B'])
```

```
A   1
B   2
C   3
Name: 0, dtype: int64
```

```
0   1
1   4
2   7
Name: A, dtype: int64
```

```
 1
 2
```

basic_demo12.py

Pandas

- Dataframe에서 특정 컬럼이나 로우 선택
  - iloc: 인덱스와 상관 없이 순서를 보고 row를 불러옴

    ex) df.iloc[1] → 두번째(0, 1) row
  - ix: 해당 인덱스의 row를 불러옴

    ex) df.ix[7] → 인텍스가 7인 row
  - 인덱스에 문자, 숫자가 혼합 된 형태라면 ix도 순서로 row를 불러옴

```
df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), index= [2, 'A', 4],
columns=[48, 49, 50])
print(df)
print(df.loc[2])
print(df.iloc[2])
print(df.ix[2])
```

```
 48  49  50
2   1   2   3
A   4   5   6
4   7   8   9
```

```
48    1
49    2
50    3
Name: 2, dtype: int32
```

```
48    7
49    8
50    9
Name: 4, dtype: int32
```

```
48    7
49    8
50    9
Name: 4, dtype: int32
```

basic_demo12.py

2-1. CSV 파일 읽고/쓰기(Pandas)

```
data_frame = pd.read_csv(input_file)
print(data_frame)
data_frame.to_csv(output_file, index=False)
```

	Supplier Name	Invoice Number	Part Number	Cost	Purchase Date
0	Supplier X	001-1001	2341	$500.00	1/20/14
1	Supplier X	001-1001	2341	$500.00	1/20/14
2	Supplier X	001-1001	5467	$750.00	1/20/14
3	Supplier X	001-1001	5467	$750.00	1/20/14
4	Supplier Y	50-9501	7009	$250.00	1/30/14
5	Supplier Y	50-9501	7009	$250.00	1/30/14
6	Supplier Y	50-9505	6650	$125.00	2/3/14
7	Supplier Y	50-9505	6650	$125.00	2/3/14
8	Supplier Z	920-4803	3321	$615.00	2/3/14
9	Supplier Z	920-4804	3321	$615.00	2/10/14
10	Supplier Z	920-4805	3321	$615.00	2/17/14
11	Supplier Z	920-4806	3321	$615.00	2/24/14

pandas_demo13.py

2-1. CSV 파일 읽고/쓰기(기본 파이썬 코드)

- 기본 문자열 파싱 실패
  - 마지막 두행의 Cost 열의 값 변경

```
12    Supplier Z,920-4805,3321,$615.00,2/17/14
13    Supplier Z,920-4806,3321,$615.00,2/24/14
```

```
12    Supplier Z,920-4805,3321,$6,015.00,2/17/14
13    Supplier Z,920-4806,3321,$1,006,015.00,2/24/14
```

  - 실행 결과

```
['Supplier Z', '920-4805', '3321', '"$6', '015.00"', '2/17/14']
['Supplier Z', '920-4806', '3321', '"$1', '006', '015.00" ', '2/24/14']
```

basic_demo12.py

2-1. CSV 파일 읽고/쓰기(기본 모듈)

- 기본 csv 모듈
  - 데이터 값에 포함된 쉼표 및 기타 복잡한 패턴을 정확하게 처리 가능

```
with open(input_file, 'r', newline='') as csv_in_file:
    with open(output_file, 'w', newline='') as csv_out_file:
        filereader = csv.reader(csv_in_file, delimiter=',')
        filewriter = csv.writer(csv_out_file, delimiter=',')
        for row_list in filereader:
            filewriter.writerow(row_list)
```

```
['Supplier Z', '920-4805', '3321', '$6,015.00', '2/17/14']
['Supplier Z', '920-4806', '3321', '$1,006,015.00 ', '2/24/14']
```

basic_demo14.py

2-1. CSV 파일 읽고/쓰기(기본 모듈)

- 기본 csv 모듈
  - 데이터 값에 포함된 쉼표 및 기타 복잡한 패턴을 정확하게 처리 가능

```
with open(input_file, 'r', newline='') as csv_in_file:
    with open(output_file, 'w', newline='') as csv_out_file:
        filereader = csv.reader(csv_in_file, delimiter=',')
        filewriter = csv.writer(csv_out_file, delimiter=',')
        for row_list in filereader:
            filewriter.writerow(row_list)
```

```
['Supplier Z', '920-4805', '3321', '$6,015.00', '2/17/14']
['Supplier Z', '920-4806', '3321', '$1,006,015.00 ', '2/24/14']
```

basic_demo14.py

2-2. 특정 행 필터링

- 특정 조건을 충족하는 행을 필터링
- 특정 집합의 값을 포함하는 행을 필터링
- 정규 표현식을 활용한 필터링

```
for   row   in   filerader:
      *** if 행에 있는 값이 특정한 규칙을 충족한다면 ***
            처리 1
      else:
            처리 2
```

2-2. 특정 행 필터링(기본 csv 모듈)

- 특정 조건을 충족하는 행을 필터링

  ex) 비용이 특정 값을 초과하는 모든 행을 선택하여 데이터셋으로 만들 경우

  ex) 구매 일자가 특정 날짜 이전인 모든 행을 데이터셋으로 만들 경우

- Supplier Name이 **Supplier Z** or **Cost가 $600.00 이상**인 행만 필터링 → 파일 출력

```
for row_list in filereader:
    supplier = str(row_list[0]).strip()
    cost = str(row_list[3]).strip('$').replace(',', '')
    if supplier == 'Supplier Z' or float(cost) > 600.0:
        filewriter.writerow(row_list)
```

```
Supplier Name,Invoice Number,Part Number,Cost,Purchase Date
Supplier X,001-1001,5467,$750.00 ,1/20/14
Supplier X,001-1001,5467,$750.00 ,1/20/14
Supplier Z,920-4803,3321,$615.00 ,2002-03-14
Supplier Z,920-4804,3321,$615.00 ,2002-10-14
Supplier Z,920-4805,3321,"$6,015.00",2/17/14
Supplier Z,920-4806,3321,"$1,006,015.00 ",2/24/14
```

basic_demo15.py

2-2. 특정 행 필터링(Pandas)

- 특정 조건을 충족하는 행을 필터링
- loc()

  - 특정 행과 열을 동시에 선택

```
data_frame.loc[(data_frame['Supplier Name'].str.contains('Z')) | (data_frame['Cost'] > 600.0), :]
```

```
Supplier Name,Invoice Number,Part Number,Cost,Purchase Date
Supplier X,001-1001,5467,$750.00 ,1/20/14
Supplier X,001-1001,5467,$750.00 ,1/20/14
Supplier Z,920-4803,3321,$615.00 ,2002-03-14
Supplier Z,920-4804,3321,$615.00 ,2002-10-14
Supplier Z,920-4805,3321,$615.00 ,2/17/14
Supplier Z,920-4806,3321,$615.00 ,2/24/14
```

pandas_demo16.py

2-2. 특정 행 필터링(기본 csv 모듈)

- 특정 집합의 값을 포함하는 행의 필터링
  - Supplier Name 열에서 집합 {**Supplier X, Supplier Y**} 중 한 값을 포함하는 모든 행
  - Purchase Date 열에서 구매 일자가 집합 {‘1/20/14’, ‘1/30/14’} 중 한 값을 포함하는 모든 행

```
important_dates = ['1/20/14', '1/30/14']

with open(input_file, 'r', newline='') as csv_in_file:
    with open(output_file, 'w', newline='') as csv_out_file:
        filereader = csv.reader(csv_in_file)
        filewriter = csv.writer(csv_out_file)
        header = next(filereader)
        filewriter.writerow(header)
        for row_list in filereader:
            a_date = row_list[4]
            if a_date in important_dates:
                filewriter.writerow(row_list)
```

basic_demo17.py

2-2. 특정 행 필터링(Pandas)

- 특정 집합의 값을 포함하는 행의 필터링
  - Supplier Name 열에서 집합 {**Supplier X, Supplier Y**} 중 한 값을 포함하는 모든 행
  - Purchase Date 열에서 구매 일자가 집합 {‘1/20/14’, ‘1/30/14’} 중 한 값을 포함하는 모든 행

```
data_frame = pd.read_csv(input_file)

important_dates = ['1/20/14', '1/30/14']
data_frame_value_in_set = data_frame.loc[data_frame['PurchaseDate'].isin(important_dates), :]

data_frame_value_in_set.to_csv(output_file, index=False)
```

pandas_demo18.py

2-2. 특정 행 필터링(기본 csv 모듈)

- 패턴/정규 표현식을 활용한 필터링
  - Invoice Number 열의 데이터 값이 001-로 시작하는 모든 행
  - Supplier Name 열의 데이터 값에 Y가 포함되어 있는 모든 행

```
pattern = re.compile(r'(?P<my_pattern_group>^001-.*)', re.I)

with open(input_file, 'r', newline='') as csv_in_file:
    with open(output_file, 'w', newline='') as csv_out_file:
        filereader = csv.reader(csv_in_file)
        filewriter = csv.writer(csv_out_file)
        header = next(filereader)
        filewriter.writerow(header)
        for row_list in filereader:
            invoice_number = row_list[1]
            if pattern.search(invoice_number):
                filewriter.writerow(row_list)
```

basic_demo19.py

2-2. 특정 행 필터링(Pandas)

- 특정 집합의 값을 포함하는 행의 필터링
  - Supplier Name 열에서 집합 {**Supplier X, Supplier Y**} 중 한 값을 포함하는 모든 행
  - Purchase Date 열에서 구매 일자가 집합 {‘1/20/14’, ‘1/30/14’} 중 한 값을 포함하는 모든 행

```
data_frame = pd.read_csv(input_file)
data_frame_value_matches_pattern =
       data_frame.ix[data_frame['Invoice Number'].str.startswith("001-"), :]

data_frame_value_matches_pattern.to_csv(output_file, index=False)
```

pandas_demo20.py

2-3. 특정 열 선택(기본 csv 모듈)

- 열의 인덱스 값을 사용하는 방법
  - row[0], row[-1]
  - Supplier Name 및 Cost 열만 포함

```
my_columns = [0, 3]

with open(input_file, 'r', newline='') as csv_in_file:
    with open(output_file, 'w', newline='') as csv_out_file:
        filereader = csv.reader(csv_in_file)
        filewriter = csv.writer(csv_out_file)
        for row_list in filereader:
            row_list_output = [ ]
            for index_value in my_columns:
                row_list_output.append(row_list[index_value])
            filewriter.writerow(row_list_output)
```

basic_demo21.py

2-3. 특정 열 선택(Pandas)

- 열의 인덱스 값을 사용하는 방법
  - Supplier Name 및 Cost 열만 포함
  - iloc() ➔ 정수기반 위치 (Integer Location)

```
data_frame = pd.read_csv(input_file)
data_frame_column_by_index = data_frame.iloc[:, [0, 3]]

data_frame_column_by_index.to_csv(output_file, index=False)
```

pandas_demo22.py

2-3. 특정 열 선택(기본 csv 모듈)

- 열의 헤더를 사용하는 방법
  - Invoice Number 및 Purchase Date 열만 포함

```
my_columns = ['Invoice Number', 'Purchase Date']
my_columns_index = []

with open(input_file, 'r', newline='') as csv_in_file:
    with open(output_file, 'w', newline='') as csv_out_file:
        filereader = csv.reader(csv_in_file)
        filewriter = csv.writer(csv_out_file)
        header = next(filereader)
        for index_value in range(len(header)):
            if header[index_value] in my_columns:
                my_columns_index.append(index_value)
        filewriter.writerow(my_columns)
        for row_list in filereader:
            row_list_output = [ ]
            for index_value in my_columns_index:
                row_list_output.append(row_list[index_value])
            filewriter.writerow(row_list_output)
```

basic_demo23.py

2-3. 특정 열 선택(Pandas)

- 열의 헤더를 사용하는 방법
  - Invoice Number 및 Purchase Date 열만 포함

```
data_frame = pd.read_csv(input_file)
data_frame_column_by_name = data_frame.loc[:, ['Invoice Number', 'Purchase Date']]

data_frame_column_by_name.to_csv(output_file, index=False)
```

pandas_demo24.py

2-4. 연속 된 행 선택(기본 csv 모듈)

- 분석에 필요 없는 맨 위 또는 맨 아래의 행 처리
  - csv 파일에 필요 없는 행 삽입 후 실행 (A1:A3, 제일 마지막 행)

```
row_counter = 0
with open(input_file, 'r', newline='') as csv_in_file:
    with open(output_file, 'w', newline='') as csv_out_file:
        filereader = csv.reader(csv_in_file)
        filewriter = csv.writer(csv_out_file)
        for row in filereader:
            if row_counter >= 3 and row_counter <= 15:
                filewriter.writerow([value.strip() for value in row])
            row_counter += 1
```

basic_demo25.py

2-4. 연속 된 행 선택(Pandas)

- 분석에 필요 없는 맨 위 또는 맨 아래의 행 처리
  - csv 파일에 필요 없는 행 삽입 후 실행 (A1:A3, 제일 마지막 행)
- drop()
  - 행 또는 열 삭제 함수
- iloc()
  - 열 헤더 행 선택 ➔ data_frame.columns
- reindex()
  - 새로운 인덱스에 맞추는 함수

```
data_frame = pd.read_csv(input_file, header=None)

data_frame = data_frame.drop([0,1,2,16,17,18])
data_frame.columns = data_frame.iloc[0]
data_frame = data_frame.reindex(data_frame.index.drop(3))

data_frame.to_csv(output_file, index=False)
```

pandas_demo26.py

2-5. 헤더 추가 (기본 csv 모듈)

- 헤더를 추가하기 위해서는 열 헤더가 포함된 헤더 행을 삭제 후 새로 생성

```
with open(input_file, 'r', newline='') as csv_in_file:
    with open(output_file, 'w', newline='') as csv_out_file:
        filereader = csv.reader(csv_in_file)
        filewriter = csv.writer(csv_out_file)
        header_list = ['Supplier Name', 'Invoice Number', \
                       'Part Number', 'Cost', 'Purchase Date']
        filewriter.writerow(header_list)
        for row in filereader:
            filewriter.writerow (row)
```

basic_demo27.py

2-5. 헤더 추가 (Pandas)

- read_csv()
  - 헤더 행이 없이 입력 파일을 읽을 수 있음

```
header_list = ['Supplier Name', 'Invoice Number', 'Part Number', 'Cost', 'Purchase Date']
data_frame = pd.read_csv(input_file, header=None, names=header_list)

data_frame.to_csv(output_file, index=False)
```

pandas_demo28.py

2-6. 여러 개의 CSV 파일 읽기

- 파이썬 내장 모듈인 glob 사용

```
file_counter = 0
for input_file in glob.glob(os.path.join(input_path,'sales_*')):
    row_counter = 1
    with open(input_file, 'r', newline='') as csv_in_file:
        filereader = csv.reader(csv_in_file)
        header = next(filereader)
        for row in filereader:
            row_counter += 1
    print('{0!s}: \t{1:d} rows \t{2:d} columns'.format(\
    os.path.basename(input_file), row_counter, len(header)))
    file_counter += 1
print('Number of files: {0:d}'.format(file_counter))
```

```
sales_february_2014.csv:        7 rows  5 columns
sales_january_2014.csv:         7 rows  5 columns
sales_march_2014.csv:   7 rows  5 columns
Number of files: 3
```

basic_demo29.py

2-7. 여러 파일의 데이터 합치기 (기본 csv 모듈)

```
first_file = True
for input_file in glob.glob(os.path.join(input_path,'sales_*')):
    print(os.path.basename(input_file))
    with open(input_file, 'r', newline='') as csv_in_file:
        with open(output_file, 'a', newline='') as csv_out_file:
            filereader = csv.reader(csv_in_file)
            filewriter = csv.writer(csv_out_file)
            if first_file:
                for row in filereader:
                    filewriter.writerow(row)
                first_file = False
            else:
                header = next(filereader)
                for row in filereader:
                    filewriter.writerow(row)
```

sales_february_2014.csv
sales_january_2014.csv
sales_march_2014.csv

basic_demo30.py

2-7. 여러 파일의 데이터 합치기 (Pandas)

- 각 입력 파일을 데이터프레임으로 읽어 들이고 all_data_frame에 추가 ➔ concat() 함수 사용
- concat()
  - axis 인수를 통해 데이터프레임 병합
    - axis=0(수직), axis=1(수평)

```
all_files = glob.glob(os.path.join(input_path,'sales_*'))

all_data_frames = []
for file in all_files:
    data_frame = pd.read_csv(file, index_col=None)
    all_data_frames.append(data_frame)
data_frame_concat = pd.concat(all_data_frames, axis=0,
ignore_index=True)

data_frame_concat.to_csv(output_file, index = False)
```

pandas_demo31.py

2-8. 파일에서 데이터 값의 합계 및 평균 계산(기본 csv 모듈)

```
output_header_list = ['file_name', 'total_sales', 'average_sales']

csv_out_file = open(output_file, 'a', newline='')
filewriter = csv.writer(csv_out_file)
filewriter.writerow(output_header_list)
```

```
        header = next(filereader)
        total_sales = 0.0
        number_of_sales = 0.0
        for row in filereader:
            sale_amount = row[3]
            total_sales += float(str(sale_amount).strip('$').replace(',',''))
            number_of_sales += 1.0
        average_sales = '{0:.2f}'.format(total_sales / number_of_sales)
```

basic_demo32.py

2-8. 파일에서 데이터 값의 합계 및 평균 계산(Pandas)

- sum() 및 mean() 같은 통계 함수 제공

```
all_files = glob.glob(os.path.join(input_path,'sales_*'))
all_data_frames = []
for input_file in all_files:
    data_frame = pd.read_csv(input_file, index_col=None)

    total_sales = pd.DataFrame([float(str(value).strip('$').replace(',','')) \
                                for value in data_frame.loc[:, 'Sale Amount']]).sum()

    average_sales = pd.DataFrame([float(str(value).strip('$').replace(',','')) \
                                for value in data_frame.loc[:, 'Sale Amount']]).mean()

    data = {'file_name': os.path.basename(input_file),
            'total_sales': total_sales,
            'average_sales': average_sales}

    all_data_frames.append(pd.DataFrame(data, columns=['file_name', 'total_sales', 'average_sales']))

data_frames_concat = pd.concat(all_data_frames, axis=0, ignore_index=True)
```

pandas_demo33.py