1. 파이썬 기반의 머신러닝과 생태계 이해

goblurry · 4일 전

ML 파완머

파이썬 머신러닝 완벽가이드

목록 보기 1/1

📊 **파이썬 머신러닝 완벽 가이드 (위키북스, 개정 2판)** 교재를 바탕으로 학습한 내용을 정리한 포스트입니다.
https://github.com/goblurry/ML-Study 에서 예시 코드를 확인할 수 있습니다.

> **[1장] 파이썬 기반의 머신러닝과 생태계 이해**
>
> - 머신러닝의 개념
> - 파이썬 주요 패키지
> - 넘파이
> - 데이터 핸들링: 판다스

1. 머신러닝의 개념

1) 머신러닝의 개념

머신러닝은 **데이터를 기반으로 패턴을 학습하고 결과를 예측**하는 알고리즘 기법을 통칭한다.
복잡한 조건과 규칙들이 시시각각 변하면서, 매번 소프트웨어 코드를 작성해 이를 관통하는 일정한 패턴을 찾기 어려운 경우에 좋은 해결책이 된다.
머신러닝 알고리즘은 데이터를 기반으로 통계적 신뢰도를 강화하고, 예측 오류를 최소화하기 위한 여러 수학적 기법을 적용해 데이터 내 패턴을 스스로 읽어 내고, 신뢰도 있는 예측 결과를 도출한다. 데이터 분석 영역은 이러한 머신러닝 기반의 예측 분석으로 재편되어 가고 있다. (새로운 예측 모델을 개발해 더 정확한 예측 및 의사 결정을 시도)

2) 머신러닝의 분류

- **지도학습 Supervised Learning**
  - 분류

    - 회귀
    - 추천 시스템
    - 시각/음성 감지, 인지
    - 텍스트 분석, NLP(자연어 처리)
- **비지도학습 Un-supervised Learning**
  - 클러스터링
  - 차원 축소
  - 강화학습 Reinforcement

3) 데이터 전쟁

머신러닝은 데이터에 의존한다. 좋은 품질의 데이터를 이용해 학습해야 머신러닝의 수행 결과도 우수하게 도출된다. 머신러닝 알고리즘과 모델 파라미터를 잘 구축하는 것도 중요하지만, 데이터를 잘 이해하고 가공, 처리, 추출해 최적의 상태로 만드는 능력 역시 아주 중요하다.

4) 파이썬과 R 기반의 머신러닝 비교

머신러닝 프로그램 작성에 사용되는 가장 대표적인 오픈소스 프로그래밍 언어는 **Python과 R**이다.
R: 통계 분석에 특화된 프로그래밍 언어. 다양한 통계 패키지 보유.
Python: 높은 생산성. 다양한 라이브러리 가짐. Interpreter Language - 확장성, 유연성, 호환성 높아 다양한 영역에서 사용됨.
딥러닝 프레임워크(텐서플로, 케라스, 파이토치 등)에서도 파이썬 우선 정책 시행.

2. 파이썬 머신러닝 생태계를 구성하는 주요 패키지

1. 머신러닝 패키지: 사이킷런(Scikit-Learn)
    - 데이터 마이닝 기반의 머신러닝에서 독보적 위치 차지
2. 행렬/선형대수/통계 패키지: Numpy
3. 데이터 핸들링: Pandas
    - 행렬 기반의 데이터 처리에 특화된 넘파이를 보완할 수 있음
    - 2차원 데이터 처리에 특화되어 편리하게 데이터 처리 가능
4. 시각화: 맷플롯립
    - 세분화된 API로 익히기 어렵고 디자인 투박함
    - 대안: Seaborn (맷플롯립 기반, 더 함축적인 API)

넘파이와 판다스의 기본 프레임워크와 주요 API를 학습하고, 머신러닝 예제를 통해 어떻게 이들이 데이터 가공에 사용하는지를 직접 체득하는 방식이 좋다!

3. Numpy

넘파이는 Numerial Python을 의미하며, 파이썬에서 **선형대수**를 기반으로 프로그램을 쉽게 만들 수 있게 지원하는 패키지이다. 루프 없이 **대량 데이터의 배열 연산**을 가능하게 하여 배열 연산 속도가 빠르고, 이에 파이썬 기반의 많은 과학, 공학 패키지가 넘파이에 의존한다. 많은 머신러닝 알고리즘이 넘파이를 기반으로 작성되어 있고, 알고리즘의 입출력 데이터를 넘파이 배열 타입으로 사용한다.

(데이터 핸들링 부분에서는 판다스의 편리성에는 미치지 못한다. 대부분의 데이터가 2차원 형태의 행렬로 이뤄져 있어 이에 대한 가공이 가능해야 하기 때문이다.)

1. Numpy 개요
2. ndarray의 데이터 타입
3. ndarray 편리하게 생성하기: arange, zeros, ones
4. ndarray의 차원과 크기 변경하기: reshape()
5. ndarray의 데이터 세트 선택하기: 인덱싱
6. 행렬의 정렬: sort(), argsort()
7. 선형대수 연산: 행렬 내적, 전치 행렬

1) numpy 개요: array()

Numpy의 기반 데이터 타입은 **ndarray**이다. 이를 이용해 넘파이에서 다차원 배열을 쉽게 생성하고 연산을 수행할 수 있다.
파이썬의 다양한 인자를 입력값으로 받아 ndarray 타입으로 변환한다. 주로 파이썬의 리스트 객체가 이 함수의 인자로 사용된다.

```
import numpy as np
array1 = np.array([1, 2, 3])
```

이렇게 ndarray로 변환을 원하는 객체를 인자로 입력하면 ndarray를 반환하며, 생성된 ndarray 배열의 크기(행과 열의 개수를 튜플의 형태로 반환)는 `shape` 변수로 알 수 있다. 이를 통해 배열의 차원도 파악 가능하다. 배열의 차원을 반환하는 변수 `ndim` 도 있다.

2) ndarray의 데이터 타입

ndarray 내의 데이터 값은 숫자, 문자열, 불 등 모두 가능하다. (숫자: int, unsigned int, float, complex 제공)
ndarray의 데이터 타입은 `dtype` 속성으로 확인할 수 있다.

이때, 서로 다른 데이터 타입을 가질 수 있는 파이썬의 리스트와는 달리 하나의 ndarray 배열 안에는 하나의 데이터 타입만 존재할 수 있다. 만약 다른 데이터 유형이 섞인 리스트를 ndarray로 변환 경우, **데이터의 크기가 더 큰** 데이터 타입으로 일괄적으로 형 변환을 적용한다.

하지만 **astype()** 메서드를 활용해 원하는 타입을 문자열로 지정하면 데이터를 절약할 수 있다.

```
# float 값이 담긴 배열의 데이터 타입을 int32로 변환
array_float = np.array([1.1, 2.1, 3.1])
array_int = array_float.astype('int32')
print(array_int, array_int.dtype)
```

위 코드의 출력 결과: [1 2 3] int32

3) ndarray를 편리하게 생성하기: arange, zeros, ones

특정 크기와 차원을 가진 ndarray를 연속값/0/1로 초기화해 쉽게 생성해야 하는 경우가 존재한다. 주로

테스트용 데이터를 만들거나 대규모 데이터를 한꺼번에 초기화해야 하는 경우에 그렇다. 이때 **arange(), zeros(), ones()** 를 이용하면 쉽게 ndarray를 생성할 수 있다.

1. arange()

- 파이썬 표준 함수 range()와 유사한 기능.
- 배열을 range()로 표현한다고 생각하면 된다.
- 가령, `array = np.arange(10)` 라고 하고 이 array를 출력하면 0부터 9까지의 연속된 숫자값으로 구성된 1차원 ndarray가 반환된다. (스타트 값도 부여할 수 있다.)

2. zeros()

- 인자로 튜플 형태의 `shape` 값을 입력하면, 모든 값을 0으로 채운 해당 shape을 가진 ndarray를 반환한다.
- 함수 인자로 dtype을 지정할 수 있는데, 지정하지 않으면 디폴트 값으로 float64형의 데이터가 채워진다.

```
zero_array = np.zeros((3,2), dtype='int32')
print(zero_array)

# 출력물
[[0 0]
 [0 0]
 [0 0]]
```

3. ones()

- zeros() 메서드와 유사하게 인자로 shape 값을 입력하면 모든 값이 1로 채워진 해당 shape의 ndarray가 반환된다.

```
one_array = np.ones((2,3))
print(one_array)

# 출력물
[[1. 1. 1.]
 [1. 1. 1.]]
```

4) ndarray의 차원과 크기 변경하기: reshape()

변환을 원하는 ndarray의 크기를 reshape() 함수의 인자로 부여하면 특정 차원 및 크기로 배열을 반환할 수 있다. 예를 들어 0~9까지의 1차원 ndarray는 2x5나 5x2 크기의 배열로 변환할 수 있다. 하지만 5x3과 같이 변경이 불가능한 사이즈를 지정하면 오류가 발생한다.

reshape()을 효율적으로 사용하는 방법은 인자에 -1을 넣는 것이다. -1를 적용하는 경우, 기존의 ndarray와 호환되는 shape로 자동 변환된다. 예를 들어, arange(10)으로 정의된 1차원 배열 array1가 있다.

```
array2 = array1.reshape(-1, 5)
array3 = array1.reshape(5, -1)
```

위와 같이 array1에 reshape 메소드를 적용할 경우, 고정된 row(col)에 맞는 col(row)를 자동으로 생성해 변환한다. array2의 경우 2개의 로우와 5개의 컬럼, array3의 경우 5개의 로우와 2개의 컬럼으로 구성된 배열이 반환된다. 이때, -1을 사용하더라도 호환될 수 없는 형태는 에러가 발생한다는 점을 유의해야 한다.

-1은 `reshape(1,1)` 과 같은 형태로도 자주 사용된다. 이는 원본 ndarray가 어떤 형태이든 2차원이고, 반드시

1개의 컬럼을 가진(로우의 개수는 무관) ndarray로 변환되게끔 한다. 앞선 예시에서 -1을 인자에 넣으면 배열과 호환되는 shape으로 자동 변환했듯이, 컬럼 개수가 1일 때 로우의 개수를 추론하여 변환하는 것이다. (원래 배열의 총 요소 수를 1로 나눈 값으로 결정) 이를 이용해 3차원 > 2차원, 1차원 > 2차원... 으로 변환할 수 있다.

5) 넘파이의 ndarray의 데이터 세트 선택하기: 인덱싱

인덱싱: ndarray 내의 일부 데이터 세트나 특정 데이터만을 선택할 수 있도록 하는 작업

1. 특정 데이터만 추출: 희망 위치의 인덱스 값을 지정하면 그 위치의 데이터가 반환
2. 슬라이싱: 연속된 인덱스상의 ndarray를 추출함. : 기호 사이에 시작/종료 인덱스를 표시하면 시작~(종료-1) 인덱스 위치에 있는 데이터의 ndarray를 반환
3. 팬시 인덱싱: 일정한 인덱싱 집합을 리스트나 ndarray 형태로 지정해 그 위치에 있는 데이터의 ndarray 반환
4. 불린 인덱싱: 특정 조건에 해당하는지 여부인 t/f 값 인덱싱 집합을 기반으로 true에 해당하는 인덱스 위치에 있는 데이터의 ndarray 반환

단일 값 추출

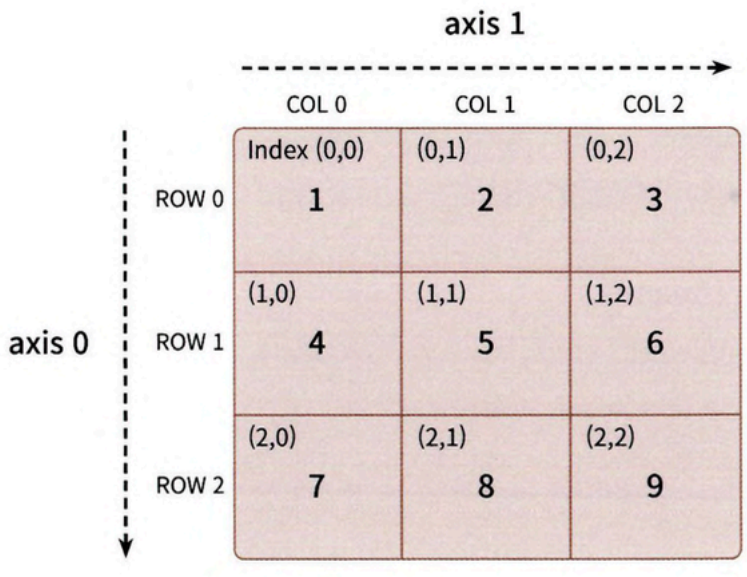
<1차원 ndarray>

- 1개의 데이터값을 추출하려면 ndarray 객체에 해당하는 위치의 인덱스 값을 [] 안에 입력하면 된다.
- 인덱스에 마이너스 기호를 사용하면 맨 뒤에서부터 데이터를 추출할 수 있다. (-1: 맨 뒤의 데이터값)
- 이때 추출된 값은 ndarray 타입이 아니라 배열 내의 데이터 값의 타입이다!
- 단일 인덱스를 이용해, 원하는 위치에 값을 대입하면 ndarray 내의 데이터값을 수정할 수 있다.

<다차원 ndarray>

- 콤마로 분리된 로우와 컬럼 위치의 인덱스를 통해 데이터에 접근한다.
- 예: array[0, 2] 라고 하면, row 0 col 2의 데이터가 출력된다.

넘파이 ndarray에서는 로우, 컬럼이 아니라 **axis 0, axis 1** 과 같은 axis 구분을 사용한다.



axis 0: row 방향의 축 (row를 따라 내려가면서(세로 방향), 열 기준으로 계산)
axis 1: col 방향의 축 (col를 따라 옆으로 가면서(가로 방향), 행 기준으로 계산)
헷갈림...

슬라이싱

- : 기호를 이용해 연속된 데이터를 슬라이싱해서 추출할 수 있다.
- 단일 값 추출과는 달리 이때 추출된 데이터 세트의 타입은 모두 ndarray이다.
- : 사이에 시작 인덱스와 종료 인덱스를 표시 > 시작~(종료-1) 위치에 있는 ndarray를 반환한다.
- 이때 시작/종료 인덱스는 생략 가능하다. 시작 인덱스 생략 시 자동으로 0 인덱스, 종료 인덱스 생략 시 마지막 인덱스, 둘 다 생략하면 맨 처음/맨 마지막 인덱스로 간주한다.
- **2차원 ndarray**에서 슬라이싱으로 데이터에 접근하는 것은 1차원에서의 것과 유사한데, 콤마를 이용해 로우와 컬럼 인덱스를 지칭해야 한다. (로우와 컬럼 각각의 인덱스에 슬라이싱을 적용하는 것이다.) 인덱스 생략하는 것도 1차원에서의 방식과 동일하다.
- 2차원 ndarray에서 슬라이싱 할 때, 뒤의 인덱스를 제거하면 1차원 ndarray를 반환한다.
  array2d[0]처럼 2차원 배열에서 로우 축의 인덱스만 표시할 경우, 첫 번째 로우 ndarray을 반환한다.
  이때 반환되는 것의 데이터 타입은 1차원 ndarray이며, 이는 shape 변수로 확인 가능하다.

팬시 인덱싱

- 리스트/ndarray로 인덱스 '집합'을 지정하면 해당하는 위치의 인덱스에 해당하는 ndarray를 반환하는 인덱싱 방식

```
# fancy indexing
array1 = np.arange(start=1, stop=10)
array2d = array1.reshape(3, 3)

array3 = array2d[[0,1], 2]
print('array2d[[0,1]],2] =>', array3.tolist())

array4 = array2d[[0,1], 0:2]
print('array2d[[0,1]], 0:2] =>', array4.tolist())

array5 = array2d[[0,1]]
print('array2d[[0,1]] =>', array5.tolist())
```

```
array2d[[0,1]],2] => [3, 6]
array2d[[0,1]], 0:2] => [[1, 2], [4, 5]]
array2d[[0,1]] => [[1, 2, 3], [4, 5, 6]]
```

헷갈리는 개념!

array3

array2d[[0,1], 2] 로우 축에 팬시 인덱싱인 [0,1]을, 칼럼 축에는 단일 값 인덱싱 2를 적용. (row, col) 인덱스가 (0,2), (1,2)로 적용되어 (3,6)을 반환한다.

array4

array2d[[0,1], 0:2]는 ((0,0), (0,1)), ((1,0),(1,1)) 인덱싱이 적용되어 [[1,2], [4,5]]를 반환한다.

array5

((0, :), (1,:)) 인덱싱이 적용되어 [[1,2,3], [4,5,6]]을 반환한다.

헷갈리기 때문에 코드와 결과물을 함께 보는 게 좋을 것 같다...

불린 인덱싱

- 자주 사용되는 인덱싱 방식: 조건 필터링 & 검색 동시 가능
- ndarray의 인덱스를 지정하는 [] 내에 **조건문**을 기재하면 된다.

내부 작동 과정: 조건문에 따라, 조건을 충족하는 값에 true, 아닌 값에 false를 매긴다. true와 false로 이루어진 **ndarray 객체**가 반환되고, 이렇게 반환된 객체를 조건문의 [] 안에 넣으면 자동으로 false는 무시하고 true값을 가지는 위치 인덱스 값으로 자동 변환해 조건을 충족하는 데이터만 반환한다.

6) 행렬의 정렬: sort(), argsort()

Numpy에서 행렬을 정렬하는 대표적인 방법: np.sort(), ndarray.sort()
정렬된 행렬의 인덱스 반환: argsort()

1. 행렬 정렬

넘파이에서 sort()를 호출하는 `np.sort()` 와 행렬 자체에서 sort()를 호출하는 `ndarray.sort()` 두 가지 방식이 존재한다.
np.sort(): 원 행렬을 유지한 채, 원 행렬의 정렬된 행렬을 반환함. 인자로 정렬하고자 하는 행렬을 삽입.
ndarray.sort(): 원 행렬 자체를 정렬된 형태로 변환, 반환 값은 none. 'ndarray' 위치에 행렬의 이름을 넣는다.
기본적으로 정렬은 오름차순으로 진행된다. (내림차순: [::-1] 적용 - np.sort()[::-1])

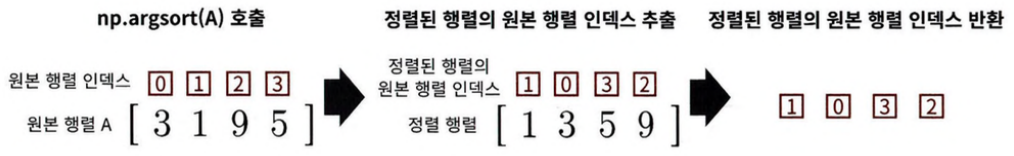
```
# 행렬 정렬
org_array = np.array([3,1,9,5])
print('원본 행렬:', org_array)

# np.sort()로 정렬
sort_array1 = np.sort(org_array)
print('np.sort() 호출 후 반환된 정렬 행렬:', sort_array1)
print('np.sort() 호출 후 반환된 원본 행렬:', org_array) # 원본은 유지된다

# ndarray.sort()로 정렬
sort_array2 = org_array.sort()
print('ndarray.sort() 호출 후 반환된 정렬 행렬:', sort_array2)
print('ndarray.sort() 호출 후 반환된 원본 행렬:', org_array)
```

```
원본 행렬: [3 1 9 5]
np.sort() 호출 후 반환된 정렬 행렬: [1 3 5 9]
np.sort() 호출 후 반환된 원본 행렬: [3 1 9 5]
ndarray.sort() 호출 후 반환된 정렬 행렬: None
ndarray.sort() 호출 후 반환된 원본 행렬: [1 3 5 9]
```

행렬이 2차원 이상인 경우, `np.sort(array2d, axis=0)` 와 같이 axis 값을 주면 로우/컬럼 기준으로 정렬 수행이 가능하다.

2. 정렬된 행렬의 인덱스를 반환하기

argsort()는 Numpy에서 활용도가 높다. 원본 행렬이 정렬되고, **기존 원본 행렬의 원소에 대한 인덱스**가 필요한 경우, `np.argsort()` 를 이용하면, 원본 행렬의 인덱스가 ndarray 형으로 반환된다. 작동 방식은 아래 사진과 같다.



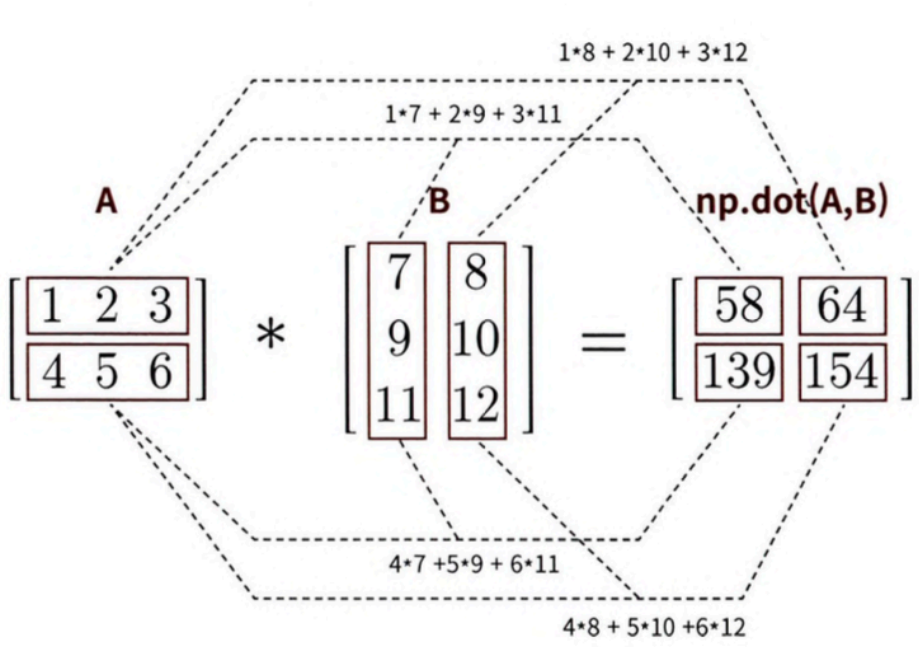
7) 선형대수 연산: 행렬 내적과 전치 행렬 구하기

넘파이는 다양한 선형대수 연산을 지원하는데, 가장 기본 연산은 **행렬 내적**과 **전치 행렬**을 구하는 것이다.

행렬 내적(행렬 곱)

두 행렬 A, B의 내적은 np.dot(행렬명, 행렬명)을 이용해서 계산 가능하다.
가령, (2,3)의 행렬 A와 (3,2)의 행렬 B가 있을 때, 이 두 행렬의 내적은 A의 로우와 B의 컬럼을 순서대로 곱한 뒤 그 값을 모두 합함으로써 구할 수 있다.

```
# 선형대수 연산: 행렬 내적

A = np.array([[1,2,3],
              [4,5,6]])
B = np.array([[7,8],
              [9,10],
              [11, 12]])
dot_product = np.dot(A, B)
print('행렬 내적 결과:\n', dot_product)
```



행렬의 내적을 구하는 방법은 위의 이미지와 같다.

전치 행렬

원본 행렬에서 행렬 위치를 교환한 원소로 구성한 행렬을 전치 행렬이라고 한다. 이 전치 행렬은 `np.transpose(행렬)` 메서드를 이용해 간단하게 구할 수 있다.

4. 데이터 핸들링: Pandas

판다스는 파이썬에서 데이터 처리를 위해 존재하는 가장 인기 있는 라이브러리로, 대부분의 데이터 세트가 2차원 데이터이다.(행x열) 판다스는 2차원 데이터를 효율적으로 가공, 처리할 수 있는 여러 기능을 제공한다. 파이썬의 리스트, 컬렉션, 넘파이 등의 내부 데이터뿐 아니라 CSV, tab과 같은 유형의 '분리 문자로 컬럼을 분리한' 파일도 쉽게 DataFrame으로 변경해 편리하게 가공/분석할 수 있게 한다.

판다스의 핵심 객체: **DataFrame**

- 여러 개의 행과 열로 이루어진 2차원 데이터를 담는 데이터 구조체

Index

- 개별 데이터를 고유하게 식별하는 key 값 (Primary Key와 같은 개념)
- Series, DataFrame은 모두 이 Index를 key 값으로 가진다.

Series

- 컬럼이 하나뿐인 데이터 구조체 (DataFrame과의 차이점)
- DataFrame은 여러 개의 Series로 이루어져 있다.

1. 판다스 시작하기: 파일 DF로 로딩, 기본 API

우선 import pandas as pd로 판다스를 임포트한다. 다음으로 캐글의 '타이타닉 탑승자' 파일을 다운받아 이를 DF로 로딩할 것이다. (https://www.kaggle.com/c/titanic/data 에서 train.csv 다운로드)
csv 파일은 콤마로 각각의 필드를 분리하는 포맷이다. 이러한 포맷의 파일은 `read_csv()` 메서드로 파일을 변환한다. 꼭 csv가 아니더라도 필드 구분 문자 기반의 파일 포맷도 이 메서드로 변환 가능하다. (탭 문자로 필드를 구분하는 경우, read_csv('파일명', sep='\t) 처럼 쓰면 됨.)

그 외에도 read_table(), read_fwf()와 같은 API가 있고, 이를 활용해 다양한 포맷의 파일을 DataFrame으로 로딩할 수 있다.

read_csv(filepath, sep='', ',...) 메서드에서 가장 중요한 인자는 **filepath**로, 나머지 인자는 생략하면 자동으로 디폴트 값을 할당한다.
filepath에 로드하려는 데이터 파일의 경로를 포함한 파일명을 입력하면 된다.

```
titanic_df = pd.read_csv(r'/Users/hyun/Dev/titanic_train.csv')
```

참고로 구글의 코랩에서 데이터 파일을 업로드하는 경우에는 파일이 임시저장되어 런타임 돌릴 때마다 업로드해야 해서 번거롭다... 주피터 노트북이나 vsc가 좋을 것 같다.

pd.read_csv() 는 호출 시 파일명의 인자로 들어온 파일을 로딩해 DF 객체로 반환한다.

- 일반적으로 파일의 첫 번째 row를 컬럼명으로 인지하고 컬럼으로 변환함 (파라미터 지정이 없을 경우)
- 콤마로 분리된 데이터값들을 해당 컬럼에 맞게 할당
- 파일에 포함되지 않은 데이터값이 로우 순서로 0부터 표시되는데, 이는 판다스의 `Index` 객체 값 > 모든 DF 내의 테이터는 생성되는 순간 고유한 인덱스 값을 가진다.

Shape & Metadata
1. 넘파이와 유사하게 shape 변수를 통해 DF의 크기를 알 수 있다. 행과 열의 개수를 튜플의 형태로 반환한다.

2. 컬럼의 타입, Null 데이터 개수, 데이터 분포도 등의 메타데이터도 조회 가능하다.

- info() : 총 데이터 건수, 데이터 타입, Null 건수
- describe() : 컬럼별 숫자형 데이터의 avg, min, max, n-percentile 분포도. 데이터의 분포도를 개략적으로 확인할 수 있어서 유용하다.

```
[13]: # 메타데이터 조회
titanic_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

```
[14]: # 컬럼별로 숫자형 데이터값의 n-percentile 분포, avg, min, max...
titanic_df.describe()
```

[14]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

- value_counts(): 지정된 컬럼의 데이터값 건수 반환. 데이터 분포도 확인에 유용하다. 많은 건수 순서로 정렬해 값을 반환한다.

Series : 앞서 설명한 인덱스와 단 하나의 컬럼으로 구성된 데이터 세트.

- 모든 시리즈 객체는 인덱스를 반드시 가진다. 이때, 인덱스는 꼭 0부터 시작하지 않아도 된다. 고유성을 가진다는 전제하에, 고유 컬럼 값을 식별자로 사용할 수 있다. (시리즈가 단일 컬럼으로 구성되기 때문)
- Series 객체에서 value_counts() 메서드를 호출하면 컬럼별 데이터 값의 분포도를 더 명시적으로 파악할 수 있다. 이때, 메서드가 반환하는 데이터의 타입 역시 Series 객체이다.

2. DF ↔ 리스트, 딕셔너리, ndarray

1) ndarray, 리스트, 딕셔너리를 DF로 변환하기

리스트와 ndarray보다 DF의 데이터 핸들링이 비교적 편한 이유는 '컬럼명'을 가지고 있기 때문이다.
일반적으로 DF는 변환 시 컬럼명을 지정하고, 지정하지 않는 경우 자동으로 할당된다.

pd.DataFrame(data, column) 에서 data 부분에는 리스트/딕셔너리/ndarray를 입력받고, column에는

컬럼명 리스트를 입력받아 DF을 생성할 수 있다.
DF은 행과 열을 가지는 2차원 데이터로, 2차원 이하의 데이터들만 변환될 수 있다.

```
col_name1 = ['col1'] # 컬럼 이름
list1 = [1, 2, 3]
array1 = np.array(list1) # ndarray 생성

# list 이용해 DF 생성
df_list1 = pd.DataFrame(list1, columns=col_name1) # 컬럼명 리스트를 입력받음
print('1차원 리스트로 만든 DataFrame:\n', df_list1)
print('\n')

# ndarray를 이용해 DF 생성
df_array1 = pd.DataFrame(array1, columns=col_name1)
print('1차원 ndarray로 만든 DataFrame:\n', df_array1)


# 출력 결과:
1차원 리스트로 만든 DataFrame:
    col1
0     1
1     2
2     3


1차원 ndarray로 만든 DataFrame:
    col1
0     1
1     2
2     3
```

2차원 형태의 배열을 DF로 변환할 경우에는 column의 개수에 맞추어 컬럼명을 지정해 주어야 한다. DF로 변환하는 메서드는 1차원에서와 동일하고, 컬럼 개수에 맞춰 지정한 변수를 `columns='변수명` 형태로 인자에 넣으면 된다.

딕셔너리를 DF으로 변환할 때는, 딕셔너리의 [Key-컬럼명, Value-키에 해당하는 컬럼 데이터] 로 매핑된다.
Key는 문자열, Value는 리스트/ndarray의 형태로 딕셔너리가 구성된다.

2) DF를 ndarray, 리스트, 딕셔너리로 변환하기

많은 머신러닝 패키지가 기본 데이터 타입으로 ndarray를 사용하기 때문에 DF를 이용하더라도 다시 ndarray로 변환해야 하는 경우가 많다.
이는 DF 객체의 values를 활용해 쉽게 가능하다.

- DF → list: .tolist()
- DF → dict: .to_dict() 이때 인자로 'list'를 넣으면 딕셔너리 값이 리스트형으로 반환된다.

```
# dictionary 정의 후 DF로 변환
dict = {'col1':[1, 11], 'col2':[2, 22], 'col3':[3, 33]}
df_dict = pd.DataFrame(dict)

# DF를 ndarray로 변환: values 활용
array = df_dict.values

# DF를 리스트로 변환
list = df_dict.values.tolist()

# DF를 딕셔너리로 변환
dict = df_dict.values.to_dict('list') # 값이 리스트형으로 반환
dict2 = df_dict.values.to_dict() # 딕셔너리 형태로...
```

3. DF의 컬럼 데이터 세트 생성과 수정

컬럼 데이터 세트 생성 및 수정은 [] 연산자로 가능하다.
`DataFrame명['컬럼명'] = 값` 와 같이 대괄호 연산자 내에 생성하고자 하는 컬럼의 이름을 입력하고 값을 할당하면 된다. 예를 들어, titanic_df라는 데이터프레임에 'Age_0'라는 이름의 컬럼을 추가하고, 일괄적으로 0이라는 값을 할당한다면 titanic_df['Age_0']=0 라고 코드를 작성하면 된다.
기존에 존재하는 Series의 데이터를 이용해 새로운 컬럼 Series를 생성할 수도 있다.
`DataFrame명['컬럼명'] = DataFrame명['컬럼명']+업데이트하려는 값` 으로 하면 된다. (연산기호는 필요한 대로 알아서...)

4. DataFrame 삭제

데이터의 삭제는 `drop()` 메서드를 활용한다. 이 메서드의 원형은 다음과 같다.

```
DataFrame.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, e
```

여기서 중요한 파라미터는 **labels, axis, inplace**이다.

1. **axis** : DataFrame의 로우를 삭제할 때는 axis=0, 칼럼을 삭제할 때는 axis=1으로 설정한다.
   일반적으로는 컬럼 삭제를 수행한다. (axis=1)
2. 원본 DataFrame은 유지하고 드롭된 DataFrame을 새롭게 객체 변수로 받고 싶다면, inplace=False로 설정 (디폴트 값: false)
   예: titanic_drop_df = titanic_df.drop('Age_0', axis=1, inplace=False)
3. 원본 DataFrame에 드롭된 결과를 적용할 경우에는 inplace=True를 적용
   예: titanic_df.drop('Age_0', axis=1, inplace=True)
4. 원본 DataFrame에서 드롭된 DataFrame을 다시 원본 DataFrame 객체 변수로 할당하면 원본 DataFrame에서 드롭된 결과를 적용할 경우와 같음 (단, 기존 원본 DataFrame 객체 변수는 메모리에서 추후 제거됨)
   예: titanic_df = titanic_df.drop('Age_0', axis=1, inplace=False)

5. Index 객체

판다스의 index 객체: Primary Key(RDBMS)와 유사하게 DF, Series의 레코드를 고유하게 식별하는 객체이다. 인덱스 객체만 추출하려면 DataFrame.index 또는 Series.index 속성을 이용하면 된다.

[Index 객체의 특징]

- 1차원 배열의 형태로 식별성 데이터를 가진다.
- ndarray와 유사하게 단일 값 추출/반환 및 슬라이싱이 가능하다.
- 한번 생성된 DF/Series의 인덱스 객체는 사용자가 임의로 변경할 수 없다.
- Series 객체에 연산 함수를 적용할 때, 인덱스는 연산에 포함되지 않는다. 오직 식별용!
- Series에 `reset_index()` 메서드를 수행하면 인덱스를 새로운 연속된 숫자형으로 할당하고, 기존의 인덱스는 `index` 라는 새로운 컬럼명으로 추가되며 DF으로 변환된다. 이는 주로 인덱스가 연속된 int 숫자형 데이터가 아닐 경우에 이를 연속되게 만들기 위해 사용한다. (깃허브의 코드 참고)

6. 데이터 셀렉션 및 필터링

넘파이에서 [] 연산자 내 단일 값 추출, 슬라이싱, 팬시 인덱싱, 불린 인덱싱을 사용해 데이터를 추출했다.
판다스에서는 동일한 작업을 **iloc[], loc[]** 연산자를 통해 수행한다.

1. [] 연산자

- 컬럼명 문자(or 컬럼명 리스트 객체) 또는 인덱스로 변환 가능한 표현식
- 컬럼만 지정할 수 있는 컬럼 지정 연산자로 이해하는 게 쉽다.

2. iloc[]과 loc[] 연산자

- iloc[]: 위치 기반 인덱싱 방식으로 동작
  - 행과 열 위치를 0을 출발점으로 하는 세로축, 가로축 좌표 정수 값으로 지정하는 방식
    - 행과 열의 좌표 위치에 해당하는 값으로 정수/정수형의 슬라이싱/팬시 리스트 값을 입력해야 함
- loc[]: 명칭(label) 기반 인덱싱
  - 데이터 프레임의 인덱스 값으로 '행 위치'를, 컬럼의 명칭으로 '열 위치'를 지정하는 방식

주의할 점

1. 개별 또는 여러 칼럼 값 전체를 추출하고자 한다면 iloc[ ]나 loc[ ]를 사용하지 않고 data_df['Name']과 같이 DataFrame['칼럼명']만으로 충분합니다. 하지만 행과 열을 함께 사용하여 데이터를 추출해야 한다면 iloc[ ]나 loc[ ]를 사용해야 합니다.
2. iloc[ ]와 loc[ ]를 이해하기 위해서는 명칭 기반 인덱싱과 위치 기반 인덱싱의 차이를 먼저 이해해야 합니다. DataFrame의 인덱스나 칼럼명으로 데이터에 접근하는 것은 명칭 기반 인덱싱입니다. 0부터 시작하는 행, 열의 위치 좌표에만 의존하는 것이 위치 기반 인덱싱입니다.
3. iloc[ ]는 위치 기반 인덱싱만 가능합니다. 따라서 행과 열 위치 값으로 정수형 값을 지정해 원하는 데이터를 반환합니다.
4. loc[ ]는 명칭 기반 인덱싱만 가능합니다. 따라서 행 위치에 DataFrame 인덱스가 오며, 열 위치에는 칼럼명을 지정해 원하는 데이터를 반환합니다.
5. 명칭 기반 인덱싱에서 슬라이싱을 '시작점:종료점'으로 지정할 때 시작점에서 종료점을 포함한 위치에 있는 데이터를 반환합니다.

3. 불린 인덱싱

넘파이에서의 불린 인덱싱과 마찬가지로, 판다스에서도 [] 연산자 내에 불린 조건을 입력하면 조건에 맞는 결과값을 필터링할 수 있다.
위의 두 가지 방식처럼 명확히 인덱싱을 지정하는 것보다 불린 인덱싱에 의존해 데이터를 가져오는 경우가 더 많다. iloc[]은 정수형 값만 지원하기 때문에 불린 인덱싱이 지원되지 않지만, [], loc[]에서는 불린 인덱싱이 공통으로 지원된다.

여러 개의 **복합 조건**도 결합하여 적용 가능하다.
1. and: &
2. or: |
3. Not: ~

7. 정렬, Aggregation 함수, GroupBy 적용

DF, Series의 정렬: sort_values()

sort_values() 메서드를 활용해 데이터프레임과 시리즈를 정렬할 수 있다

- 주요 입력 파라미터: by, ascending, inplace
  - by: 특정 컬럼을 입력하면 해당 컬럼으로 정렬 수행
    - ascending: =True로 설정하면 오름차순 정렬, =False로 설정하면 내림차순 정렬 (default: ascending)
    - inplace: False로 설정하면 sort_values()를 호출한 데이터프레임은 그대로 유지하되 정렬된 데이터프레임 반환 / True로 설정하면 호출한 데이터프레임의 정렬 결과를 적용 (default: False)

```
titanic_sorted = titanic_df.sort_values(by=['Name'])
```

라고 하면, Name 컬럼에 대해 오름차순으로 정렬된다.
['Name', 'Age']와 같이 리스트 형식으로 컬럼들을 입력하면 여러 개의 컬럼으로도 정렬 가능하다.

Aggregation 함수 적용

RDBMS의 aggregation 함수와 유사하게, **min(), max(), sum(), count()** 와 같은 작업을 데이터프레임에 대해서도 수행할 수 있다.
단, DF에서 aggr.를 바로 호출할 경우, '모든 컬럼'에 해당 aggr.를 적용한다. 특정 컬럼에만 적용하고 싶다면 대상 컬럼을 추출한 뒤 작업을 수행하면 된다.

Aggregation 함수 적용: min, max, sum, count

```
# DF에서 바로 count()를 호출하는 경우: 모든 컬럼에 대해 결과를 반환한다.
titanic_df.count()
```

```
PassengerId    891
Survived       891
Pclass         891
Name           891
Sex            891
Age            714
SibSp          891
Parch          891
Ticket         891
Fare           891
Cabin          204
Embarked       889
dtype: int64
```

```
# 특정 컬럼에만 agg. 함수를 적용하고 싶다면, 대상 컬럼을 추출해 적용하면 된다.
titanic_df[['Age', 'Fare']].mean()
```

```
Age     29.699118
Fare    32.204208
dtype: float64
```

groupby() 적용

groupby()는 판다스에서 SQL의 group by와 유사한 기능을 제공하는 메서드이다.

- DataFrame에서 특정 컬럼을 기준으로 그룹을 묶어 **DataFrameGroupBy 객체** 를 반환한다.
- 반환된 객체에 집계 함수(aggregation function) 를 적용하여 그룹별 통계나 연산을 수행할 수 있다.

8. 결손 데이터 처리하기: isna(), fillna()

결손 데이터: 컬럼에 값이 없는, 즉 `null` 인 경우 - 넘파이의 NaN으로 표시됨.
ML 알고리즘은 이 NaN 값을 처리하지 않으며, 함수 연산(min, sum 등)에도 포함되지 않기 때문에 다른 값으로 대체되어야 한다.

1. isna(): 결손 데이터 여부(NaN 여부)를 확인하는 API - True / False

- 결손 데이터의 개수를 isna().sum() 처럼 결과값에 sum() 함수를 추가함으로써 구할 수 있음.

2. fillna(): 결손 데이터를 다른 값으로 대체하는 API

- 가령, Name 컬럼의 결손 데이터에 일괄적으로 'A'이라는 값을 부여하고 싶다면, titanic_df['Name'] = titanic_df['Name'].fillna('A') 와 같이 하면 된다.

9. apply lambda 식으로 데이터 가공

apply 함수에 lambda 식을 결합해 DF/Series의 **레코드별**로 데이터를 가공할 수 있다. (일괄적으로 데이터를 가공하는 것이 속도 면에서 좋지만, 복잡한 가공이 필요한 경우에는 이를 활용한다.)
lambda 식: 파이썬에서 함수형 프로그래밍을 지원하기 위해 만들어짐.
사용 방법은 파이썬에서와 크게 다른 것이 없다. 예시 코드 참고!

```
# pandas의 lambda
# Name 컬럼의 문자열 개수를 별도 컬럼 Name_len에 생성

titanic_df['Name_len'] = titanic_df['Name'].apply(lambda x:len(x))
titanic_df[['Name', 'Name_len']].head(3)


# Age 값에 따라 child, adult, elderly로 분류하는 Age_Cat 컬럼 생성
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x : 'Child' if x<= 15 else ('Adult'
                                                                                   else 'E
titanic_df['Age_cat'].value_counts()
```