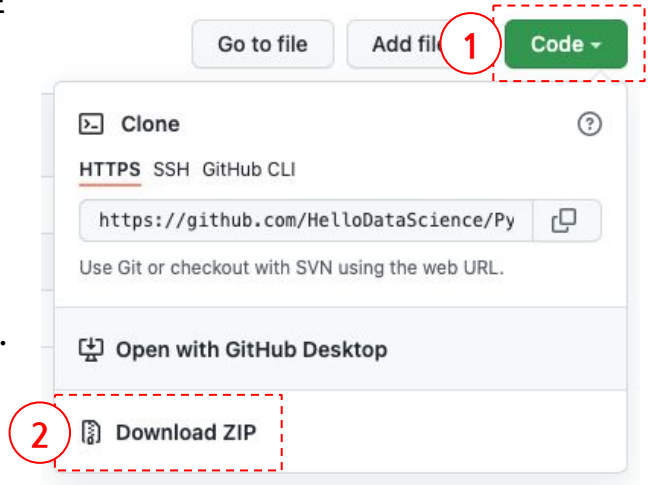
Python Basic Programming

강의 내용

- Python 변수와 객체
- Python 자료형
  - 숫자(정수, 실수, 문자열, 부울)
- Python 자료구조
  - 리스트, 튜플, 딕셔너리, 집합
- Python 제어문
- Python 함수
- Python 표준 라이브러리
- numpy 라이브러리
- pandas 라이브러리
- 데이터 입출력
- 데이터프레임 전처리
- 데이터 시각화

실습 데이터셋 내려받기

- 크롬 브라우저를 열고, 아래 링크로 접속합니다.
  - 깃허브 저장소: **https://github.com/HelloDataScience/PythonBasicProgramming**
- ① Code, ② Download ZIP 버튼을 차례대로 클릭하면 다운로드Downloads 폴더에 zip 파일이 저장됩니다.
- zip 파일을 문서Documents 폴더로 옮기고 압축을 풀면 code와 data 폴더가 포함되어 있습니다.
  - code 폴더에 주피터 노트북 파일이 저장되어 있습니다.
  - data 폴더에 실습 데이터 파일이 저장되어 있습니다.



Python 변수와 객체

Python 변수variable

- Python 프로그래밍을 할 때, '변수'라는 개념을 알아야 합니다.
  - 변수는 어떤 값을 가리키는 것이며, 가리키는 값은 매번 바뀔 수 있습니다.
  - 만약 변수를 사용하지 않는다면 바뀌는 값을 매번 코딩해야 하므로 불편합니다.
  - 하지만 변수를 사용하면 코드를 재사용할 수 있으므로 상당히 편리해 집니다.
- 변수의 이름(변수명)은 알파벳, 숫자 및 밑줄underscore을 조합하여 만듭니다.
  - 변수명은 숫자로 시작할 수 없고, 알파벳 또는 밑줄로 시작할 수 있습니다.
  - 변수명은 알파벳 대소문자를 구분합니다.
  - 변수명을 한글로 작성할 수 있지만, 추천하지 않습니다.

[참고] 변수명으로 사용할 수 없는 예약어

- 변수명을 만들 때 예약어를 사용할 수 없습니다.
- Python 예약어를 확인하는 방법입니다.

```
>>> help('keywords')
```

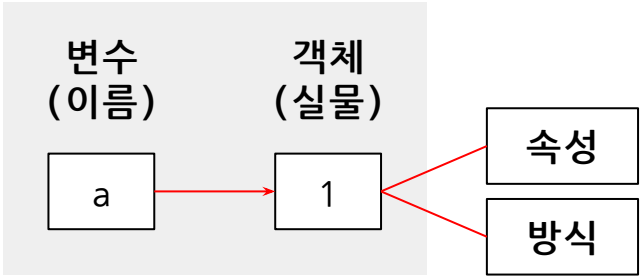
- 아울러 내장 함수명 또는 모듈명은 피하는 것이 좋습니다.
- Python 내장 함수명을 확인하는 방법입니다.

```
>>> dir(__builtin__)
```

Python 객체object

- Python에서 변수에 어떤 값을 할당하면, 그 값의 자료형에 따라 객체가 생성되고 객체는 메모리에 할당되며 변수는 객체가 할당된 메모리 주소를 저장합니다.
  - 변수는 객체가 할당된 메모리 주소로 객체와 연결되며, 객체가 바뀌면 주소도 바뀝니다.
- 주피터 노트북에서 아래 코드를 실행하여 직접 확인해보겠습니다.

```
>>> a = 1  # a에 정수 1을 할당합니다. 정수 1은 객체가 되어 메모리에 할당됩니다.
           # 등호(=)는 할당 연산자입니다.
>>> id(a)  # a가 가리키는 메모리 주소를 출력합니다.
>>> a = 2  # a에 정수 2를 할당합니다.
>>> id(a)  # a가 가리키는 메모리 주소가 바뀌었습니다.
```



Python 객체의 속성과 방식

- 객체는 클래스class를 통해 생성됩니다.
  - 클래스는 데이터를 다룰 때 필요한 여러 속성attribute과 방식method을 정의한 것입니다.
  - 클래스를 통해 생성된 객체를 인스턴스instance 객체라고 합니다.
  - (필요시) 외부 클래스를 생성할 수 있지만, 이 강의에서는 다루지 않습니다.
    - 사전에 정의된 클래스로 객체를 생성하고, 객체의 속성과 방식을 활용하는 방법을 소개합니다.
- 객체에 할당되는 값에 따라 객체의 클래스가 결정됩니다.
  - **type**() 함수는 변수가 가리키는 객체가 어떤 클래스를 통해 생성되었는지 출력합니다.
  - 같은 클래스를 통해 생성된 객체는 같은 속성과 방식을 가집니다.

Python 객체의 속성과 방식(계속)

- 변수에 값을 할당하여 객체를 생성하고, 변수의 클래스를 확인합니다.

```
>>> a = 10 # a에 정수 10을 할당합니다.

>>> a  # 변수명만 입력하고 실행하면 변수가 가리키는 객체에 저장된 값을 출력합니다.

>>> print(a) # print() 함수로 a를 출력합니다.
            # [참고] print() 함수를 실행한 결과는 변수명만 입력할 때와 다를 수 있습니다.

>>> type(a) # a의 클래스를 확인합니다. int 클래스가 사용되었습니다.
```

- a가 갖는 속성과 방식을 출력하려면 **dir**() 함수를 이용합니다.

```
>>> dir(a) # a에는 많은 속성과 방식을 포함되어 있지만, 알파벳으로 시작하는 것만 활용하겠습니다.
           # 밑줄로 시작하는 속성이나 방식은 특별한 기능을 가지는 것입니다.

# [참고] type(), print(), dir() 등은 Python의 내장 함수이며, 괄호 안에 변수를 입력하는 형태로 코드를 작성합니다.
```

Python 객체의 속성과 방식에 접근

- 객체의 속성attribute 또는 방식method에 접근하려면 변수명 뒤에 마침표(온점)를 찍고, 접근하려는 속성 또는 방식을 덧붙입니다.

```
Object.attribute
```

```
Object.method()
```

- 아래는 객체의 속성 또는 방식에 접근하는 예시 코드입니다.

```
>>> a.numerator  # denominator은 '속성'입니다. 따라서 뒤에 빈 괄호를 추가하면 아래 메시지가 출력됩니다.
                 TypeError: 'int' object is not callable
>>> a.bit_length()  # bit.length는 '방식'입니다. 따라서 뒤에 빈 괄호를 생략하면 아래 메시지가 출력됩니다.
                    <function int.bit_length()>

                    # 이 강의안에서는 객체의 방식method을 편의상 '함수'로 지칭하겠습니다.
```

Python 객체의 생성 및 교환

- a와 b에 같은 문자열을 동시에 할당합니다.

```
>>> a = b = 'Language' # a와 b에 같은 문자열을 동시에 할당합니다.
>>> print(a); print(b) # a와 b를 출력합니다.
                      # 세미콜론을 이용하면 한 줄에 여러 줄의 코드를 작성할 수 있습니다.
```

- a와 b에 다른 문자열을 각각 할당하고, 할당되었던 값을 서로 교환합니다.

```
>>> a, b = 'Python', 'R' # a와 b에 두 개의 문자열을 각각 할당합니다.
>>> print(a); print(b) # a에는 'Python', b에는 'R'이 할당되었습니다.
>>> a, b = b, a        # a와 b에 할당되었던 값을 서로 교환합니다.
                       # [참고] https://stackoverflow.com/questions/31374721/
>>> print(a); print(b) # a와 b에 할당되었던 값이 서로 바뀌었음을 확인할 수 있습니다.
```

[참고] 키보드에서의 주요 키key 위치

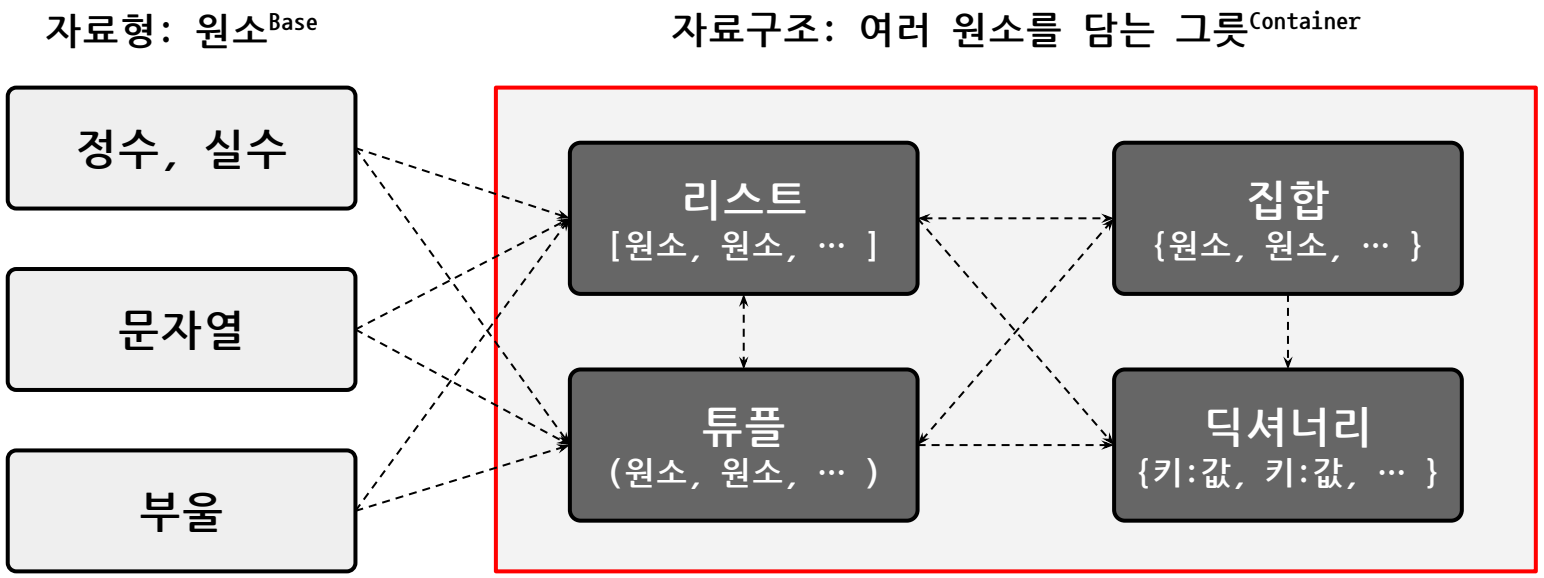
 # 틸데tilde: ~ (shift 누른 채)
 # 백틱backtick: `

 # 바bar: | (shift 누른 채)
 # 역슬래시backslash: \ 또는 ₩

 # 출처: https://m.post.naver.com/viewer/postView.nhn?volumeNo=17962736&memberNo=11534881

Python 자료형

Python 자료형과 자료구조



숫자Number: 정수, 실수

- 정수는 소수점이 없는 숫자 자료형입니다.

```
>>> a = 3 # a에 정수 3을 할당합니다.

>>> a  # a를 출력합니다.

>>> type(a) # a의 클래스를 확인합니다. a의 클래스는 int입니다.
```

- 실수는 소수점이 있는 숫자 자료형입니다.

```
>>> b = 3.0 # b에 실수 3.0을 할당합니다.

>>> b  # b를 출력합니다.

>>> type(b) # b의 클래스를 확인합니다. b의 클래스는 float입니다.
```

산술 연산자

- 산술 연산자는 두 개의 숫자에 대한 산술 연산을 실행합니다.

구분	상세 내용
+	• 왼쪽과 오른쪽 숫자를 더합니다.
-	• 왼쪽 숫자에서 오른쪽 숫자를 뺍니다.
*	• 왼쪽과 오른쪽 숫자를 곱합니다.
**	• 왼쪽 숫자를 오른쪽 숫자로 거듭제곱합니다.
/	• 왼쪽 숫자를 오른쪽 숫자로 나눕니다.
//	• 왼쪽 숫자를 오른쪽 숫자로 나눈 몫을 반환합니다.
%	• 왼쪽 숫자를 오른쪽 숫자로 나눈 나머지를 반환합니다.

산술 연산자 실습

- 숫자 변수로 산술 연산을 실행합니다.

```
>>> a + b  # a와 b를 더합니다.
        # 둘 다 정수일 때는 정수를 반환하지만, 하나라도 실수일 때는 실수를 반환합니다.
>>> a - b  # a에서 b를 뺍니다.
>>> a * b  # a와 b를 곱합니다.
>>> a ** b # a를 b로 거듭제곱합니다.
>>> a / b  # a를 b로 나눕니다.
        # 나눗셈 결과는 항상 실수로 반환됩니다.(정수 간 나눗셈도 마찬가지입니다.)
>>> a // b # a를 b로 나눈 (소수점 이하를 절사한) 몫을 반환합니다.
>>> a % b  # a를 b로 나눈 나머지를 반환합니다.
```

할당 연산자

- 할당 연산자는 변수에 값을 할당(입력)합니다.

구분	상세 내용
=	• 왼쪽 변수에 오른쪽 값을 할당합니다.
+=	• 왼쪽 변수에 오른쪽 값을 더한 결과를 왼쪽 변수에 할당합니다.
-=	• 왼쪽 변수에서 오른쪽 값을 뺀 결과를 왼쪽 변수에 할당합니다.
*=	• 왼쪽 변수에 오른쪽 값을 곱한 결과를 왼쪽 변수에 할당합니다.
**=	• 왼쪽 변수를 오른쪽 값으로 거듭제곱한 결과를 왼쪽 변수에 할당합니다.
/=	• 왼쪽 변수를 오른쪽 값으로 나눈 결과를 왼쪽 변수에 할당합니다.
//=	• 왼쪽 변수를 오른쪽 값으로 나눈 몫을 왼쪽 변수에 할당합니다.
%=	• 왼쪽 변수를 오른쪽 값으로 나눈 나머지를 왼쪽 변수에 할당합니다.

할당 연산자

- 숫자 변수로 할당 연산자를 실습합니다.

```
>>> a = 5; a  # a에 정수 5를 할당하고, a를 출력합니다.
>>> a += 1; a # a에 1을 더한 값을 a에 재할당하고, a를 출력합니다.
>>> a -= 2; a # a에서 2를 뺀 값을 a에 재할당하고, a를 출력합니다.
>>> b *= 3; b # b에 3을 곱한 값을 b에 재할당하고, a를 출력합니다.
>>> b /= 4; b # b를 4로 나눈 값을 b에 재할당하고, a를 출력합니다.
```

비교 연산자

- 비교 연산자는 두 숫자의 크기를 비교하여 True 또는 False로 반환합니다.

구분	상세 내용
>	• 왼쪽이 오른쪽보다 크면 True, 작거나 같으면 False 입니다.
>=	• 왼쪽이 오른쪽보다 크거나 같으면 True, 작으면 False 입니다.
<	• 왼쪽이 오른쪽보다 작으면 True, 크거나 같으면 False 입니다.
<=	• 왼쪽이 오른쪽보다 작거나 같으면 True, 크면 False 입니다.
==	• 왼쪽이 오른쪽과 같으면 True, 다르면 False 입니다.
!=	• 왼쪽이 오른쪽과 다르면 True, 같으면 False 입니다.

비교 연산자 실습

- 숫자 변수의 크기를 비교하는 연산을 실행합니다.

```
>>> a > 4  # a가 4보다 크면 True, 작거나 같으면 False를 반환합니다.

>>> a >= 4  # a가 4보다 크거나 같으면 True, 작으면 False를 반환합니다.

>>> a < 4  # a가 4보다 작으면 True, 크거나 같으면 False를 반환합니다.

>>> a <= 4  # a가 4보다 작거나 같으면 True, 크면 False를 반환합니다.

>>> a == 4  # a가 4이면 True, 다르면 False를 반환합니다.
            # [주의] 등호(=)가 2개 사용되었습니다.

>>> a != 4  # a가 4가 아니면 True, 같으면 False를 반환합니다.
```

논리 연산자

- 논리 연산자는 진리값Truth value 중간에 위치하여 True 또는 False로 반환합니다.

구분	상세 내용
and	• [논리곱] 양쪽 진리값 모두 True이면 True, 아니면 False를 반환합니다.
or	• [논리합] 양쪽 진리값 중 하나 이상 True이면 True, 아니면 False를 반환합니다.
not	• [논리부정] not 뒤에 오는 진리값을 반전합니다.(True ↔ False)

True	and	True	True
True		False	False
False		True	False
False		False	False

True	or	True	True
True		False	True
False		True	True
False		False	False

not	True	False
	False	True

논리 연산자 실습

- 논리곱 연산을 실행합니다.

```
>>> a > 3 and b > 3  # [논리곱] 양쪽의 진리값이 모두 True이면 True, 아니면 False를 반환합니다.
```

- 논리합 연산을 실행합니다.

```
>>> a > 3 or b > 3  # [논리합] 양쪽의 진리값 중 하나 이상 True이면 True, 아니면 False를 반환합니다.
```

- 논리 부정 연산을 실행합니다.

```
>>> not a > 3  # [논리부정] not 뒤에 오는 진리값을 반전합니다.

>>> not b > 3
```

비트 연산자

- 비트 연산자는 정수를 2진수(비트)로 연산한 결과를 반환합니다.

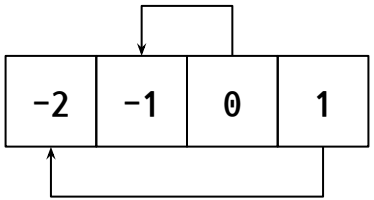
구분	상세 내용
&	• [논리곱] 비트로 표현된 두 정수에 대해 **and** 연산을 실행합니다.
¦	• [논리합] 비트로 표현된 두 정수에 대해 **or** 연산을 실행합니다.
~	• [논리부정] 반전 연산을 실행합니다.

- [참고] Python에서 엑셀 파일을 읽을 때, pandas의 데이터프레임을 생성합니다.
  - 데이터프레임을 필터링할 때, 여러 조건에 대해 논리 연산자를 사용할 수 없습니다.
  - 대신 비트 연산자를 사용하는데, 개별 조건을 반드시 소괄호로 묶어 주어야 합니다.

비트 연산자 실습

- 비트 연산자로 논리곱, 논리합, 논리부정 연산을 실행합니다.

```
>>> (a > 3) & (b > 3)

>>> (a > 3) | (b > 3)

>>> ~ (a > 3) # 소괄호 안 결과는 True인데, True는 정수 1이므로 반전하면 -2가 됩니다.

>>> ~ (b > 3) # 소괄호 안 결과는 False인데, False는 정수 0이므로 반전하면 -1이 됩니다.
```



문자열String

- 문자열은 문자, 숫자 등을 따옴표로 감싼 자료형입니다.

```
>>> str1 = 'Life is short, ' # str1에 문자열을 할당합니다.
                              문자열 양쪽으로 홑따옴표로 감싸주었습니다.
>>> str1 # str1을 출력합니다.
>>> print(str1) # print() 함수를 실행하면 출력되는 결과가 달라집니다.
>>> type(str1) # str1의 클래스를 확인합니다. str1의 클래스는 str입니다.
>>> len(str1) # str1의 글자수를 반환합니다.
>>> str2 = "you need Python!" # str2에 문자열을 할당합니다.
                               문자열 양쪽으로 겹따옴표로 감싸주었습니다.
>>> str2 # str2를 출력합니다.
```

[참고] 다양한 문자열 생성법

- 다양한 문자열 생성 방법을 소개합니다.

```
>>> 'Monty Python's Flying Circus'  # 홑따옴표로 감싼 문자열 안에 홑따옴표를 추가하면 에러가 발생합니다.
>>> "Monty Python's Flying Circus"  # 문자열 안에 홑따옴표를 추가하려면 문자열을 겹따옴표로 감싸주어야 합니다.
>>> 'Monty Python\'s Flying Circus'  # 만약 홑따옴표를 쓰고 싶다면 안쪽 따옴표 앞에 역슬래시(\) 기호를 추가해야 합니다. 정규표현식의 '이스케이프'입니다.
>>> 'I\'m saying "You need Python!" now.'  # 두 가지 경우를 모두 포함하는 예제입니다.
                                           # print() 함수는 \ 기호를 제외한 결과를 출력합니다.
>>> "I'm saying \"You need Python!\" now."  # 문자열을 겹따옴표로 감싸주었으면, 문자열 안의
                                            # 겹따옴표 앞에 \ 기호를 추가합니다.
>>> '''Hello!                  # 여러 줄의 문자열을 생성하려면 홑따옴표나 겹따옴표를 세 번 반복합니다.
                               # 예) '''문자열''', """문자열"""
    Good to see you.'''        # 왼쪽 코드를 실행하면 '!'와 'G' 사이에 '\n'이 출력됩니다.
                               # '\n'은 줄바꿈을 의미합니다.
```

문자열 연산자

- + 연산자는 두 개의 문자열을 하나로 결합합니다.

```
>>> str1 + str2 # + 기호는 문자열을 하나로 결합합니다.
             # [주의] 숫자와 문자열을 섞으면 에러가 발생합니다.
```

- * 연산자는 문자열을 지정된 횟수만큼 반복합니다.

```
>>> str2 * 3 # * 기호는 문자열을 지정된 횟수만큼 반복합니다.
```

- 멤버 연산자 **in**과 **not in**은 오른쪽 문자열에 왼쪽 문자열이 포함되어 있으면 True, 아니면 False를 반환합니다. 멤버 연산자는 리스트에도 적용할 수 있습니다.

```
>>> 'a' in str1 # str1에 문자열 'a'가 포함되어 있으면 True, 아니면 False를 반환합니다.

>>> 'a' not in str1 # str1에 문자열 'a'가 포함되어 있지 않으면 True, 아니면 False를 반환합니다.
```

문자열의 인덱싱indexing

- 문자열 인덱싱은 대괄호 안에 지정된 정수 인덱스의 문자를 선택하는 것입니다.

인덱스	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
문자	아	버	지	가		안	방	에		들	어	가	신	다	.

- 문자열을 생성하고 인덱싱을 실습합니다.

```
>>> sen = '아버지가 안방에 들어가신다.' # sen에 문자열을 할당합니다.

>>> sen[0] # sen의 첫 번째 문자(0번 인덱스)를 선택합니다. 왼쪽 코드를 실행하면 '아'가 출력됩니다.
           # Python의 인덱스는 0부터 시작합니다.

>>> sen[1] # sen의 두 번째 문자(1번 인덱스)를 선택합니다. 왼쪽 코드를 실행하면 '버'가 출력됩니다.

>>> sen[-1] # 인덱스 앞에 마이너스 부호를 추가하면 인덱스가 뒤에서부터 거꾸로 적용됩니다.
            # [참고] -0은 0과 같습니다!
```

문자열의 슬라이싱slicing

- 문자열 슬라이싱은 대괄호 안에 콜론으로 연결된 두 정수 인덱스 사이의 연속된 문자를 선택하는 것입니다.(시작:끝+1)

```
>>> sen[0] + sen[1] + sen[2] # sen의 1~3번째 문자를 하나씩 선택하여 하나로 결합합니다.

>>> sen[0:3] # '0:3'은 '첫 번째 문자(0번 인덱스)부터 세 번째 문자(3-1번 인덱스)'를 선택합니다.
# 왼쪽 코드를 실행하면, sen의 첫 번째부터 세 번째 문자인 '아버지'가 반환됩니다.

>>> sen[:3] # 콜론 앞에 정수를 생략하면 '첫 번째 문자부터' 선택합니다.
# sen의 첫 번째부터 세 번째 문자 문자인 '아버지'가 반환됩니다.

>>> sen[9:] # 콜론 뒤에 정수를 생략하면 '마지막 문자까지' 선택합니다.
# sen의 열 번째 문자부터 마지막 문자까지 연속된 '들어가신다.'가 반환됩니다.(마침표 포함)

>>> sen[9:-1] # sen의 열 번째 문자부터 끝에서 두 번째 문자인 '들어가신다'가 반환됩니다.(마침표 생략)

>>> sen[:] # sen의 전체 문자열을 선택합니다. 실제로는 sen을 복제합니다.
# [주의] 대괄호 안에 아무것도 할당하지 않으면 에러가 발생합니다.
```

문자열 공백 제거 및 변경

- sen에 새로운 문자열을 할당하고, sen의 속성과 방식으로 전처리합니다.

```
>>> sen = '\n 나의 살던 고향은 꽃피는 산골 \t' # 문자열 양쪽에 공백을 추가합니다.
                                               # '\n'은 줄바꿈, '\t'는 탭을 의미합니다.
>>> sen  # sen의 값을 출력하면 문자열 양쪽에 공백이 보입니다.
         # [참고] print() 함수를 사용하여 sen을 출력한 결과와 다릅니다.
>>> dir(sen) # sen에 포함된 속성과 방식 목록을 출력합니다.
>>> sen = sen.strip() # sen의 양쪽 공백을 제거하고 sen에 재할당합니다.
                      # [참고] lstrip(), rstrip() 함수도 있습니다.
>>> sen  # 문자열 양쪽에 공백이 제거되었습니다.
>>> sen.replace('산골', '공원') # sen의 '산골'을 '공원'으로 바꾼 결과를 출력합니다.
                                # [참고] 문자열을 변경한 결과를 sen에 재할당하지 않았습니다.
>>> sen  # sen은 값이 업데이트되지 않았으므로, '산골'이 그대로 있습니다.
```

문자열 인덱스 확인

- 문자열에 포함된 특정 문자(열) 개수 및 처음 나오는 인덱스를 반환합니다.

```
>>> sen.count(' ') # sen에 포함된 공백white space 개수를 반환합니다.

>>> sen.index(' ') # sen에서 공백이 처음 나오는 인덱스를 반환합니다.

>>> sen.index(' ', 3, len(sen)) # index() 함수에 탐색할 범위를 설정할 수 있습니다.
                                # 왼쪽 코드는 3번부터 마지막 인덱스 사이에 있는 공백을 탐색합니다.

>>> sen.index('산골') # 글자수가 2 이상인 문자도 탐색할 수 있습니다.

>>> sen.index('공원') # 문자열에 없는 문자를 입력하면 에러가 발생합니다.

>>> sen.find('공원') # find() 함수는 index() 함수와 같은 기능을 수행합니다.
                     # 다만 문자열에 없는 문자를 입력하면 에러를 발생시키는 대신 -1을 반환합니다.
```

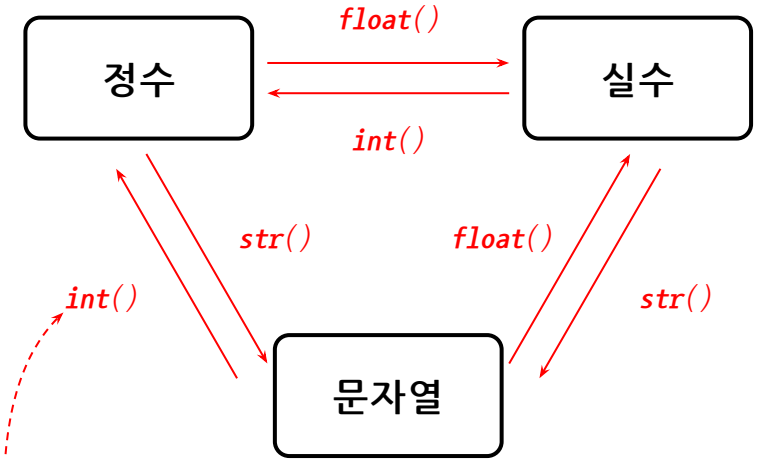
문자열 분리 및 결합

- 문자열을 구분자separator로 분리하거나 결합할 수 있습니다.

```
>>> sen.split('은') # sen을 문자 '은'을 기준으로 분리합니다. '은'은 구분자로 사용되었습니다.
                    # [참고] 문자열을 분리한 결과를 sen에 재할당하지 않았습니다.

>>> strs = sen.split() # sen을 공백으로 분리한 결과로 strs를 생성합니다.

>>> strs  # strs를 출력합니다. 대괄호 안에 여러 문자열이 포함되어 있음을 알 수 있습니다.

>>> type(strs) # strs의 클래스를 확인합니다. strs의 클래스는 list입니다.

>>> ' '.join(strs) # strs의 원소 사이에 공백을 추가하여 하나의 문자열로 결합합니다.

>>> '-'.join(strs) # 원하는 문자를 구분자로 설정할 수 있습니다.
```

원소의 형 변환casting

- 클래스로 원소의 형 변환을 실행합니다.



 # [주의] 소수점이 있는 문자열은 정수로 변환할 수 없으므로 먼저 실수로 변환한 다음, 실수를 정수로 변환해야 합니다.

- 형 변환을 실습합니다.

```
>>> a = '1.2' # a에 문자열 '1.2'를 할당합니다.
>>> type(a) # a의 클래스를 확인합니다.
# a의 클래스는 str입니다.
>>> b = float(a) # a를 실수로 형 변환한 다음 b에 할당합니다.
>>> type(b) # b의 클래스를 확인합니다.
# b의 클래스는 float입니다.
>>> c = int(a) # 소수점 있는 문자열을 정수로 형 변환할 수 없습니다.
>>> c = int(b) # b를 정수로 형 변환한 다음 c에 할당합니다.
>>> type(c) # c의 클래스를 확인합니다.
# c의 클래스는 int입니다.
```

문자열 포맷팅formatting

- 문자열 포맷팅은 포맷 코드를 사용하여 문자열 안에 변수 값을 삽입하는 것입니다.

%s(문자열)	%d(정수)	%f(실수)	%%('%' 출력)

- 문자열 포맷팅을 실습합니다.

```
>>> rate = 3.2  # rate와 name에 적당한 값을 할당합니다.
>>> name = '홍길동'
>>> '%s 고객님의 이자율은 %f %%입니다.' %(rate, rate)  # name과 rate의 값을 순차적으로 '%s'와 '%f'에 반영합니다.
>>> '%s 고객님의 이자율은 %.2f %%입니다.' %(rate, rate)  # 실수의 소수점 자리수를 설정하여 출력합니다.
```

format() 함수를 사용한 포맷팅

- **format**() 함수에 입력된 변수의 인덱스를 중괄호 안에 지정할 수 있습니다.

```
>>> '{0} 고객님의 이자율은 {1} %입니다.'.format(name, rate)
>>> '{} 고객님의 이자율은 {} %입니다.'.format(name, rate)  # 변수를 순서대로 지정했다면 인덱스를 생략해도 됩니다.
```

- 실수의 소수점 자릿수를 설정할 수 있습니다.

```
>>> '{} 고객님의 이자율은 {:.2f} %입니다.'.format(name, rate)  # [주의] 반드시 콜론을 추가해야 합니다.
```

- Python 3.6에서 도입된 f 문자열을 사용하면 간결하게 코딩할 수 있습니다.

```
>>> f'{name} 고객님의 이자율은 {rate:.2f} %입니다.'
```

부울bool

- 부울은 True 또는 False를 표현하는 자료형입니다.

```
>>> a = True  # a에 True를 할당합니다.

>>> type(a) # a의 클래스를 확인합니다. a의 클래스는 bool입니다.
            # False로도 실행해보세요.
```

- 어떤 객체에 값 또는 원소가 있으면 True, 없으면 False로 반환됩니다.

값/원소가 있으면 True	값/원소가 없으면 False
1, '사과', ['a', 'b'], (1, 2), {'키':165}	0, '', [], (), {}, None

  - if 조건문과 while 반복문에서 사용할 수 있으므로 위 내용을 숙지하시기 바랍니다.

```
>>> bool(1) # 위 표에 있는 내용을 차례대로 실행해보세요.
```

Python 자료구조

리스트List

- 리스트는 숫자, 문자열 또는 리스트 등 다양한 원소를 갖는 자료구조입니다.

```
>>> a = []; a  # 빈 리스트를 생성하고 출력합니다. 리스트는 원소를 대괄호로 감싸고 있습니다.
            # a = list()를 실행해도 같은 결과를 얻습니다.
>>> type(a) # a의 클래스를 확인합니다.
            # a의 클래스는 list입니다.
>>> b = [1, 5, 3]; b  # 리스트 원소 사이에 콤마로 구분하여 생성합니다.
                      # b를 출력하면 입력된 원소의 순서가 유지되는 것을 알 수 있습니다.
>>> len(b) # len() 함수는 리스트에 포함된 원소 개수를 반환합니다.
           # [참고] 문자열은 글자수를 반환합니다.
>>> c = [1, 2.0, '3']; c  # c를 출력합니다.
                          # 리스트는 다양한 원소를 포함할 수 있습니다.
>>> d = [a, b, c]; d  # 리스트를 원소로 갖는 리스트를 생성합니다.
```

리스트 연산자

- + 연산자는 두 개의 리스트를 하나의 리스트로 결합합니다.

```
>>> b + c # b와 c를 결합합니다.
```

- * 연산자는 리스트 원소를 지정된 횟수만큼 반복합니다.

```
>>> c * 2 # c의 전체 원소를 두 번 반복합니다.
```

- 멤버 연산자 **in**과 **not in**으로 어떤 값이 리스트의 원소인지 여부를 확인합니다.

```
>>> 1 in b # 1이 b의 원소이면 True, 원소가 아니면 False를 반환합니다.

>>> 1 not in b # 1이 b의 원소가 아니면 True, 원소이면 False를 반환합니다.
```

리스트 인덱싱

- 리스트 인덱싱은 원소의 인덱스(위치 번호)를 스칼라로 지정하여 선택합니다.

인덱스	0	1	2	3	4
원소	1	2	'3'	'4'	['가', '나', '다']

- 리스트를 생성하고 리스트 인덱싱을 실습합니다.

```
>>> a = [1, 2, '3', '4', ['가', '나', '다']]
>>> a[0] + a[1] # a의 0번 인덱스 원소와 1번 인덱스 원소를 더합니다.
                # 0~1번 인덱스 원소는 모두 숫자이므로 덧셈 연산 결과가 반환됩니다.
>>> a[2] + a[3] # a의 2번 인덱스 원소와 3번 인덱스 원소를 결합합니다.
                # 2~3번 인덱스 원소는 모두 문자열이므로 하나로 결합된 문자열이 반환됩니다.
>>> a[4][0] # a의 4번 인덱스 원소는 리스트이므로, 대괄호를 이어서 사용하면 리스트의 개별 원소를 반환합니다.
```

리스트 슬라이싱

- 콜론 연산자는 리스트의 연속된 원소를 선택하고 리스트로 반환합니다.

```
>>> a[0:2] # a의 0~1번 인덱스 원소를 선택하고 리스트로 반환합니다.
>>> a[2:4] # a의 2~3번 인덱스 원소를 선택하고 리스트로 반환합니다.
>>> a[:4] # a의 0~3번 인덱스 원소를 선택하고 리스트로 반환합니다.
>>> a[4:] # a의 4~마지막 원소를 선택하고 리스트로 반환합니다.
>>> a[:-1] # a의 처음부터 끝에서 두 번째 원소를 선택하고 리스트로 반환합니다.
```

[참고] 리스트 인덱싱 및 슬라이싱 관련 주의사항

- 리스트 인덱싱 및 슬라이싱과 관련하여 주의해야 할 내용입니다.

```
>>> a[] # 대괄호 안에 아무것도 입력하지 않으면 에러가 발생합니다.

>>> a[:] # 빈 콜론을 입력하면 리스트 a의 전체 원소를 반환합니다.

>>> a[0] # 대괄호 안에 인덱스를 스칼라로 설정하면 해당 원소를 본래 자료형으로 반환합니다.

>>> a[0:1] # 대괄호 안에 콜론을 사용하면 연속된 원소를 항상 리스트로 반환합니다.

>>> a[[0, 2]] # 대괄호 안에 리스트를 입력하면 에러가 발생합니다.
             # [주의] 대괄호 안에는 인덱스 스칼라 또는 콜론만 가능합니다.

>>> [a[0], a[2]] # 만약 a의 0번과 2번 인덱스 원소로 리스트를 생성하려면 왼쪽과 같이 코딩합니다.
```

이중 콜론 연산자

- 이중 콜론 연산자는 연속된 정수의 간격을 설정합니다.(시작:끝+1:간격)

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> a[1:5:2] # a의 1~4번 인덱스 원소에서 1번, 3번 인덱스 원소를 선택하고 리스트로 반환합니다.
>>> a[:5:2] # a의 0~4번 인덱스 원소에서 0번, 2번, 4번 인덱스 원소를 선택하고 리스트로 반환합니다.
>>> a[1::2] # a의 1~마지막 원소까지 두 칸 간격으로 원소를 선택하고 리스트로 반환합니다.
>>> a[::2] # a의 처음부터 마지막 원소까지 두 칸 간격으로 원소를 선택하고 리스트로 반환합니다.
>>> a[::-1] # a의 전체 원소를 역순으로 반환합니다. 내림차순 정렬이 아닙니다.
```

range() 함수 사용법

- **range**() 함수는 0부터 입력된 정수 앞까지 연속된 정수를 반환합니다.

```
>>> a = range(5); a  # 0부터 4까지 연속된 정수를 a에 할당하고, a를 출력합니다.
>>> type(a) # a의 클래스를 확인합니다. a의 클래스는 range입니다.
>>> a = list(a); a  # a를 리스트로 변환하여 a에 재할당한 다음, a를 출력합니다.
>>> type(a) # a의 클래스를 확인합니다. a의 클래스는 list입니다.
```

- **range**() 함수에 입력하는 정수 개수에 따라 결과가 달라집니다.

```
>>> list(range(1, 6)) # range() 함수에 정수를 두 개 입력하면, 두 숫자 사이의 연속된 정수를 반환합니다.
>>> list(range(1, 11, 2)) # range() 함수에 정수를 세 개 입력하면, 마지막 숫자는 간격으로 설정됩니다.
```

리스트 원소 추가

- 리스트에 마지막 원소로 어떤 값을 추가하려면 **append**() 함수를 이용합니다.

```
>>> a.append(5) # a에 정수 5를 마지막 원소로 추가합니다.
# append() 함수는 스칼라/리스트를 1개만 추가할 수 있습니다.

>>> a  # a를 출력합니다. a의 마지막 원소로 5가 추가되었습니다.

>>> a.append([6, 7]) # append() 함수에 리스트를 입력하면, 해당 리스트가 원소로 추가됩니다.

>>> a  # a를 출력합니다. a의 마지막 원소로 [6, 7]이 추가되었습니다.
```

- 리스트에 2개 이상의 원소를 동시에 추가하려면 **extend**() 함수를 이용합니다.

```
>>> a.extend([8, 9]) # extend() 함수에 입력된 리스트 원소를 a에 추가합니다.
# [주의] extend() 함수는 1개의 리스트만 입력받습니다.

>>> a  # a를 출력합니다. a의 마지막 원소로 8, 9가 추가되었습니다.
```

리스트 원소 삽입 및 삭제

- 리스트의 특정 인덱스 앞에 원소를 추가하려면 **insert**() 함수를 이용합니다.

```
>>> a.insert(1, 10) # a의 1번 인덱스 앞에 정수 10을 삽입합니다.
                    # a의 1번 인덱스부터 마지막 원소까지 모두 한 칸씩 뒤로 밀립니다.

>>> a  # a를 출력합니다. a의 두 번째 원소로 10가 삽입되었습니다.
```

- 리스트 원소를 삭제할 때 **remove**() 및 **pop**() 함수를 이용합니다.

```
>>> a.remove([6, 7]) # remove() 함수는 입력된 값을 (중복이 있어도 맨 처음) 하나만 삭제합니다.
                     # [참고] remove() 함수는 리스트에 없는 원소를 입력하면 에러가 발생합니다.

>>> a  # a를 출력합니다. a의 원소 [6, 7]이 삭제되었습니다.

>>> a.pop(2) # pop() 함수는 입력된 인덱스의 원소를 출력하고, 해당 원소를 제거한 결과를 재할당합니다.
             # [참고] pop() 함수의 괄호 안을 생략하면 마지막 원소가 제거됩니다.

>>> a  # a를 출력합니다. a의 2번 인덱스(세 번째) 원소 1이 삭제되었습니다.
```

리스트 원소 변경

- 리스트 원소를 변경하려면 인덱스를 이용합니다.

```
>>> a[0] = 1 # a의 0번 인덱스 원소를 정수 1로 변경합니다.

>>> a  # a를 출력합니다. a의 0번 인덱스(첫 번째) 원소가 1로 변경되었습니다.

>>> a[3:5] = [7, 6] # a의 3~4번 인덱스 원소를 리스트 [7, 6]의 원소로 변경합니다.
                    # [참고] 왼쪽 리스트 원소 개수와 오른쪽 리스트 원소 개수가 달라도 됩니다.

>>> a  # a를 출력합니다. a의 3~4번 인덱스 원소인 3, 4가 각각 6, 7로 변경되었습니다.

>>> a[3] = [4, 3] # a의 3번 인덱스 원소를 리스트 [4, 3]으로 변경합니다.

>>> a  # a를 출력합니다. a의 3번 인덱스(네 번째) 원소가 [4, 3]으로 변경되었습니다.
```

리스트 원소 확인

- 리스트에 포함된 특정 원소 개수를 세려면 **count**() 함수를 이용합니다.

```
>>> a.count(1) # a의 원소에 1이 몇 개 있는지 확인합니다.

>>> a.count(7) # 리스트에 없는 원소를 입력하면 0을 반환합니다.
```

- 리스트에서 특정 원소의 인덱스를 반환하려면 **index**() 함수를 이용합니다.

```
>>> a.index(1) # 리스트 원소 중 1이 처음 나오는 인덱스를 반환합니다.

>>> a.index(7) # 리스트에 없는 원소를 입력하면 에러가 발생합니다.
# 한편, 리스트는 문자열과 다르게 find() 방식이 없습니다.
```

리스트 원소 정렬

- **sort**() 함수로 리스트 원소를 오름차순으로 정렬합니다.

```
>>> a.sort()  # sort() 함수는 리스트 원소의 클래스가 모두 같을 때 정상적으로 실행됩니다.
              # 따라서 자료형이 다른 리스트의 원소를 정렬하려고 하면 에러가 발생합니다.

>>> a.remove([4, 3]) # a에 포함된 리스트 [4, 3]을 삭제합니다.

>>> a.sort()  # a의 원소를 오름차순으로 정렬합니다.
              # 내림차순으로 정렬하려면 sort() 함수에 reverse = True를 추가합니다.

>>> a  # a를 출력합니다. a의 원소가 오름차순으로 정렬되었습니다.
```

- 리스트 원소를 역순으로 변경하려면 **reverse**() 함수를 이용합니다.

```
>>> b.reverse()  # reverse() 함수는 내림차순으로 정렬이 아니고, 원소의 순서를 거꾸로 뒤집는 것입니다.
                 # 따라서 리스트 원소의 클래스가 섞여 있어도 에러가 발생하지 않습니다.

>>> b  # b를 출력합니다. b의 원소가 역순으로 변경되었습니다.
```

튜플Tuple

- 튜플은 원소를 변경할 수 없는 리스트입니다.

```
>>> t = (1, 2.0, '3'); t # 튜플을 생성하고 출력합니다.
                         # 리스트와 다른 점은 소괄호로 감싸져 있다는 것입니다.
>>> type(t) # t의 클래스를 확인합니다. t의 클래스는 tuple입니다.
>>> tuple(c) # tuple() 클래스는 리스트를 튜플로 변환합니다.
>>> list(t) # list() 클래스는 튜플을 리스트로 변환합니다.
>>> t.append(4) # 튜플 객체에는 append() 방식이 없으므로 에러가 발생합니다.
>>> t.remove(1) # 튜플 객체에는 remove() 방식이 없으므로 에러가 발생합니다.
>>> t[2] = 3 # 튜플 객체에는 원소를 할당할 수 없으므로 에러가 발생합니다.
```

딕셔너리Dictionary

- 딕셔너리는 키key와 값value이 콜론으로 결합된 원소를 갖는 자료구조입니다.
  - 딕셔너리는 중괄호 안에 'key': value를 콤마로 연결하거나 **dict**() 클래스로 생성합니다.
  - 딕셔너리의 value에는 스칼라(숫자, 문자열), 리스트 및 튜플을 할당할 수 있습니다.
- 딕셔너리는 원소의 순서를 유지하지 않습니다. [참고] 정수 인덱스가 없습니다!
  - 따라서 인덱스를 사용하지 못하며, 대신 key를 기준으로 value를 출력할 수 있습니다.
- 딕셔너리의 원소를 추가, 삭제, 변경할 때는 key를 이용합니다.
- 딕셔너리에 포함된 key는 중복을 허용하지 않습니다.
  - 같은 key에 value가 여러 번 할당되면, 가장 마지막에 할당된 value로 업데이트합니다.

딕셔너리 생성

- 딕셔너리를 생성하고 클래스를 확인합니다.

```
>>> d = {'item': 'pants', 'size': ['S', 'M', 'L', 'XL']}
>>> d  # d를 출력합니다.
>>> type(d) # d의 클래스를 확인합니다. d의 클래스는 dict입니다.
```

 # [주의] 중괄호 안의 key를 따옴표로 감싸주어야 합니다.
그렇지 않으면 Python 변수의 값을 인식합니다.

- **dict**() 클래스로 딕셔너리를 생성합니다.

```
>>> dict(item = 'pants', size = ['S', 'M', 'L', 'XL'])
```

 # key를 따옴표로 감싸지 않고 콜론 대신 등호를 사용합니다.
 # [주의] key를 따옴표로 감싸면 에러가 발생합니다.

딕셔너리 원소 확인 및 선택

- 딕셔너리에 어떤 key가 포함되어 있는지 확인합니다.

```
>>> 'item' in d  # d의 key에 'item'이 있으면 True, 없으면 False를 반환합니다.
>>> 'shop' in d  # d의 key에 'shop'이 있으면 True, 없으면 False를 반환합니다.
```

- 딕셔너리의 key로 value를 반환합니다.

```
>>> d['item'] # d에서 key가 'item'인 원소의 value를 반환합니다.
>>> d['shop'] # d에서 key가 'shop'인 원소의 value를 반환합니다.
              # [주의] 딕셔너리에 없는 key를 입력하면 에러가 발생합니다.
```

딕셔너리 원소 추가, 변경 및 삭제

- 딕셔너리에 원소를 추가, 변경 및 삭제합니다.

```
>>> d['shop'] = 'A1' # d에 key가 'shop'이고, value가 'A1'인 원소를 추가합니다.
>>> d  # d를 출력합니다. key가 'shop'인 원소가 추가되었습니다.
>>> d['item'] = 'skirt' # d에서 key가 'item'인 원소의 value를 'skirt'로 변경합니다.
>>> d['item'] # d에서 key가 'item'인 원소의 value를 출력합니다. value가 'skirt'로 변경되었습니다.
>>> del d['item'] # del 문을 이용하여 d에서 key가 'item'인 원소를 제거합니다.
>>> d  # d를 출력합니다. key가 'item'인 원소가 삭제되었습니다.
```

[참고] 딕셔너리 key와 value 반환

- **keys**() 함수는 딕셔너리의 key를 리스트로 반환합니다.

```
>>> list(d.keys())
```

- **values**() 함수는 딕셔너리의 values를 리스트로 반환합니다.

```
>>> list(d.values())
```

- **items**() 함수는 딕셔너리의 key와 value를 함께 리스트로 반환합니다.

```
>>> list(d.items()) # key와 value가 튜플로 묶이고, 튜플은 반환되는 리스트의 원소가 됩니다.
```

집합Set

- 집합은 원소의 중복을 허용하지 않고, 순서도 없는 자료구조입니다.

```
>>> s = {3, 1, 2, 1}; s # 집합을 생성하고, 출력합니다.
                        # 중복된 원소가 제거되고 오름차순으로 정렬되었습니다.
>>> type(s) # s의 클래스를 확인합니다. s의 클래스는 set입니다.
>>> set(c) # set() 클래스는 리스트를 집합으로 변환합니다.
>>> list(s) # list() 클래스는 집합을 리스트로 변환합니다.
>>> s & set(c) # 두 집합의 교집합을 반환합니다.
>>> s | set(c) # 두 집합의 합집합을 반환합니다.
>>> s - set(c) # 두 집합의 차집합을 반환합니다.
```

Python 제어문

if 조건문 기본 구주

- 어떤 조건을 만족하는지 여부에 따라 코드가 실행되는 순서를 통제합니다.

```
>>> if 조건1:          # if 조건문에 조건을 추가하고, 끝에 콜론을 추가합니다.
        코드A           # 조건1을 만족하면 코드A를 실행하고 조건문을 종료합니다.
                        # 탭 또는 공백 4칸으로 들여쓰기합니다. 만약 들여쓰기 하지 않으면 에러가 발생합니다.
    elif 조건2:         # 추가 조건을 설정하려면 elif 문을 사용합니다.
                        # elif 문은 단독으로 사용할 수 없으며, elif 문 끝에 콜론을 추가합니다.
        코드B           # 조건2를 만족하면 코드B를 실행하고 조건문을 종료합니다.
    else:               # 전체 조건을 만족하지 않을 때 마지막으로 실행할 코드가 있으면 else 문을 사용합니다.
                        # else 문은 생략할 수 있으며, else 문 끝에 콜론을 추가합니다.
        코드C           # 전체 조건을 만족하지 않으면 코드C를 실행하고 조건문을 종료합니다.
```

if 조건문 실습

- 학생의 전공을 묻고, 학습할 언어를 답변하는 조건문을 작성합니다.

```
>>> major = input('전공: ') # input() 함수는 값을 입력받아 문자열로 반환하여 major에 할당합니다.
                            # [참고] %whos를 실행하면 글로벌 변수 목록을 출력합니다.
>>> if '통계' in major: # major에 '통계'가 포함되어 있으면 True, 아니면 False를 반환합니다.
        print('Study R!') # 첫 번째 조건을 만족하면 왼쪽 코드를 실행하고 조건문을 종료합니다.
    elif '컴퓨터' in major:
        print('Study Python!') # 두 번째 조건을 만족하면 왼쪽 코드를 실행하고 조건문을 종료합니다.
    else:
        print('Study what you want!') # 전체 조건을 만족하지 않으면 왼쪽 코드를 실행합니다.
```

for 반복문 기본 구조

- for 반복문은 반복 실행할 범위가 정해져 있을 때 사용합니다.

```
>>> for 변수 in 리스트: # 변수는 리스트의 원소를 처음부터 입력받아 뒤따르는 코드를 반복 실행합니다.
                        # 리스트 자리에는 튜플 또는 문자열이 대신 사용되기도 합니다.
        코드A
        코드B  # 코드 블럭에는 for 반복문에서 설정된 변수가 포함되는 경우가 일반적입니다.
        코드C
    else:
        코드D  # 반복문을 완료하고 실행할 코드가 있으면 else문을 추가합니다.
```

for 반복문 실습

- 중식당 메뉴를 리스트로 생성하고 반복문을 사용하여 차례대로 출력합니다.

```
>>> menu = ['짜장면', '탕수육', '짬뽕', '깐풍기', '전가복', '샥스핀']
>>> for order in menu:
        print(order) # [주의] print(order) 대신 order라는 변수만 입력하면 아무 것도 출력되지 않습니다.
```

- 문자열 결합을 통해 출력되는 안내문을 변경합니다.

```
>>> for order in menu:
        print(order, '시킬까요?')
```

반복문 + 조건문

- 반복문 실행 도중 조건에 따라 코드를 다르게 실행합니다.

```
>>> for order in menu:
        print(order, '시킬까요?')
        if order in ['짜장면', '짬뽕']: # order가 '짜장면' 또는 '짬뽕'이면 아래 코드를 실행하고
                                        반복문의 처음으로 되돌아가서 다음 원소를 실행합니다.
            print('-> 요리부터 주문합시다!\n')
        else: # order가 '짜장면' 또는 '짬뽕'이 아닐 때 아래 코드를 대신 실행합니다.
            print('-> 다음 메뉴는 뭔가요?\n')
```

반복문 제어: continue

- 반복문에서 continue를 만나면 이후 코드를 실행하지 않고 처음으로 되돌아갑니다.

```
>>> for 변수 in 리스트:
        코드A          # 코드A는 조건에 상관 없이 항상 실행되는 코드이며, 생략할 수 있습니다.
        if 조건:       # 반복문 안에 if 조건문을 설정하여, 조건을 만족하면 뒤따르는 코드를 실행합니다.
            continue   # 반복문 실행 도중 continue를 만나면 반복문의 처음으로 되돌아갑니다.
                       # 리스트의 다음 원소를 변수로 입력받아 코드A부터 실행합니다.
        코드B          # if 조건문을 만족하지 않으면 코드B를 실행하고 반복문의 처음으로 되돌아갑니다.
```

반복문 제어: continue 실습

- 반복문 실행 도중 continue를 만나면 반복문의 처음으로 되돌아갑니다.

```
>>> for order in menu:
        print(order, '시킬까요?')
        if order in ['짜장면', '짬뽕']: # order가 '짜장면' 또는 '짬뽕'이면 아래 코드를 실행하고
                                        반복문의 처음으로 되돌아가서 다음 원소를 실행합니다.
            continue # continue를 만나면 반복문의 처음으로 되돌아갑니다.
        else: # order가 '짜장면' 또는 '짬뽕'이 아닐 때 아래 코드를 대신 실행합니다.
            print('-> 다음 메뉴는 뭔가요?\n')
```

반복문 제어: break

- 반복문에서 break를 만나면 반복문 전체를 중단시킵니다.

```
>>> for 변수 in 리스트:
        코드A          # 코드A는 조건에 상관 없이 항상 실행되는 코드이며, 생략할 수 있습니다.
        if 조건:       # 반복문 안에 if 조건문을 설정하여, 조건을 만족하면 뒤따르는 코드를 실행합니다.
            break      # 반복문 실행 도중 break를 만나면 반복문 전체를 중단합니다.
                       # [참고] continue는 반복문의 처음으로 되돌아간다는 점에서 break와 다릅니다.
        코드B          # if 조건문을 만족하지 않으면 코드B를 실행하고 반복문의 처음으로 되돌아갑니다.
```

반복문 제어: break 실습

- 반복문 실행 도중 break를 만나면 반복문을 중단시킵니다.

```
>>> for order in menu:
        print(order, '시킬까요?')
        if order in ['전가복', '샥스핀']:
            break # break를 만나면 반복문이 종료됩니다.
        else:
            print('-> 다음 메뉴는 뭔가요?\n')
```

[참고] 중첩 for 반복문

- 두 개의 리스트 원소에 대한 모든 경우의 수를 반복 실행합니다.(구구단 만들기)

```
>>> for i in range(2, 10): # i는 처음에 2를 입력받아 안쪽 반복문을 실행합니다.
                           # 안쪽 반복문이 모두 완료되면 i의 값이 다음 원소를 입력받습니다.
        print(f'*** {i}단 ***') # i의 값으로 '단'을 출력합니다.
        for j in range(1, 10): # j는 1~9의 값을 차례로 입력받아 안쪽 반복문을 실행합니다.
            print(f'{i} * {j} = {i*j}')
```

- 두 개의 리스트에서 서로 대응하는 원소끼리 반복문을 실행합니다.

```
>>> for i, j in zip(range(1, 10), range(1, 10)): # zip() 함수 안에 있는 두 리스트에서 서로
                                                 # 대응하는 원소를 각각 i, j로 전달합니다.
        print(f'{i} * {j} = {i*j}') # 위 코드와 실행 결과가 다릅니다.
```

반복문 실행 결과를 리스트로 반환

- 반복문을 실행하여 반환되는 값을 리스트의 원소로 추가하려면 반복문을 실행하기 전에 빈 리스트를 미리 생성해두어야 합니다.

```
>>> nums = list(range(1, 11)) # 정수 1~10을 원소로 갖는 리스트를 생성합니다.
                              # 각 원소를 제곱하여 리스트로 반환하는 2가지 방법을 비교합니다.
>>> sqrs = [] # for 반복문을 실행하기 전에 결과를 저장할 빈 리스트를 미리 생성합니다.
>>> for num in nums:
        sqrs.append(num ** 2) # nums의 각 원소를 제곱한 값을 sqrs에 마지막 원소로 추가합니다.
                              # [주의] sqrs를 미리 생성하지 않으면 에러가 발생합니다.
>>> sqrs # sqrs를 출력합니다. 1~10의 정수를 제곱한 값을 원소로 갖습니다.
```

[참고] 리스트 컴프리헨션Comprehension

- 리스트 컴프리헨션은 반복문 또는 조건문 등을 대괄호 안에서 실행하게 함으로써 코드 실행 결과를 리스트로 반환합니다.

```
>>> [num ** 2 for num in nums]
```
 # 한 줄 코드로 for 반복문과 같은 결과를 반환합니다.
[참고] 빈 리스트를 미리 생성할 필요가 없습니다.

- 리스트 컴프리헨션 안에 조건문을 추가하여 특정 조건을 만족하는 리스트 원소만 대상으로 코드를 실행할 수 있습니다.

```
>>> [num ** 2 for num in nums if num % 2 == 0]
```
 # nums의 원소 중 짝수 원소만 제곱한 결과를 리스트로 반환합니다.

- 중첩 for 반복문을 리스트 컴프리헨션으로 실행합니다.

```
>>> [f'{i} * {j} = {i*j}' for i in range(2, 10) for j in range(1, 10)]
```

while 반복문 기본 구조

- 사전에 정해진 범위는 없지만, 어떤 조건을 만족하는 한 반복문을 계속 실행하려면 while 반복문을 사용합니다.

```
>>> while 조건: # 조건이 True이면 아래 코드를 반복 실행합니다.
        코드A
        코드B   # 코드 블럭에 while 반복문에서 설정된 조건이 False가 되도록 하는 증감식이 포함되는 경우가
        코드C     일반적입니다. 그렇지 않으면 while 반복문은 무한 반복될 수 있습니다!
    else:
        코드D # 반복문을 완료하고 실행할 코드가 있으면 else문을 추가합니다.
```

while 반복문 실습

- 변수의 크기를 하나씩 줄여가면서 변경된 값을 출력합니다.

```
>>> i = 5  # i에 정수 5를 할당합니다.
>>> while i > 0:  # i가 0을 초과하면 True이므로, while 반복문이 실행됩니다.
                  # 그러다가 i가 0이 되면 False가 되면서 while 반복문을 중단합니다.
        print(i)
        i -= 1 # 반복문이 실행될 때마다 i에서 1씩 차감합니다.(증감식)
>>> while True: # 강제로 중단하기 전까지 무한 반복합니다.(멈추려면 interrupt키를 누르세요!)
        print(i)
        i += 1
```

Python 함수

사용자 정의 함수의 필요성

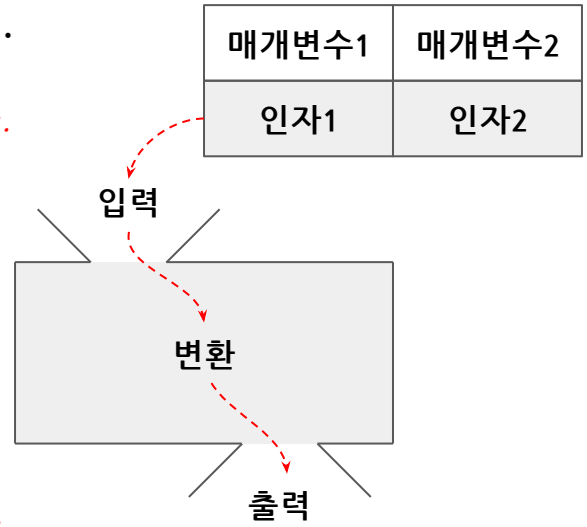
- 코딩 초심자가 당장 실행되는 코딩에 집중하면 같은 코드를 중복하기 쉽습니다.
  - 예를 들어, 체질량지수BMI를 계산하는 코드에 값만 바꿔서 중복하는 경우가 발생합니다.
- 코드 뭉치를 중복 사용하면 전체 코드가 불필요하게 길어지고, 수정해야 할 때 상당히 번거롭게 됩니다.
- 사용자 정의 함수를 생성하면 간단하게 해결됩니다.
  - (한 줄짜리) 함수에 값만 바꿔서 실행할 수 있으므로 코드가 간명해집니다.
  - 수정사항이 발생하면 사용자 정의 함수에 있는 코드만 수정하면 되므로 코드의 유지보수가 매우 쉬워집니다.

```
>>> hgt = 190
>>> wgt = 95
>>> hgt /= 100
>>> bmi = wgt / hgt**2
>>> bmi
```

사용자 정의 함수 기본 구조

- 체질량지수를 반환하는 사용자 정의 함수를 생성합니다.

```
>>> def BMI(height, weight): # 함수명과 매개변수를 정의합니다.
        bmi = weight / (height/100)**2
        return bmi # 함수가 반환할 값을 설정합니다.
```



- 사용자 정의 함수를 실행합니다.

```
>>> BMI(175, 65) # 함수의 매개변수를 생략하고 전달인자만 지정합니다.
>>> BMI(weight = 65, height = 175) # 함수의 매개변수를 추가하면 순서가 바뀌어도 정상 실행됩니다.
                                   # height는 매개변수parameter, 175는 전달인자argument입니다.
```

사용자 정의 함수: 인자의 기본값 설정

- 사용자 정의 함수를 생성할 때, 인자의 기본값을 설정할 수 있습니다.

```
>>> def BMI(height = 175, weight = 65):
        bmi = weight / (height/100)**2
        return bmi
```

- 사용자 정의 함수를 실행합니다.

```
>>> BMI() # 사용자 정의 함수를 실행할 때 인자를 생략하면 기본값이 자동으로 적용됩니다.
>>> BMI(height = 190, weight = 85) # 사용자 정의 함수의 인자를 원하는 값으로 입력합니다.
```

[참고] 람다lambda 표현식 사용법

- lambda 표현식은 '한 줄의 코드'로 함수를 만들 때 사용합니다.
  - 앞에서 만든 BMI 함수를 lambda 표현식으로 바꾸면 아래와 같습니다.

```
>>> BMI2 = lambda height = 175, weight = 65: weight / (height/100)**2
>>> BMI2(190, 85)
```

 # 콜론 뒤에 오는 코드를 실행하고 반환합니다.

- lambda 표현식은 함수명 없이도 사용할 수 있으므로 '익명함수'로 불립니다.

```
>>> (lambda height, weight: weight / (height/100)**2)(190, 85)
```

[참고] 다양한 괄호 사용법

- 소괄호, 중괄호, 대괄호 사용법에 대해 표로 정리한 것입니다.

구분	상세내용
소괄호 ()	- 튜플을 생성할 때 소괄호 안에 원소를 콤마로 연결합니다. - 함수 또는 객체의 방식method 뒤에 추가해야 합니다.
중괄호 {}	- 딕셔너리 및 집합을 생성할 때 중괄호 안에 원소를 콤마로 연결합니다. - f 문자열에서 변수를 중괄호로 감싸주어야 합니다.
대괄호 []	- 리스트를 생성할 때 대괄호 안에 원소를 콤마로 연결합니다. - 객체를 인덱싱할 때 대괄호 안에 정수 스칼라, 콜론 등을 지정합니다.

Python 표준 라이브러리

폴더와 파일 다루기

- Python 기본 라이브러리 중 폴더와 파일을 다룰 때 os와 shutil을 사용합니다.

구분	상세 내용
os.getcwd()	현재 작업경로를 출력합니다.
os.chdir()	작업경로를 변경합니다.
os.mkdir()	새로운 폴더를 생성합니다.
os.rmdir()	빈 폴더를 삭제합니다.
os.listdir()	폴더 및 파일명을 출력합니다.
shutil.rmtree()	파일과 함께 폴더를 삭제합니다.

구분	상세 내용
os.rename()	폴더 및 파일명을 변경합니다.
os.remove()	파일을 삭제합니다.
shutil.copy()	파일을 복사합니다.
shutil.move()	파일을 이동시킵니다.
os.path.isdir()	폴더 여부를 반환합니다.
os.path.isfile()	파일 여부를 반환합니다.

절대경로 vs 상대경로

- 절대경로는 '경로의 시작과 끝'을 모두 표기한 경로입니다.
  - Windows의 경우, 'C' 또는 'D' 드라이브로 시작합니다.
  - 절대경로는 유일한 경로이므로 절대경로를 사용하면 언제나 같은 결과를 가져옵니다.
  - 하지만 절대경로가 매우 길면 코딩할 때 불편할 수 있습니다.
- 상대경로는 현재 설정된 경로에서의 상대적인 위치를 의미합니다.
  - '.'은 현재 폴더, '..'은 현재 폴더의 상위 폴더를 의미합니다.
  - 상대경로는 몇 글자만으로도 경로를 지정할 수 있으므로 코딩할 때 편리합니다.
  - 하지만 실행 중인 코드 파일의 위치가 다를 수 있으므로 주의를 기울여야 합니다.

작업경로 확인, 변경 및 새 폴더 생성

- 관련 라이브러리를 호출합니다.

```
>>> import os, shutil # 여러 라이브러리를 동시에 호출하려면 콤마를 사용합니다.
                      # os, shutil 라이브러리는 파일과 폴더를 다루는데 필요한 함수를 포함하고 있습니다.
```

- 현재 작업경로를 확인합니다.

```
>>> os.getcwd() # 현재 작업경로current working directory를 확인합니다.
```

- 데이터 파일이 저장된 폴더로 작업경로를 변경합니다.(상대경로 사용)

```
>>> os.chdir(path = '../data') # '..'은 상위 폴더를 의미합니다.
```

- 새로운 폴더를 생성합니다.

```
>>> os.mkdir(path = './temp') # '.'은 현재 폴더를 의미하므로 현재 폴더에 temp 폴더를 추가합니다.
                              # [참고] temp 폴더 앞 './'를 생략해도 같은 결과를 반환합니다.
```

새로운 텍스트 파일 생성

- 텍스트 파일을 쓰기모드로 열고, 문자열을 입력합니다.

```
>>> file = open(file = 'test.txt', # 현재 작업경로에 있는 'test.txt' 파일을 지정합니다.
                                   # 만약 같은 이름의 파일이 없으면 새로 생성합니다.
                  mode = 'w', # 기존 텍스트 파일을 새로운 데이터로 덮어 씁니다.
                  encoding = 'UTF-8') # Windows 사용자는 encoding을 추가하는 것이 좋습니다.
                                      # [참고] encoding을 지정하지 않으면 'CP949'가 적용됩니다.
>>> for i in range(1, 6): # 반복문을 실행하여 여러 개의 문자열을 텍스트 파일에 입력합니다.
        file.write(f'{i} 페이지 수집!\n') # test.txt 파일에 문자열을 추가합니다.
>>> file.close() # 텍스트 파일을 닫아야 텍스트 파일이 저장됩니다.
```

기존 텍스트 파일에 데이터 추가

- 텍스트 파일을 추가모드로 열고, 문자열을 입력합니다.

```
>>> file = open(file = 'test.txt', # 현재 작업경로에 있는 'test.txt' 파일을 지정합니다.
                                   만약 같은 이름의 파일이 없으면 새로 생성합니다.
                mode = 'a', # 기존 텍스트 파일에 데이터를 추가합니다.
                encoding = 'UTF-8')
>>> for i in range(11, 16):
        file.write(f'{i} 페이지 수집!\n')
>>> file.close()
```

텍스트 파일을 문자열로 읽기

- 텍스트 파일을 문자열str 읽기모드로 엽니다.

```
>>> file = open(file = 'test.txt', # 현재 작업경로에 있는 'test.txt' 파일을 지정합니다.
                                   # 만약 같은 이름의 파일이 없으면 에러가 발생합니다.
           mode = 'r', # 텍스트 파일을 사람이 읽을 수 있는 문자열str로 읽습니다.
           encoding = 'UTF-8') # Windows 사용자가 'test.txt' 파일을 생성할 때 encoding =
                               # 'UTF-8'을 추가했다면, 읽을 때에도 지정해주어야 합니다.
>>> text = file.read() # file을 읽고, text에 할당합니다.
>>> text  # text를 출력합니다. 사람이 읽을 수 있는 문자열로 출력됩니다.
>>> type(text) # text의 클래스를 확인합니다. text의 클래스는 str입니다.
               # [참고] str은 사람이 읽을 수 있는 문자열입니다.
>>> text.encode(encoding = 'UTF-8') # encode() 함수는 str을 bytes로 변환합니다.
                                    # [참고] encoding 매개변수의 기본값은 'UTF-8'입니다.
```

텍스트 파일을 바이너리로 읽기

- 텍스트 파일을 바이너리binary 읽기모드로 엽니다.

```
>>> file = open(file = 'test.txt',  # 현재 작업경로에 있는 'test.txt' 파일을 지정합니다.
                                    # 만약 같은 이름의 파일이 없으면 에러가 발생합니다.
                 mode = 'rb') # 텍스트 파일을 사람이 읽을 수 없는 바이너리binary 데이터로 읽습니다.
                              # [주의] 바이너리 모드로 읽을 때에는 인코딩을 추가할 수 없습니다.
>>> text = file.read() # file을 읽고, text에 할당합니다.

>>> text  # text를 출력합니다. 사람이 읽을 수 없는 코드로 출력됩니다.

>>> type(text) # text의 클래스를 확인합니다. text의 클래스는 bytes입니다.
               # [참고] bytes는 사람이 읽을 수 없는 코드입니다.

>>> text.decode(encoding = 'UTF-8') # decode() 함수는 bytes를 str로 변환합니다.
                                    # [참고] encoding 매개변수의 기본값은 'UTF-8'입니다.
                                    # Windows 사용자가 'test.txt' 파일을 생성할 때 encoding =
                                      'UTF-8'을 추가하지 않았다면, 'CP949'를 지정해야 합니다.
```

파일 복사 및 이동

- 현재 작업경로에 포함되어 있는 폴더명과 파일명을 출력합니다.

```
>>> os.listdir() # 폴더명과 파일명을 문자열 원소로 갖는 리스트를 반환합니다.
```

- 기존 파일을 복사합니다.(파일명, 파일명)

```
>>> shutil.copy(src = 'test.txt', dst = 'copy1.txt')
```

- 기존 파일을 다른 폴더로 복사합니다.(파일명, 폴더명)

```
>>> shutil.copy(src = 'test.txt', dst = './temp')
```

- 파일을 이동시킵니다.(파일명, 폴더명)

```
>>> shutil.move(src = 'copy1.txt', dst = './temp')
```

파일명 변경

- 작업경로를 변경합니다.

```
>>> os.chdir(path = './temp')
```

- 현재 작업경로에 포함되어 있는 폴더명과 파일명을 출력합니다.

```
>>> os.listdir()
```

- 파일명을 변경합니다.(파일명, 파일명)

```
>>> os.rename(src = 'test.txt', dst = 'copy2.txt')
```

- 파일을 복사합니다.(파일명, 파일명)

```
>>> shutil.copy(src = 'copy2.txt', dst = 'copy3.txt')
```

 # 여러 파일을 삭제하는 실습을 위해 txt 파일을 3개로 만듭니다.

파일 삭제

- 현재 작업경로에 새로운 폴더를 생성합니다.

```
>>> os.mkdir(path = './temp2')
```

- 파일은 한 개씩 삭제할 수 있지만, 폴더는 삭제할 수 없습니다.

```
>>> os.remove(path = 'copy1.txt') # 파일을 삭제합니다. 한 번에 한 개씩 삭제할 수 있습니다.
>>> os.remove(path = 'temp2') # [주의] 폴더는 삭제할 수 없으므로, 에러가 발생합니다.
```

- 따라서 현재 작업경로에 포함된 모든 파일을 삭제하려고 하면 에러가 발생합니다.

```
>>> for i in os.listdir():
        os.remove(path = i) # i가 가리키는 값이 폴더명인 'temp2'일 때, 에러가 발생합니다.
```

파일 삭제(계속)

- 파일명에서 확장자를 분리합니다.

```
>>> print(i) # for 반복문에서 에러가 발생할 때의 i가 가리키는 값을 출력합니다.

>>> os.path.isdir(path = i) # i가 가리키는 값이 폴더명이면 True, 아니면 False를 반환합니다.

>>> os.path.isfile(path = i) # i가 가리키는 값이 파일명이면 True, 아니면 False를 반환합니다.
```

- 여러 파일을 모두 삭제하려면 아래와 같이 반복문에 조건문을 추가합니다.

```
>>> for i in os.listdir():
        if os.path.isfile(path = i): # i가 가리키는 값이 파일명이면 삭제합니다.
            os.remove(path = i)
```

폴더 삭제

- 상위 폴더로 이동합니다.

```
>>> os.chdir(path = '..') # '..'은 상위 폴더를 의미하므로 작업경로를 상위 폴더로 이동합니다.
```

- 현재 작업경로에 포함되어 있는 폴더명과 파일명을 출력합니다.

```
>>> os.listdir()
```

- 빈 폴더가 아닌 폴더를 삭제하려고 하면 에러가 발생합니다.

```
>>> os.rmdir(path = './temp') # os.rmdir() 함수는 비어 있는 폴더만 삭제합니다.
```

- 빈 폴더가 아닌 폴더도 삭제합니다.

```
>>> shutil.rmtree(path = './temp') # [주의] shutil.rmtree() 함수는 파일이 있는 폴더도 삭제하므로
                                  # 실행하기 전에 폴더에 포함된 파일 목록을 확인하시기 바랍니다.
```

날짜 시간 데이터 다루기

- 관련 라이브러리를 호출합니다.

```
>>> from datetime import datetime, timedelta # datetime 라이브러리에서 datetime과 timedelta 함수만 호출합니다.
```

- 현재 날짜와 시간을 출력합니다.

```
>>> ctime = datetime.now() # [참고] now() 대신 today() 함수를 사용할 수 있습니다.

>>> ctime # ctime을 출력합니다. 년, 월, 일, 시, 분, 초, 마이크로초(백만분의 1초)를 차례대로 출력합니다.

>>> ctime.date() # ctime의 날짜를 출력합니다.

>>> ctime.time() # ctime의 시간을 출력합니다.

>>> ctime.timestamp() # ctime의 유닉스 시간(POSIX 시간 또는 Epoch 시간)을 출력합니다.
# 협정 세계시(UTC) 기준 1970년 1월 1일 00:00:00부터 현재까지 경과된 누적 초입니다.
```

날짜 시간 데이터를 문자열로 변환

- 날짜 시간 데이터를 문자열로 변환합니다. 날짜 포맷은 뒷 페이지를 참조하세요.

```
>>> ctime.strftime('%Y년 %m월 %d일 %A %H:%M:%S %p') # 2021년 01월 01일 Friday 09:00:00 AM 형태로 출력됩니다.
```

- **strftime**() 함수에 지정하는 문자열의 인코딩 문제로 에러가 발생할 수 있습니다.
  - 날짜 포맷 문자열을 지정하고, 유니코드 인코딩 및 디코딩합니다.
  - **strftime**() 함수로 반환되는 문자열을 인코딩 및 유니코드 디코딩합니다.
- [참고] 문자열의 인코딩 문제로 발생하는 에러를 해결합니다.

```
>>> dtf = '%Y년 %m월 %d일 %A %H:%M:%S %p'.encode('unicode-escape').decode()
>>> ctime.strftime(dtf).encode().decode('unicode-escape')
```

[참고] 날짜 시간 관련 주요 포맷

- 아래 포맷을 참고하면 관심 있는 날짜 및 시간 데이터만 출력할 수 있습니다.

포맷	상세 내용	예시	포맷	상세 내용	예시
%a	요일(영어 약자)	Wed	%M	분(숫자)	01
%A	요일(영어)	Wednesday	%p	AM/PM 표기	AM
%b	월(영어 약자)	Jan	%S	초(숫자)	01
%B	월(영어)	January	%w	요일(숫자)	3
%c	날짜와 시간	time.ctime()	%x	날짜만 출력	01/01/20
%d	일(숫자)	01	%X	시간만 출력	01:01:01
%H	시(숫자-24시간)	12	%Y	연도(4자리)	2020
%I	시(숫자-12시간)	01	%y	연도(2자리)	20
%m	월(숫자)	01	%Z	타임존time zone 출력	KST

문자열 및 정수를 날짜 시간 데이터로 변환

- 문자열을 날짜 시간 데이터로 변환합니다.

```
>>> birth = '2001년 2월 3일 4시 5분 6초'  # 날짜 시간 데이터로 변환할 문자열을 지정합니다.
                                          # strptime() 함수에 문자열과 날짜 포맷을 지정합니다.
>>> birth = datetime.strptime(birth, '%Y년 %m월 %d일 %H시 %M분 %S초')
>>> birth  # birth를 출력합니다. 년, 월, 일, 시, 분, 초, 마이크로초(백만분의 1초)를 차례대로 출력합니다.
```

- 년, 월, 일, 시, 분, 초에 해당하는 정수를 날짜 시간 데이터로 변환합니다.

```
>>> birth = datetime(2001, 2, 3, 4, 5, 6)
>>> birth  # 문자열을 날짜 시간 데이터로 변환한 결과와 동일합니다.
```

날짜 시간 데이터의 연산

- 태어난 날로부터 현재까지의 기간 차이를 일수와 초로 확인합니다.

```
>>> dtGap = ctime - birth  # 태어난 날로부터 현재까지의 기간 차이를 dtGap으로 생성합니다.

>>> dtGap.days  # 두 날짜의 기간 차이에서 일수만 출력합니다.
```

- 날짜 및 시간 차이를 설정하고, 덧셈과 뺄셈 연산을 실행합니다.

```
>>> d_day = datetime(2021, 1, 1, 19) # 기준일자를 설정합니다. [참고] 분과 초를 생략했습니다.

>>> d_day + timedelta(days = 100) # [참고] timedelta() 함수에 days와 weeks를 지정할 수 있습니다.
                                  # 하지만 years 또는 months는 지정할 수 없습니다.

>>> d_day - timedelta(hours = 2) # [참고] timedelta() 함수에 hours, minutes 및 seconds를 지정할 수
                                 # 있습니다!
```

[참고] 로케일Locale 이란?

- 국가마다 서로 다른 문화를 가지고 있으므로 문자, 통화, 숫자 및 날짜/시간 등을 표기하는 방법에서 차이가 있습니다.
- 컴퓨터 사용자가 거주하는 국가에 따라 문자, 통화, 숫자 및 날짜/시간을 다르게 표기할 수 있도록 운영체제(OS)는 국가별 로케일을 제공하고 있습니다.
- 로케일은 국가마다 표기 형식을 설정하는 것을 의미합니다.
    - LC_CTYPE(문자 유형), LC_COLLATE(문자열 정렬), LC_TIME(날짜/시간 형식), LC_MONETARY(통화 형식), LC_MESSAGES(메시지 표시), LC_NUMERIC(숫자 형식)
    - 로케일이 변경되면 위 항목의 표기 방식이 달라집니다. 예를 들어, 날짜/시간 로케일을 대한민국으로 설정하면 영어로 출력되었던 요일이 한글로 출력됩니다.

[참고] 운영체제별 로케일 이름과 인코딩 방식

로케일	Windows		Mac	
	로케일 이름	인코딩 방식	로케일 이름	인코딩 방식
대한민국	korean	CP949	ko_KR	UTF-8
미국	english	ISO8859-1	en_US	UTF-8
중국	chinese	CP936	zh_CN	UTF-8
일본	japanese	CP932	ja_JP	UTF-8
기본값	C	ASCII	C	ASCII

[참고] 날짜/시간 로케일 확인 및 변경

- 관련 라이브러리를 호출합니다.

```
>>> import locale
```

- 현재 설정된 날짜/시간 로케일을 확인합니다.

```
>>> locale.getlocale(category = locale.LC_TIME)
```

- 한국 날짜/시간 로케일로 변경합니다.

```
>>> locale.setlocale(category = locale.LC_TIME, locale = 'ko_KR')
```

- 날짜 시간 데이터를 문자열로 변환합니다.

```
>>> ctime.strftime('%Y년 %m월 %d일 %A %H:%M:%S %p') # 요일이 한글로 출력됩니다.
```

numpy 라이브러리

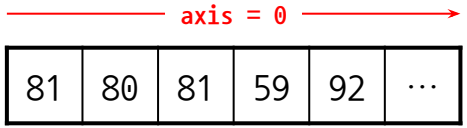
numpy 라이브러리 소개

- numpy 라이브러리를 이용하면 다음과 같은 작업을 수행할 수 있습니다.
  - 1~3차원의 배열ndarray을 생성할 수 있습니다.
  - 생성된 배열의 차원dimension을 확인하고 형태shape를 변경할 수 있습니다.
  - 1차원 배열(벡터) 및 2차원 배열(행렬) 연산을 수행할 수 있습니다. [선형대수]
    - 1차원 배열: 벡터의 덧셈과 뺄셈, 스칼라배, 벡터의 내적 연산 등
    - 2차원 배열: 행렬의 덧셈과 뺄셈, 스칼라배, 행렬의 곱셈, 판별식, 역행렬 등
- numpy 라이브러리를 호출합니다.

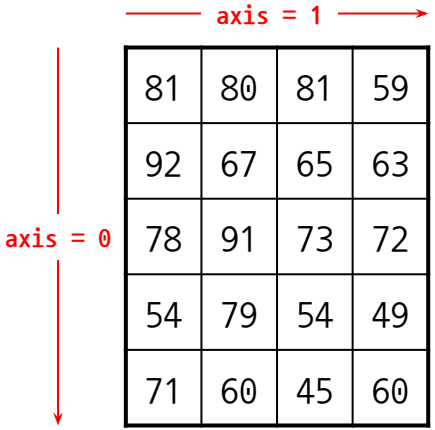
```
>>> import numpy as np  # numpy 라이브러리를 호출하고, 'np'로 사용하도록 설정합니다.
```

[참고] 배열의 시각적 예시

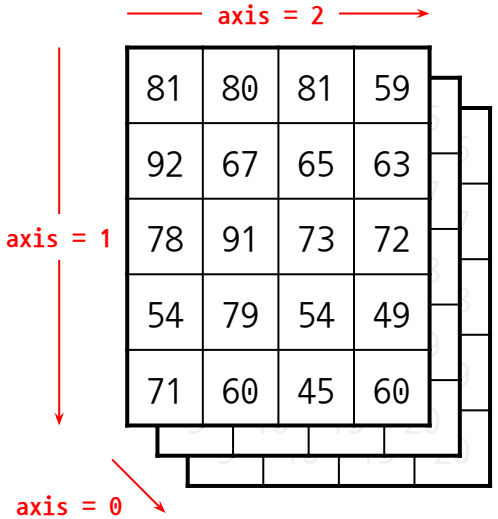
[1차원 배열]



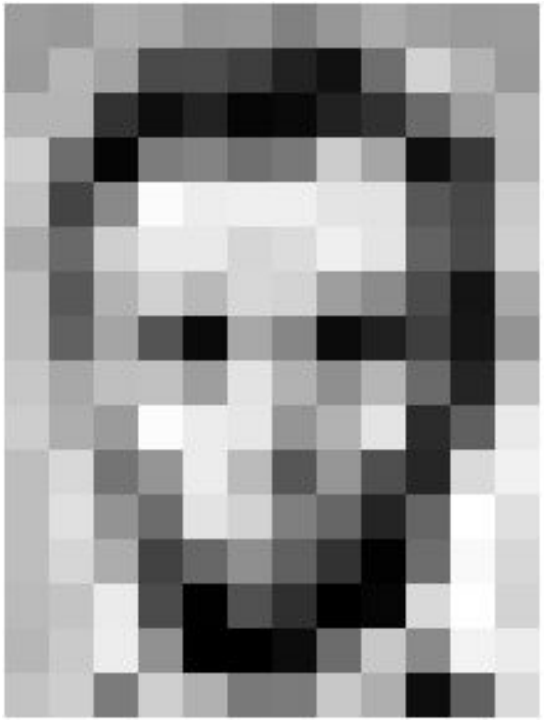
[2차원 배열]



[3차원 배열]



[참고] 행렬로 표현된 흑백 이미지 데이터 예제



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

 # 출처: http://openframeworks.kr/ofBook/chapters/image_processing_computer_vision.html

1차원 배열 생성

- 1차원 배열을 생성하고 클래스를 확인합니다.

```
>>> ar1 = np.array(object = [1, 2, 3], dtype = 'int') # 리스트로 1차원 배열을 생성합니다.
>>> ar1 # ar1을 출력합니다. 전체 원소가 한 줄로 늘어서 있는 형태입니다.
>>> type(ar1) # ar1의 클래스를 확인합니다. ar1의 클래스는 numpy.ndarray입니다.
```

- 1차원 배열의 차원수와 형태를 확인합니다.

```
>>> ar1.ndim # ar1의 차원을 확인합니다.(1차원)
>>> ar1.shape # ar1의 형태를 확인합니다. ar1은 1차원이며 원소 개수는 3입니다.
>>> ar1.dtype # ar1 원소의 자료형을 확인합니다.
```

1차원 배열의 형 변환

- 배열이 리스트와 다른 점은, 같은 자료형의 원소만 가질 수 있다는 것입니다.
  - 원소의 자료형이 다른 리스트로 배열을 생성하면 상위 자료형으로 형 변환이 일어납니다.

```
>>> ar1 = np.array(object = [1, 2.0, '3']); ar1  # 원소의 자료형이 섞여 있는 리스트로 1차원 배열을 생성합니다.
>>> ar1.dtype  # ar1 원소의 자료형은 '<U32'입니다.
```

- **astype**() 함수로 배열의 원소에 대해 형 변환을 실행할 수 있습니다.

```
>>> ar1.astype('float') # ar1의 원소를 실수로 변환합니다.
>>> ar1.astype('int') # ar1의 원소를 정수로 변환합니다. 그런데 문자열 '2.0' 때문에 에러가 발생합니다.
>>> ar1.astype('float').astype('int') # 따라서 ar1의 원소를 먼저 실수로 변환하면, 정수로 변환할 수 있습니다.
```

원소의 간격이 일정한 배열 생성

- 간격이 일정한 실수를 원소로 갖는 1차원 배열을 생성합니다.

```
>>> np.arange(5) # 0부터 4까지 연속된 정수를 반환합니다.

>>> np.arange(start = 1, stop = 6) # 1부터 5까지 연속된 정수를 반환합니다.

>>> np.arange(start = 1, stop = 11, step = 2) # 1부터 10까지 홀수를 정수로 반환합니다.

>>> np.arange(start = 0, stop = 1, step = 0.1) # 0부터 1까지 0.1 간격의 연속된 실수를 반환
                                               합니다. [참고] 1을 포함하지 않습니다.
```

- 지정된 개수만큼 원소를 갖는 1차원 배열을 생성합니다.

```
>>> np.linspace(start = 0, stop = 1, num = 11) # 0부터 1까지 원소 개수가 11개인 실수를 반환
                                               합니다. [참고] 1을 포함합니다.

>>> np.linspace(start = 80, stop = 75, num = 32)
```

원소가 반복되는 배열 생성

- 배열 전체를 두 번 반복합니다.

```
>>> np.tile(A = ar1, reps = 2)
```

- 배열 원소를 두 번씩 반복합니다.

```
>>> np.repeat(a = ar1, repeats = 2)
```

- 배열 원소별로 반복할 횟수를 지정합니다.

```
>>> np.repeat(a = ar1, repeats = [1, 2, 3])
```

- 반복 횟수를 지정할 때, **range**() 함수를 사용할 수 있습니다.

```
>>> np.repeat(a = ar1, repeats = range(3, 0, -1))
```

1차원 배열의 인덱싱 및 슬라이싱

- 배열의 인덱싱은 인덱스를 정수 스칼라로 지정하여 해당 원소를 선택합니다.

인덱스	*0*	*1*	*2*	*3*	*4*	*5*
원소	**1**	**3**	**5**	**7**	**9**	**11**

- 배열 뒤에 대괄호를 추가하고 정수 스칼라를 지정하면 해당 원소를 선택합니다.

```
>>> ar1 = np.arange(start = 1, stop = 12, step = 2)
>>> ar1[0] # ar1의 0번 인덱스(첫 번째) 원소를 선택합니다.
>>> ar1[1] # ar1의 1번 인덱스(두 번째) 원소를 선택합니다.
>>> ar1[-1] # ar1의 마지막 원소를 선택합니다.
```

1차원 배열의 인덱싱 및 슬라이싱(계속)

- 배열을 인덱싱할 때, 대괄호 안에 리스트를 사용할 수 있습니다!

```
>>> ar1[[3, 2, 1]] # 정수 스칼라 대신 정수를 원소로 갖는 리스트를 지정할 수 있습니다.(팬시 인덱싱)
                   # [주의] 리스트에서는 팬시 인덱싱이 실행되지 않습니다!
>>> ar1[[3, 3, 3]] # 리스트의 원소를 반복하면 같은 원소가 여러 번 선택됩니다.
```

- 배열의 슬라이싱은 콜론을 사용하여 연속된 원소를 선택하는 것입니다.

```
>>> ar1[:3] # ar1의 0~2번 인덱스 원소를 선택합니다.
>>> ar1[3:] # ar1의 3번 인덱스 원소부터 마지막 원소까지 선택합니다.
>>> ar1[:] # ar1의 전체 원소를 선택합니다. [주의] 콜론을 생략하면 에러가 발생합니다.
```

2차원 배열 생성

- 2차원 배열을 생성하고 클래스를 확인합니다.

```
>>> ar2 = np.array(object = [[1, 2, 3], [4, 5, 6]]) # 같은 길이의 리스트를 원소로 갖는 리스트입니다!
>>> ar2 # ar2를 출력합니다. 전체 원소가 2행 3열로 되어 있습니다.
>>> type(ar2) # ar2의 클래스를 확인합니다. ar2의 클래스는 numpy.ndarray입니다.
```

- 2차원 배열의 차원수와 형태를 확인합니다.

```
>>> ar2.ndim  # ar2의 차원을 확인합니다.(2차원)
>>> ar2.shape # ar2의 형태를 확인합니다. ar2는 2차원이며 2행 3열이므로 원소 개수는 6입니다.
>>> ar2.dtype # ar2 원소의 자료형을 확인합니다.
```

배열의 재구조화

- **reshape**() 함수로 1차원 배열을 2차원 배열로 변환합니다.

```
>>> ar1 = np.arange(12) # 0부터 11까지 12개의 원소를 갖는 1차원 배열을 생성합니다.

>>> ar1.reshape(3, 4) # ar1을 3행 4열인 2차원 배열로 변환합니다. 원소 입력 방향은 가로입니다.
                      # [주의] 원소 개수의 약수를 행 또는 열 길이로 설정할 수 있습니다.

>>> ar1.reshape(3, 4, order = 'F') # 매개변수 order에 전달되는 인자의 기본값은 'C'입니다.
                                   # 하지만 'F'로 변경하면 원소 입력 방향이 세로로 변경됩니다.

>>> ar1.reshape(4, -1) # 행 위치에 숫자, 열 위치에 -1을 할당하면 열의 크기가 자동 계산됩니다.
```

- 2차원 배열의 인덱싱 및 슬라이싱 실습을 위해 ar2를 생성합니다.

```
>>> ar2 = ar1.reshape(4, -1)

>>> ar2.flatten(order = 'C') # [참고] 2차원 배열을 1차원 배열로 변환합니다.
```

2차원 배열의 인덱싱 및 슬라이싱

- 2차원 배열은 대괄호 안에 콤마를 추가하고 행/열 인덱스를 차례로 지정합니다.

```
>>> ar2[0, 0] # ar2의 1행 1열 원소를 선택합니다.
>>> ar2[1, 1] # ar2의 2행 2열 원소를 선택합니다.
```

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11

- 콜론을 이용한 슬라이싱을 실행합니다.

```
>>> ar2[0:2, 0:2] # ar2의 1~2행 1~2열 원소를 선택합니다.
>>> ar2[:, 1:3] # ar2의 전체 행 2~3열 원소를 선택합니다.
                # [주의] 반드시 빈 콜론을 추가해야 합니다.
```

- 정수 스칼라나 콜론 대신 리스트를 사용한 팬시 인덱싱도 가능합니다.

```
>>> ar2[:, [2, 1]] # ar2의 일부 열 순서를 변경합니다.
                   # [주의] 행과 열 순서를 동시에 변경하는 것은 안됩니다.
```

pandas 라이브러리

pandas 라이브러리 소개

- pandas 라이브러리의 함수를 이용하면 다음과 같은 객체를 생성할 수 있습니다.
  - 1차원 시리즈Series: 원소를 세로 방향으로 정렬한 1차원 자료구조이며, 데이터프레임의 원소(열벡터)로 사용됩니다.
  - 2차원 데이터프레임DataFrame: 1차원 시리즈를 원소로 갖는 2차원 자료구조입니다.
    - R의 데이터프레임에서 유래한 것으로 알려져 있습니다.
  - 3차원 패널Panel: 시간의 흐름에 따라 여러 개의 데이터프레임을 중첩한 것입니다.
    - 동일한 관찰 대상이 시간의 흐름에 따라 어떻게 바뀌어 가는지 확인하는 분석이 가능합니다.
- pandas 라이브러리를 호출합니다.

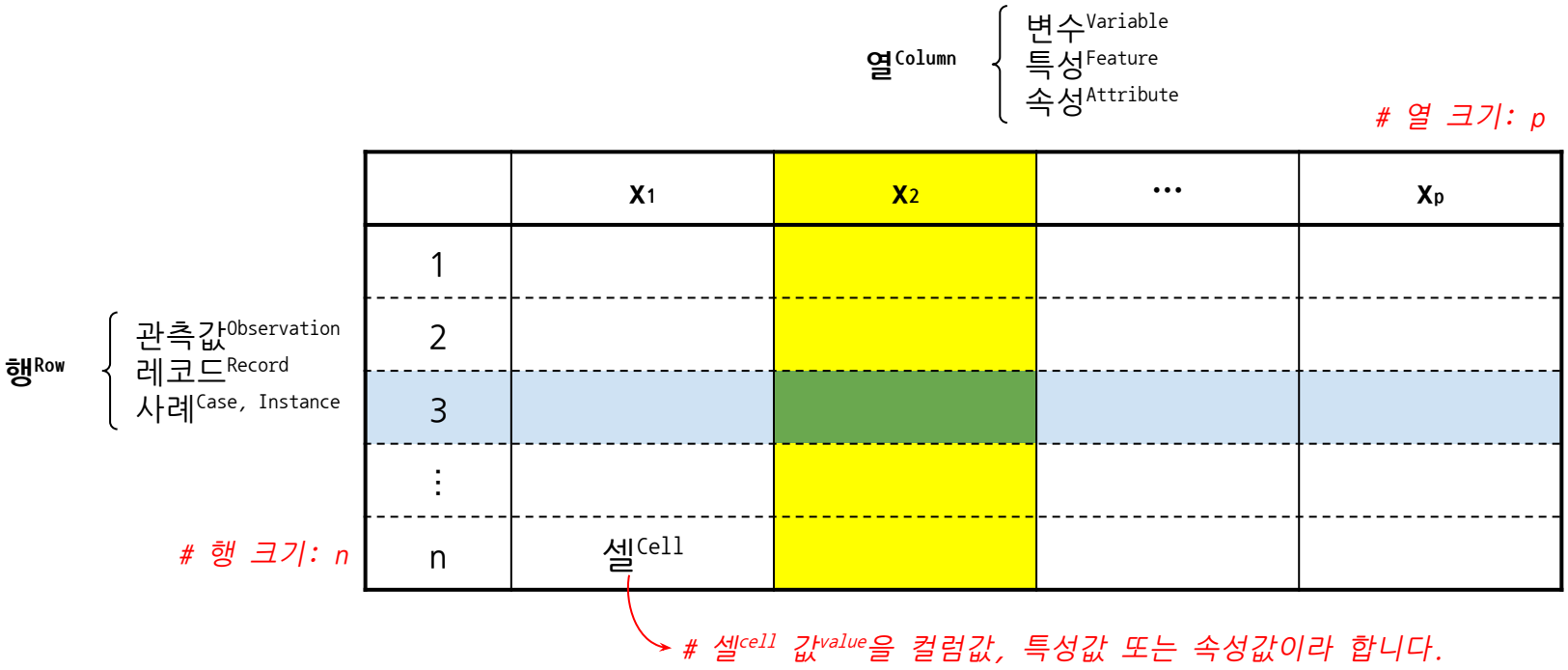
```
>>> import pandas as pd  # pandas 라이브러리를 호출하고, 'pd'로 사용하도록 설정합니다.
```

[참고] 데이터프레임의 시각적 예시

- 데이터프레임은 행row과 열column을 갖는 2차원 자료구조입니다.
  - 열별 자료형이 다를 수 있으며, 행 또는 열을 하나만 선택하면 시리즈로 반환됩니다.
  - 2차원 배열은 전체 원소의 자료형이 같다는 점에서 데이터프레임과 다릅니다.

문자열	문자열	문자열	정수형	실수형	
고객ID	**고객명**	**성별**	**나이**	**키**	**…**
0001	정우성	M	48	186.9	…
0002	장동건	M	49	182.2	…
0003	김태희	F	41	163.4	…
⋮	⋮	⋮	⋮	⋮	…

[참고] 데이터프레임의 명칭



시리즈 생성

- 시리즈는 배열, 리스트 및 딕셔너리로 생성하고 인덱스를 가집니다.

```
>>> sr = pd.Series(data = np.arange(start = 1, stop = 10, step = 2))
>>> sr  # sr을 출력합니다.
       # 인덱스index와 값value이 세로 방향으로 출력되고, 맨 아래에 원소의 자료형dtype이 추가됩니다.
>>> type(sr) # sr의 클래스를 확인합니다. sr의 클래스는 pandas.core.series.Series입니다.
```

- [참고] 시리즈를 생성하는 다른 방법을 소개합니다.

```
>>> pd.Series(data = [1, 3, 5, 7, 9], index = ['a', 'b', 'c', 'd', 'e'])
>>> pd.Series(data = {'a': 1, 'b': 3, 'c': 5, 'd': 7, 'e': 9}) # key가 인덱스로 자동
                                                               # 설정됩니다.
```

시리즈 확인

- 시리즈의 형태, 값, 인덱스명 및 자료형을 차례대로 확인합니다.

```
>>> sr.shape # sr의 형태를 출력합니다. sr은 1차원이며 원소 개수는 5입니다.

>>> sr.values # sr의 값을 출력합니다.

>>> sr.dtype # sr의 자료형을 출력합니다.

>>> sr.index # sr의 인덱스명 출력합니다.
```

- 시리즈의 인덱스명을 변경합니다.

```
>>> sr.index = ['a', 'b', 'c', 'd', 'e'] # 원소 개수와 같은 길이의 리스트를 지정해야 합니다!
                                         # [참고] 인덱스명은 중복이 가능합니다!

>>> sr # sr을 출력하면 인덱스명이 변경된 것을 확인할 수 있습니다.
```

시리즈의 인덱싱 및 슬라이싱: iloc 인덱서

- iloc[integer location] 인덱서는 정수 인덱스를 사용하여 원소를 선택합니다.

```
>>> sr.iloc[0] # 대괄호 안에 정수 인덱스 스칼라를 지정하면 해당 원소를 반환합니다.

>>> sr.iloc[[0]] # 대괄호 안에 스칼라 대신 리스트를 지정하면 해당 원소를 시리즈로 반환합니다.
```

- 대괄호 안에 지정된 배열/리스트 원소인 정수 인덱스에 해당하는 원소를 선택하는 방식을 팬시 인덱싱[Fancy Indexing]이라고 합니다.

```
>>> sr.iloc[[1, 3]] # 연속하지 않는 인덱스를 지정할 때 반드시 리스트를 사용해야 합니다.
                    # 왼쪽 코드는 sr의 1, 3번 인덱스 원소를 시리즈로 반환합니다.
```

- 콜론을 사용하여 연속된 원소를 시리즈로 반환합니다.

```
>>> sr.iloc[1:3] # sr의 1~2번 인덱스 원소를 시리즈로 반환합니다.
```

시리즈의 인덱싱 및 슬라이싱: loc 인덱서

- loc[location] 인덱서는 정수 대신 인덱스명을 사용하여 원소를 선택합니다.

```
>>> sr.loc['a'] # 대괄호 안에 인덱스명 스칼라를 지정하면 해당 원소를 반환합니다.

>>> sr.loc[['a']] # 대괄호 안에 스칼라 대신 리스트를 지정하면 해당 원소를 시리즈로 반환합니다.
```

- 인덱스명을 리스트로 지정하면 순서대로 원소를 선택하여 시리즈로 반환합니다.

```
>>> sr.loc[['b', 'd']] # sr의 인덱스명이 'b', 'd'인 원소를 시리즈로 반환합니다.
```

- 인덱스명에 콜론을 이용하여 슬라이싱하면 시리즈로 반환합니다.

```
>>> sr.loc['b':'d'] # sr의 인덱스명이 'b', 'c', 'd'인 원소를 시리즈로 반환합니다.
                    # [주의] loc 인덱서는 콜론의 마지막 인덱스명을 포함한다는 점에 유의하시기 바랍니다.
```

시리즈의 인덱싱 및 슬라이싱: loc 인덱서(계속)

- 시리즈로 비교 연산을 실행하면, True/False를 원소로 갖는 시리즈가 반환됩니다.

```
>>> sr >= 5 # sr로 비교 연산을 실행합니다. 코드 실행 결과로 True/False를 원소로 갖는 시리즈가 반환됩니다.
```

- 대괄호 안에 비교 연산 코드로 반환되는 True의 위치에 해당하는 원소를 선택하는 방식을 불리언 인덱싱Boolean Indexing이라고 합니다.

```
>>> sr.loc[sr >= 5] # 대괄호 안에 비교 연산 코드를 지정하면 True에 해당하는 원소만 선택합니다.

>>> sr.iloc[sr >= 5] # [주의] iloc 인덱서를 사용하면 에러가 발생합니다.

>>> sr.loc[(sr >= 5) & (sr <= 7)] # 두 개 이상의 비교 연산 결과로 불리언 인덱싱을 실행합니다.
                                  # [주의] 반드시 개별 조건을 소괄호로 묶어주어야 합니다.

>>> sr[(sr >= 5) & (sr <= 7)] # [참고] loc 인덱서를 생략해도 실행됩니다.
```

데이터프레임 생성

- 데이터프레임은 2차원 배열, 리스트 및 딕셔너리로 생성하며, 행이름(인덱스) 및 열이름(컬럼명)을 가집니다.
- 리스트로 데이터프레임을 생성하고 클래스를 확인합니다.

```
>>> df = pd.DataFrame(data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]], # 리스트의 각 원소가 행이 됩니다.
                      index = [1, 2, 3], # 행이름(인덱스명)을 지정합니다.
                      columns = ['A반', 'B반', 'C반']) # 열이름(컬럼명)을 지정합니다.
>>> df # df를 출력합니다.
>>> type(df) # df의 클래스를 확인합니다. df의 클래스는 pandas.core.frame.DataFrame입니다.
```

[참고] 데이터프레임을 생성하는 다른 방법 소개

- 딕셔너리로 데이터프레임을 생성합니다.

```
>>> pd.DataFrame(data = {'A반': [1, 2, 3], # 딕셔너리의 key는 열이름, value는 열이 됩니다.
                         'B반': [4, 5, 6],
                         'C반': [7, 8, 9]})
```

- 딕셔너리를 원소로 갖는 리스트로 데이터프레임을 생성합니다.

```
>>> pd.DataFrame(data = [{'A반': 1, 'B반': 2, 'C반': 3}, # 리스트의 각 원소는 행으로 입력됩니다.
                         {'A반': 4, 'B반': 5, 'C반': 6},
                         {'A반': 7, 'B반': 8, 'C반': 9}])
```

데이터프레임 확인

- 데이터프레임의 정보를 확인합니다.

```
>>> df.shape # df의 형태를 튜플로 출력합니다.(행 길이, 열 길이)

>>> df.values # df의 값을 2차원 배열로 출력합니다.

>>> df.dtypes # df의 열별 자료형을 출력합니다.

>>> df.index # df의 행이름(인덱스명)을 출력합니다.

>>> df.columns # df의 열이름(컬럼명)을 출력합니다.

>>> df.info() # df의 정보를 출력합니다.
              (행 길이, 열 길이, 행이름, 열이름, 자료형, 열별 결측값을 제외한 원소 개수 등)
```

데이터 입출력

작업경로 확인 및 변경

- 관련 라이브러리를 호출합니다.

```
>>> import os, chardet  # chardet는 바이너리 문자열의 인코딩 방식을 확인하는 함수를 포함하고 있습니다.
>>> import numpy as np
>>> import pandas as pd
```

- 현재 작업경로 확인 및 data 폴더로 작업경로를 변경하고, 파일명을 출력합니다.

```
>>> os.getcwd()
>>> os.chdir('../data')
>>> os.listdir()
```

xlsx 파일 입출력

- xlsx 파일을 읽고 데이터프레임을 생성합니다.

```
>>> df = pd.read_excel(io = 'KBO_Hitters_2020.xlsx',  # 만약 에러가 발생한다면 openpyxl
                                                      라이브러리를 설치해보세요.
                       sheet_name = 'KBO2020',  # 한 시트만 읽을 수 있으므로, 시트의 인덱스
                                                  또는 시트명을 지정합니다.(기본값: 0)
                       skiprows = 4) # 열이름 위로 생략할 행 길이를 지정합니다.
>>> df.head() # df의 처음 5행을 출력합니다. 괄호 안에 입력하는 숫자만큼 출력합니다.(기본값: 5)
              # tail() 함수는 마지막 5행을 출력합니다.
>>> df.index = range(1, 297) # df의 인덱스를 1부터 시작하도록 변경합니다.
```

- 데이터프레임을 xlsx 파일로 저장합니다.

```
>>> df.to_excel(excel_writer = 'test.xlsx', index = None)
```

[참고] 프로야구 타자 스탯 데이터 간단 설명

구분	상세 내용
타석/타수	타수는 타석에서 볼넷, 사구, 희생타 등을 제외한 것
안타	1루타, 2루타, 3루타, 홈런을 모두 합한 개수
득점/타점	홈을 밟으면 득점, 안타를 쳐서 다른 선수가 득점하면 타점
볼넷/삼진	볼 네 개면 1루에 걸어가는 볼넷, 스트라이크 세 개는 삼진 아웃
BABIP	(안타-홈런)÷(타수-삼진-홈런-희생타) *# 인플레이 타율*
타율	안타÷타수 *# 타율 3할은 우수한 타자의 기준*
출루율	(안타+볼넷+사구)÷(타수+볼넷+사구+희생타) *# 출루율 4할이 우수*
장타율	(1루타+2루타×2+3루타×3+홈런×4)÷타수 *# 장타율 5할이 우수*
OPS	출루율+장타율 *# OPS 0.9 이상이면 리그 수위타자로 인정*
WAR	대체 선수 대비 승리 기여 *# WAR 4 이상이면 핵심 선수로 인정*

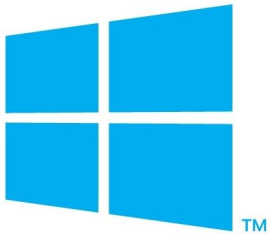
 # 출처: http://blog.daum.net/dr_rafael/5012

[참고] 한글 인코딩이란?

- 컴퓨터는 사람들이 사용하는 문자(자연어)를 이해할 수 없습니다.
  - 사람이 사용하는 자연어를 컴퓨터가 이해할 수 있는 16진수로 표기한 것이 인코딩입니다. 반대로 사람은 16진수로 된 문자를 빠른 속도로 정확하게 읽는 것이 어렵습니다.
  - 초창기에는 자연어를 0과 1의 2진수 코드로 변환하였지만 나중에 16진수로 발전했습니다.
- 한글에 사용되는 인코딩 방식에는 EUC-KR, CP949 및 UTF-8 등이 있습니다.

사람이 이해하는 문자	한글 인코딩 방식	컴퓨터가 이해하는 코드
'가'	CP949(EUC-KR)	B0A1
	UTF-8	U+AC00

[참고] 컴퓨터 운영체제별 한글 인코딩 방식



컴퓨터
운영체제

CP949
(EUC-KR)

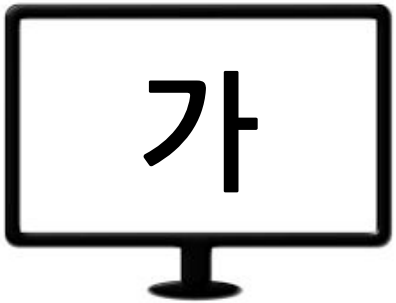
기본 한글
인코딩 방식

B0A1

문자코드
(컴퓨터가 읽을 수 있음)

Decoding

Encoding

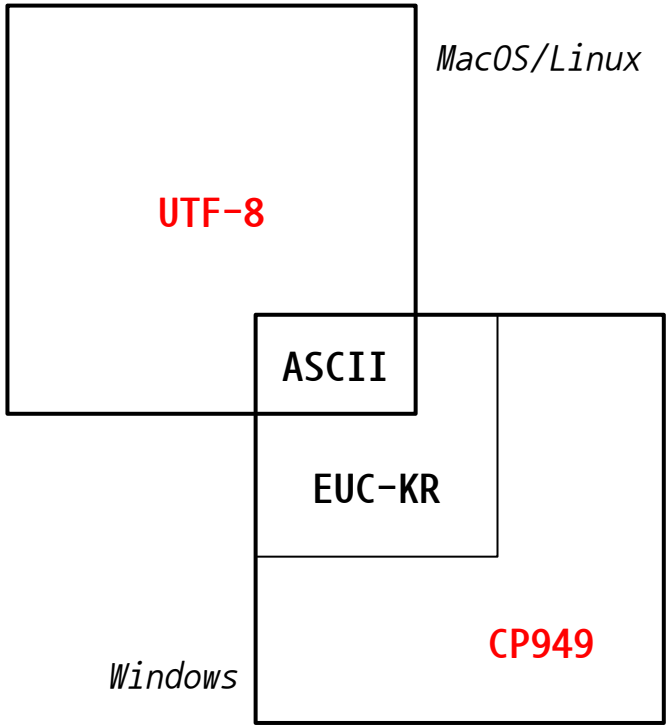


화면에 보이는 문자
(사람이 읽을 수 있음)

UTF-8

U+AC00

[참고] 한글 인코딩 방식 관계도



구분	설명
ASCII	• 미국에서 개발된 대표적인 영문 인코딩 방식입니다. • 알파벳, 숫자, 기호 등 128개가 지정되어 있습니다.
UTF-8	• 유니코드에 기반한 인코딩 방식입니다.(가변 길이) • **한글 완성형/조합형 모두 포함합니다.(국제 표준)**
Unicode	• 전 세계 모든 문자를 포함하는 인코딩 방식입니다. • 한 글자를 나타내기 위해 4 bytes를 사용합니다.
EUC-KR	• 한글 초기 완성형 인코딩 방식입니다.(2350자) • 조합형은 다른 문자 체계와 호환이 되지 않습니다.
CP949	• 8822자를 추가한 통합 완성형 인코딩 방식입니다. • **Windows 점유율 높은 한국에서 사실상 표준입니다.**

csv 파일 인코딩 방식 확인 및 입출력

- csv 파일을 바이너리 모드로 읽고, 문자 인코딩 방식을 확인합니다.

```
>>> raw = open(file = 'KBO_Hitters_2020.csv', mode = 'rb').read()
>>> chardet.detect(raw) # csv 파일의 문자 인코딩 방식을 확인합니다.
```

- csv 파일을 읽고 데이터프레임을 생성합니다.

```
>>> df1 = pd.read_csv(filepath_or_buffer = 'KBO_Hitters_2020.csv',
                      encoding = 'EUC-KR') # [주의] csv 파일의 인코딩 방식이 'UTF-8'이 아니면
                                             인코딩 방식을 지정해주어야 합니다!
```

- 데이터프레임을 csv 파일로 저장합니다.

```
>>> df1.to_csv(path_or_buf = 'test.csv', index = None, encoding = 'UTF-8')
```

데이터프레임 인덱싱 및 슬라이싱: iloc 인덱서

- 데이터프레임은 iloc 인덱서에 정수 인덱스로 행과 열을 선택합니다.

```
>>> df.iloc[0] # 대괄호 안에 정수 인덱스 스칼라를 지정하면 해당 행을 시리즈로 반환합니다.
               # 데이터프레임은 대괄호 안에 콤마를 생략하면 행을 선택합니다.

>>> df.iloc[[0]] # 대괄호 안에 스칼라 대신 리스트를 지정하면 해당 행을 데이터프레임으로 반환합니다.

>>> df.iloc[0:10] # 대괄호 안에 콜론을 사용하여 슬라이싱하면 해당 행을 데이터프레임으로 반환합니다.
                  # [주의] 마지막 인덱스를 포함하지 않습니다.

>>> df.iloc[0:10, :] # 대괄호 안에 콤마를 추가하고, 콤마 뒤에 선택할 열의 정수 인덱스를 지정합니다.
                     # 전체 열을 선택하려면 콤마 오른쪽에 빈 콜론을 추가합니다.

>>> df.iloc[0:10, 0:12] # 행 인덱스가 0~9인 행과 열 인덱스가 0~11인 열을 데이터프레임으로 반환합니다.

>>> df.iloc[:, [1, 0, 4, 5]] # 대괄호 안에 열의 인덱스를 원하는 순서대로 설정할 수 있습니다.
```

데이터프레임 인덱싱 및 슬라이싱: loc 인덱서

- 데이터프레임은 loc 인덱서에 인덱스명과 컬럼명으로 행과 열을 선택합니다.

```
>>> df.loc[1] # 대괄호 안에 인덱스명 스칼라를 지정하면 해당 행을 시리즈로 반환합니다.
# [주의] 행이름에 0이 없으므로, 0을 지정하면 에러가 발생합니다.

>>> df.loc[[1]] # 대괄호 안에 스칼라 대신 리스트를 지정하면 해당 행을 데이터프레임으로 반환합니다.

>>> df.loc[1:10] # 대괄호 안에 콜론을 사용하여 슬라이싱하면 해당 행을 데이터프레임으로 반환합니다.
# [주의] 마지막 인덱스명을 포함합니다.

>>> df.loc[1:10, :] # 대괄호 안에 콤마를 추가하고, 콤마 뒤에 선택할 컬럼명을 지정합니다.
# 전체 열을 선택하려면 콤마 오른쪽에 빈 콜론을 추가합니다.

>>> df.loc[1:10, '선수명':'도루'] # 인덱스명이 1~10인 행과 컬럼명이 선수명~도루인 열을 데이터
프레임으로 반환합니다.

>>> df.loc[:, ['팀명', '선수명', '타수', '안타']] # 대괄호 안에 컬럼명을 원하는 순서대로
설정할 수 있습니다.
```

[참고] 인덱싱 방법에 따른 결과 비교

- 리스트, 배열, 시리즈 및 데이터프레임은 인덱싱 방법에 따라 반환되는 값이 서로 다르므로 아래 표로 정리된 내용을 숙지하시기 바랍니다.

구분	[정수 스칼라]	[콜론]	[리스트]
리스트	원소	리스트	불가능
배열	원소	배열	배열
시리즈	원소	시리즈	시리즈
데이터프레임	시리즈	데이터프레임	데이터프레임

데이터프레임 전처리

실습 데이터셋 소개

- 공공데이터포털에서 제공하는 '국토교통부 아파트매매 실거래자료'에서 2020년에 서울특별시에서 거래된 가격 데이터를 2가지 형태로 제공합니다.
  - 'APT_Price_Seoul_2020.csv'
  - 'APT_Price_Seoul_2020.xlsx'
- 국내 대형 포털에서 제공하는 서울특별시 소재 아파트단지별 상세정보를 수집하여 2가지 형태로 제공합니다.
  - 'Naver_APT_Detail.csv'
  - 'Naver_APT_Detail.xlsx'

실습 데이터셋 준비

- 관련 라이브러리를 호출합니다.

```
>>> import os, chardet
>>> import numpy as np
>>> import pandas as pd
```

- 현재 작업경로 확인 및 data 폴더로 작업경로를 변경하고, 파일명을 출력합니다.

```
>>> os.getcwd()
>>> os.chdir(path = '../data')
>>> os.listdir()
```

실습 데이터셋 준비(계속)

- 가격 xlsx 파일을 읽고 데이터프레임으로 생성합니다.

```
>>> price = pd.read_excel(io = 'APT_Price_Seoul_2020.xlsx') # 약 15초 정도 소요됩니다.
>>> price.head() # price의 처음 5행을 출력합니다.
>>> price.info() # price의 정보를 출력합니다.
# 거래일의 자료형이 datetime64입니다. xlsx 파일은 날짜를 날짜형datetime64으로 생성합니다.
```

```
# 코드 앞에 %time을 추가하면
  코드 실행 시간을 반환합니다.
```

- 상세정보 xlsx 파일을 읽고 데이터프레임으로 생성합니다.

```
>>> detail = pd.read_excel(io = 'Naver_APT_Detail.xlsx') # 약 1.3초 정도 소요됩니다.
>>> detail.head() # detail의 처음 5행을 출력합니다.
>>> detail.info() # detail의 정보를 출력합니다.
```

[참고] csv 파일 읽을 때 날짜형으로 생성하는 방법

- csv 파일을 데이터프레임으로 읽고, 열별 자료형을 확인합니다.

```
>>> raw = open(file = 'APT_Price_Seoul_2020.csv', mode = 'rb').read()
>>> chardet.detect(raw[:100]) # 텍스트 용량이 크면 오래 걸리므로, 일부만 확인합니다.
                              # [참고] 인코딩 방식이 'UTF-8'이면 추가하지 않아도 됩니다.
>>> price = pd.read_csv(filepath_or_buffer = 'APT_Price_Seoul_2020.csv')
>>> price.info() # 거래일의 자료형이 object입니다. csv 파일은 날짜를 문자열로 읽습니다.
```

- parse_dates 매개변수에 날짜형으로 읽을 열이름을 리스트로 지정합니다.

```
>>> price = pd.read_csv('APT_Price_Seoul_2020.csv', parse_dates = ['거래일'])
>>> price.info() # 거래일의 자료형이 datetime64로 변경되었습니다.
```

데이터프레임 전처리 항목

- 필요한 열을 선택합니다.
- 불필요한 열을 삭제합니다.
- 열이름을 변경합니다.
- 조건에 맞는 행을 선택합니다.
- 불필요한 행을 삭제하고, 행이름을 초기화합니다.
- 결측값을 삭제하거나 대체합니다.
- 열의 자료형을 변경합니다.

데이터프레임 전처리 항목(계속)

- 새로운 파생변수를 생성합니다.
- 그룹을 설정하고, 집계함수를 사용하여 데이터를 요약합니다.
- 오름차순 또는 내림차순으로 정렬합니다.
- 데이터프레임의 구조를 변환합니다.(Long type ↔ Wide type)
- 두 개의 데이터프레임을 행 또는 열 방향으로 결합합니다.
- 데이터프레임의 특정 열 기준 중복 여부를 확인하고 필요시 삭제합니다.
- 두 개의 데이터프레임에서 서로 일치하는 행을 병합합니다.

열 선택select

- 데이터프레임의 열을 선택할 때, 딕셔너리 인덱싱처럼 열이름을 지정합니다.

```
>>> price['거래금액'] # 열이름을 문자열로 지정하면 시리즈로 반환합니다.
                      # 데이터프레임.열이름 방식으로도 하나의 열을 선택할 수 있습니다.
>>> price[['거래금액']] # 열이름을 리스트로 지정하면 데이터프레임으로 반환합니다.
>>> price[['거래일', '거래금액']] # 여러 열을 선택하려면 항상 리스트로 지정해야 합니다.
```

- 열이름에 콜론을 사용하려면 loc 인덱서를 사용해야 합니다.(팬시 인덱싱)

```
>>> price.loc[:, '거래일':'거래금액'] # [주의] loc 인덱서를 생략하면 에러가 발생합니다.
```

- 조건을 만족하는 열을 선택할 수 있습니다.(불리언 인덱싱)

```
>>> price.loc[:, price.dtypes == 'int64'] # 자료형이 정수인 열만 선택합니다.
```

열 삭제drop

- 삭제할 열이름을 **drop**() 함수에 리스트로 지정합니다.

```
>>> price.drop(labels = ['일련번호'], axis = 1) # [주의] 열을 삭제할 때, 반드시 axis = 1을 추가해야 합니다.
>>> price.head() # price에는 일련번호가 여전히 포함되어 있습니다.
```

- 열을 삭제한 결과를 price에 재할당합니다.

```
>>> price = price.drop(labels = ['일련번호'], axis = 1)
>>> price.head() # price에서 일련번호가 삭제되었습니다.
```

열이름 변경rename

- price의 열이름을 출력합니다.

```
>>> price.columns
```

- **rename**() 함수로 일부 열이름을 변경합니다.

```
>>> price.rename(columns = {'시도명': '시도', '시군구': '자치구'})
```

 # 기존 열이름, 변경할 열이름 순으로 입력합니다.
 # [주의] 매개변수 columns를 생략하면 안됩니다!

- columns 속성을 사용하면 열이름 전체를 변경합니다.

```
>>> price.columns = ['아파트', '시도', '자치구', '읍면동', '지번',
                     '거래일', '전용면적', '층', '거래금액']
>>> price.head()
```

 # [주의] 데이터프레임의 열이름 개수와 같은 길이의 리스트를 지정해야 합니다.

조건에 맞는 행 선택filtering: 연속형 변수

- 거래금액이 70억 이상인 행을 선택하여 df1에 할당합니다.

```
>>> df1 = price[price['거래금액'] >= 700000].copy()
>>> df1.head()
```

 # 불리언 인덱싱으로 행만 선택할 때는 loc 인덱서를 생략해도 됩니다.

*# 원본에서 새로운 데이터프레임을 생성할 때는 **copy**() 방식을 추가합니다.*

 # df1의 처음 5행을 출력합니다.
 # 행이름이 연속이 아닌 것을 확인할 수 있습니다.

- 거래금액이 70억 미만이고, 60층 이상인 행을 선택하여 df2에 할당합니다.

```
>>> df2 = price[(price['거래금액'] < 700000) & (price['층'] >= 60)].copy()
>>> df2.head()
```

 # 두 조건을 동시에 만족하는 행을 선택하려면 & 기호를 사용합니다.[논리곱]
조건을 하나 이상 만족하는 행을 선택하려면 | 기호를 사용합니다.[논리합]
조건을 만족하지 않는 행을 선택하려면 ~ 기호를 사용합니다.[논리부정]

 # [주의] 비트 연산자 앞뒤 코드를 반드시 소괄호로 묶어주어야 합니다.

[참고] 시리즈의 비교 연산 함수 소개

- 시리즈에서 비교 연산을 실행하는 함수를 소개합니다.

```
>>> price['층'].gt(60).sum()  # 층이 60 초과일 때 True, 아닐 때 False를 갖는 시리즈를 반환합니다.
                              # 마지막에 추가한 sum() 함수는 True의 개수를 반환합니다.
>>> price['층'].ge(60).sum()  # 층이 60 이상일 때 True, 아닐 때 False를 갖는 시리즈를 반환합니다.
>>> price['층'].lt(60).sum()  # 층이 60 미만일 때 True, 아닐 때 False를 갖는 시리즈를 반환합니다.
>>> price['층'].le(60).sum()  # 층이 60 이하일 때 True, 아닐 때 False를 갖는 시리즈를 반환합니다.
>>> price['층'].eq(60).sum()  # 층이 60일 때 True, 60일 아닐 때 False를 갖는 시리즈를 반환합니다.
>>> price['층'].ne(60).sum()  # 층이 60이 아닐 때 True, 60일 때 False를 갖는 시리즈를 반환합니다.
```

조건에 맞는 행 선택filtering: 범주형 변수

- 자치구가 '강남구'인 행을 선택합니다.

```
>>> price[price['자치구'].eq('강남구')]
```

- 자치구가 '강남구' 또는 '서초구'인 행을 선택합니다.

```
>>> price[price['자치구'].eq('강남구') | price['자치구'].eq('서초구')]
```

- 자치구에서 패턴을 포함하면 True, 아니면 False인 시리즈를 반환합니다.

```
>>> price['자치구'].str.contains(pat = '강남|서초|송파')
```

- 자치구에서 패턴을 포함하는 행을 선택합니다.

```
>>> price[price['자치구'].str.contains(pat = '강남|서초|송파')]
```

[참고] 시리즈를 문자열로 처리하는 주요 함수 소개

- 시리즈를 문자열로 처리할 때 str을 추가합니다.

```
>>> drID = pd.Series(data = ['서울 00-123456-01 ', ' 경기 01-654321-02'])
>>> drID.str.strip() # 각 원소의 양 옆에 있는 공백을 삭제한 결과를 출력합니다.
>>> drID.str.split(pat = ' |-', expand = True) # 각 원소를 지정된 구분자로 분리한 다음 데이터프레임으로 반환합니다.
>>> drID.str.find(sub = '서울') # 각 원소에서 지정된 문자열이 시작하는 인덱스를 반환합니다.
                                # 지정된 문자열이 없으면 -1을 반환합니다.
>>> drID.str.replace(pat = ' ', repl = '') # 각 원소에서 지정된 문자열 패턴을 변경합니다.
>>> drID.str.slice(start = 0, stop = 3) # 각 원소에서 지정된 인덱스로 문자열을 잘라냅니다.
>>> drID.str.extract(pat = r'([가-힣]+)') # 각 원소에서 지정된 정규표현식 패턴에 해당하는 문자열을 추출합니다. 괄호로 묶어주어야 합니다.
```

행이름으로 행 삭제 및 행이름 초기화reset index

- **drop**() 함수에 행이름을 스칼라 또는 리스트로 입력하여 행을 삭제합니다.

```
>>> df1  # df1을 출력합니다. 행이름(인덱스명)이 0부터 시작하지 않습니다.

>>> df1.drop(labels = [21625, 59241])  # 행을 삭제할 때 axis = 0을 생략할 수 있습니다.
                                       # [주의] 행이름에 없는 값을 입력하면 에러가 발생합니다.

>>> df1.drop(labels = df1.index[0:2]) # 연속된 행이름을 슬라이싱으로 삭제할 수 있습니다.
```

- df1의 행이름을 초기화합니다.

```
>>> df1.reset_index()  # 행이름을 초기화한 결과를 출력하므로, 아직 df1에 재할당하지 않았습니다.
                       # 기존 행이름이 index로 추가됩니다.

>>> df1.reset_index(drop = True) # index 열이 생성되지 않도록 합니다.

>>> df1 = df1.reset_index(drop = True) # df1의 행이름을 초기화한 결과를 df1에 재할당합니다.
```

결측값 처리: 단순대체simple imputation

- 열별 결측값 개수를 확인합니다.

```
>>> price.isna().sum() # is.na() 함수는 각 원소의 결측값 여부를 True 또는 False로 반환합니다.
                       # sum() 함수는 True 개수를 반환하므로, 열별 결측값 개수를 확인할 수 있습니다.
```

- 지번이 결측값인 행만 선택합니다.

```
>>> price[price['지번'].isna()]
```

- 데이터프레임에 포함된 결측값을 단순대체합니다.

```
>>> price.iloc[4784:4787].fillna(value = '') # 결측값을 빈 문자열로 채웁니다.

>>> price.iloc[4784:4787].fillna(method = 'ffill') # 결측값을 이전 값으로 채웁니다.

>>> price.iloc[4784:4787].fillna(method = 'bfill') # 결측값을 이후 값으로 채웁니다.
```

결측값 처리: 행 삭제

- 지번이 결측값인 행을 삭제합니다.

```
>>> price[price['지번'].notna()] # 지번이 결측값이 아닌 행을 선택합니다.
```

- price에서 결측값이 있는 모든 행을 삭제한 결과를 price에 재할당합니다.

```
>>> price = price.dropna()
>>> price = price.dropna(subset = ['지번']) # [참고] subset 매개변수에 일부 열이름을 지정하면 해당 열에서 결측값이 있는 행을 삭제합니다.
```

- price의 행 길이를 확인합니다.

```
>>> price.shape[0] # price의 행 길이가 25개 감소했습니다.(80734 -> 80709)
```

열 자료형 변환

- price의 열별 자료형을 확인합니다.

```
>>> price.dtypes
```

- 시리즈의 자료형을 변환합니다.(문자열: str, 실수: float, 정수: int)

```
>>> price['거래일'] = price['거래일'].astype('str') # 거래일을 문자열로 변환합니다.
>>> price['거래금액'] = price['거래금액'].astype('float') # 거래금액을 실수형으로 변환합니다.
```

- price의 열별 자료형을 다시 확인합니다.

```
>>> price.dtypes # 거래일이 object, 거래금액이 float64로 변환되었습니다.
```

열 자료형 변환(계속)

- 데이터프레임의 열별 자료형 변환 방법을 딕셔너리로 지정합니다.

```
>>> price = price.astype({'거래일': 'datetime64', '층': 'float'})
>>> price.dtypes # 거래일은 datetime64, 층은 float64로 변환되었습니다.
```

- 일부 열을 같은 자료형으로 동시에 변환합니다.

```
>>> cols = ['층', '거래금액'] # 정수형으로 변환할 열이름을 리스트로 지정합니다.
>>> price[cols] = price[cols].astype('int') # 선택된 열을 정수형으로 동시에 변환합니다.
```

- price의 열별 자료형을 다시 확인합니다.

```
>>> price.dtypes # 층, 거래금액 열이 모두 int64로 변환되었습니다.
```

[참고] 문자열을 날짜형으로 변환

- 날짜 기본형이 아닌 문자열로 데이터프레임을 생성합니다.

```
>>> dates = pd.DataFrame(data = ['2021년 1월 1일'], columns = ['Date'])
>>> dates.info() # dates의 정보를 확인합니다.
                 # Date의 자료형은 object입니다.
```

- Date를 날짜형으로 변환하려고 하면 에러가 발생합니다.

```
>>> dates['Date'].astype('datetime64') # [참고] 문자열이 아래 포맷일 때 정상적으로 변환됩니다.
                                       'yyyy-mm-dd', 'yyyy mm dd', 'yyyy/mm/dd', 'yyyy.mm.dd'
```

- **to_datetime**() 함수의 format 매개변수에 날짜포맷을 지정하면 변환됩니다.

```
>>> pd.to_datetime(arg = dates['Date'], format = '%Y년 %m월 %d일')
```

[참고] 날짜 분해 함수 소개

- 날짜 변수에서 년, 월, 일, 시, 분, 초 및 요일을 추출할 수 있습니다.

```
>>> price['거래일'].dt.year # 날짜 변수에서 연(year)을 추출하고, 정수형 시리즈로 반환합니다.
                            # [참고] 왼쪽 코드 실행 결과를 새로운 변수로 생성할 수 있습니다.
>>> price['거래일'].dt.month # 날짜 변수에서 월(month)을 추출하고, 정수형 시리즈로 반환합니다.
>>> price['거래일'].dt.day # 날짜 변수에서 일(day)을 추출하고, 정수형 시리즈로 반환합니다.
>>> price['거래일'].dt.hour # 날짜 변수에서 시(hour)를 추출하고, 정수형 시리즈로 반환합니다.
                            # [참고] 거래일에는 년월일 정보만 있으므로 시분초는 0으로 반환됩니다.
>>> price['거래일'].dt.minute # 날짜 변수에서 분(minute)을 추출하고, 정수형 시리즈로 반환합니다.
>>> price['거래일'].dt.second # 날짜 변수에서 초(second)를 추출하고, 정수형 시리즈로 반환합니다.
>>> price['거래일'].dt.strftime('%A') # 날짜 변수에서 요일을 문자형 시리즈로 반환합니다.
                                      >>> price['거래일'].dt.day_name('ko_KR')
```

파생변수 생성mutate: 연속형 변수

- 거래금액을 전용면적으로 나눈 연속형 파생변수를 생성합니다.

```
>>> price['단위금액'] = price['거래금액'] / price['전용면적']
>>> price.head()
```

*# [주의] 파생변수를 생성할 때는 **데이터프레임['열이름']**으로 작성해야 합니다.*
*# **price.단위금액**으로 하면 생성되지 않습니다!*

- 단위금액을 반올림하여 소수점 둘째 자리까지 남깁니다.

```
>>> price['단위금액'] = price['단위금액'].round(2)
```

- 거래금액을 만원 단위에서 억원 단위로 변경합니다.

```
>>> price['거래금액'] = price['거래금액'] / 10000
>>> price.head()
```

[참고] 데이터프레임에 열 삽입

- price에서 단위금액을 삭제한 데이터프레임 imsi를 생성합니다.

```
>>> imsi = price.drop(labels = ['단위금액'], axis = 1).copy()
>>> imsi.head()
```

- 데이터프레임의 원하는 위치에 새로운 열이름으로 시리즈를 삽입합니다.

```
>>> imsi.insert(loc = 8, column = '단위금액', value = price['단위금액'])
>>> imsi.head()
```

[참고] 시리즈 데이터 지연lagging

- 시계열 데이터는 일정 간격만큼 뒤로 미뤄야lagging 할 때가 있습니다.

```
>>> imsi['거래금액'].shift(1) # 거래금액을 한 칸 뒤로 미룬 시리즈를 생성합니다.
>>> imsi['거래금액'].shift(-1) # 음수를 지정하면 앞으로 당깁니다.
```

- 거래금액을 1일 및 2일 지연시킨 새로운 변수를 추가합니다.

```
>>> imsi['거래금액1'] = imsi['거래금액'].shift(1)
>>> imsi['거래금액2'] = imsi['거래금액'].shift(2)
>>> imsi.head()
```

[참고] 시리즈의 이동평균moving average

- 정수 1~5를 원소로 갖는 시리즈를 생성합니다.(시계열 데이터로 가정합니다.)

```
>>> nums = pd.Series(data = np.arange(1, 6)); nums
```

- nums로 3일 이동평균을 계산합니다.

```
>>> nums.rolling(window = 3).mean() # 원소가 3개 미만이면 결측값을 반환합니다.
>>> nums.rolling(window = 3, min_periods = 1).mean() # nums로 3일 이동평균을 계산할 때
                                                     # 최소 원소 개수를 1로 설정합니다.
```

- 결측값을 추가하고, 이동평균을 계산합니다.

```
>>> nums.iloc[2] = np.nan; nums  # 결측값을 추가합니다.
>>> nums.rolling(window = 3, min_periods = 1).mean() # nums로 3일 이동평균을 계산합니다.
                                                     # 결측값은 평균 계산에서 제외됩니다.
```

파생변수 생성mutate: 범주형 변수

- 연속형 변수를 특정 구간으로 나누어 범주형 변수를 생성합니다.
- 단위금액이 1000 이상인 행에 '1천 이상'인 값을 갖는 금액구분 열을 생성합니다.

```
>>> price.loc[price['단위금액'].ge(1000), '금액구분'] = '1천 이상'
>>> price.head() # 금액구분은 단위금액이 1000 이상이면 '1천 이상', 나머지는 NaN의 값을 가집니다.
```

- 단위금액이 1000 미만인 행의 금액구분 열에 '1천 미만'인 값을 추가합니다.

```
>>> price.loc[price['단위금액'].lt(1000), '금액구분'] = '1천 미만'
>>> price.head()
```

[참고] 연속형 변수의 구간화binning 관련 함수 소개

- 조건 만족 여부에 따라 두 가지 값을 갖는 범주형 변수를 생성합니다.

```
>>> np.where(price['단위금액'].ge(1000), '1천 이상', '1천 미만')
```

- 조건 만족 여부에 따라 세 가지 이상의 값을 갖는 범주형 변수를 생성합니다.

```
>>> cond = [(price['단위금액'].ge(2000)),  # 세 가지 이상의 조건을 리스트로 생성합니다.
                                           # [주의] 반드시 조건을 소괄호로 묶어주어야 합니다.
            (price['단위금액'].lt(2000) & price['단위금액'].ge(1000)),
            (price['단위금액'].lt(1000))]
>>> value = ['2천 이상', '1천 이상', '1천 미만']  # 조건에 따라 반환할 값을 리스트로 생성합니다.
>>> np.select(condlist = cond, choicelist = value) # 조건에 맞는 값을 반환합니다.
```

파생변수 생성mutate: 문자형 변수 결합

- 여러 개의 문자형 변수를 결합하여 주소를 생성합니다.

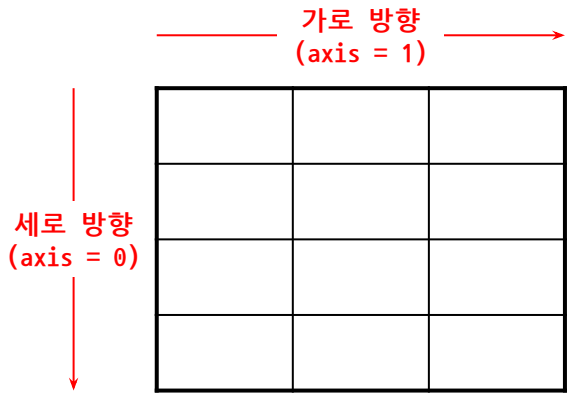
```
>>> price['주소'] = price['시도'] + ' ' + price['자치구'] + ' ' + \
                    price['읍면동'] + ' ' + price['지번']
>>> price.head()
```

- 데이터프레임의 행별 원소를 하나의 문자열로 결합하는 함수를 반복 실행합니다.

```
>>> cols = ['시도', '자치구', '읍면동', '지번']  # 문자열로 결합할 문자형 열이름을 리스트로 설정합니다.
>>> price[cols].apply(func = lambda x: ' '.join(x), axis = 1)  # 행별 원소를 하나의 문자열로 결합합니다.
```

[참고] 같은 함수 반복 실행apply 함수 소개

- 시리즈 및 데이터프레임의 여러 원소에 대하여 지정된 함수를 반복 실행해야 하는 경우에는 **map()**, **apply()**, **applymap()** 함수를 사용합니다.
  - **map()** 함수는 시리즈의 원소별로 지정된 함수를 반복 실행합니다.
  - **apply()** 함수는 데이터프레임의 행 또는 열별로 시리즈를 선택하여 반복 실행합니다.
    - 행별로 시리즈를 선택하는 것은 가로 방향이므로 **apply()** 함수 안에 axis = 1을 추가합니다.
    - 열별로 시리즈를 선택하는 것은 세로 방향하므로 **apply()** 함수 안에 axis = 0을 추가합니다.
  - **applymap()** 함수는 데이터프레임의 셀값별로 지정된 함수를 반복 실행합니다.



[참고] 같은 함수 반복 실행 실습

- **map**() 함수는 시리즈의 원소별로 지정된 함수를 반복 실행합니다.

```
>>> price['아파트'].map(arg = len) # 아파트의 원소(문자열)별 글자수를 반환합니다.
```

- **apply**() 함수는 데이터프레임의 행 또는 열별로 지정된 함수를 반복 실행합니다.

```
>>> price[cols].apply(func = len, axis = 0) # 데이터프레임의 열(시리즈)별 원소 개수를 반환합니다.
>>> price[cols].apply(func = len, axis = 1) # 데이터프레임의 행(시리즈)별 원소 개수를 반환합니다.
```

- **applymap**() 함수는 데이터프레임의 셀값별로 지정된 함수를 반복 실행합니다.

```
>>> price[cols].applymap(func = len) # 데이터프레임의 셀값(문자열)별 길이를 반환합니다.
```

그룹 설정 및 집계함수로 데이터 요약summarize

- **groupby**() 함수에 그룹핑할 열이름을 설정하고, 집계함수로 데이터를 요약합니다.

```
>>> price.groupby(by = ['자치구']).count()['거래금액']  # 자치구별 결측값 아닌 거래금액 빈도수를 시리즈로 반환합니다.
>>> price.groupby(by = ['자치구']).mean()['거래금액']  # 자치구별 거래금액의 평균을 시리즈로 반환합니다.
>>> price.groupby(by = ['자치구']).std()['거래금액']  # 자치구별 거래금액의 표준편차를 시리즈로 반환합니다.
>>> price.groupby(by = ['자치구']).min()['거래금액']  # 자치구별 거래금액의 최솟값을 시리즈로 반환합니다.
>>> price.groupby(by = ['자치구']).max()['거래금액']  # 자치구별 거래금액의 최댓값을 시리즈로 반환합니다.
>>> price.groupby(by = ['자치구']).describe()['거래금액']  # 자치구별 거래금액의 다양한 기술통계량을 반환합니다.
```

[참고] 범주형 변수의 빈도수 확인 함수 소개

- 데이터프레임의 변수인 시리즈의 범주별 빈도수를 반환합니다.

```
>>> price['자치구'].value_counts() # 자치구의 범주별 빈도수를 내림차순으로 반환합니다.
>>> price['자치구'].value_counts().sort_index() # 시리즈 인덱스를 오름차순으로 반환합니다.
>>> price['자치구'].value_counts(ascending = True) # 빈도수를 오름차순으로 반환합니다.
>>> price['자치구'].value_counts(normalize = True) # 빈도수 대신 상대도수를 반환합니다.
```

- 데이터프레임의 여러 변수 기준으로 빈도수를 반환합니다.

```
>>> cols = ['자치구', '금액구분']
>>> price.value_counts(subset = cols).sort_index().reset_index()
```

데이터프레임 정렬arrange

- 시리즈를 오름차순 또는 내림차순으로 정렬합니다.

```
>>> price['거래금액'].sort_values()  # 거래금액(시리즈)을 오름차순으로 정렬합니다.
                                     # ascending = False를 추가하면 내림차순으로 정렬합니다.
```

- 데이터프레임의 특정 열 기준으로 오름차순 또는 내림차순으로 정렬합니다.

```
>>> price.sort_values(by = ['거래금액'])

>>> price.sort_values(by = ['거래금액'], ascending = False)
```

- 두 개 이상의 열 기준으로 정렬할 때, 열별로 방향을 설정할 수 있습니다.

```
>>> price.sort_values(by = ['층', '거래금액'], ascending = False)

>>> price.sort_values(by = ['층', '거래금액'], ascending = [False, True])
```

데이터프레임의 2가지 형태

- 데이터프레임은 Long type과 Wide type으로 구분할 수 있습니다.
  - 데이터 분석 과정에서 Wide type이 주로 사용됩니다.
  - 하지만 데이터를 요약하거나 준비된 데이터셋이 Long type인 경우가 있습니다.
  - 그래프를 그릴 때 Long type으로 변환해야 할 필요도 있습니다.
  - 따라서 형태를 Wide type과 Long type으로 상호 변환하는 방법을 알고 있어야 합니다.
- 데이터프레임의 형태를 변환할 때, **pivot**() 및 **melt**() 방식이 사용됩니다.
  - **pivot**() 방식은 Long type을 Wide type으로 변환합니다.
  - **melt**() 방식은 Wide type을 Long type으로 변환합니다.

데이터프레임의 형태 비교

자치구	금액구분	건수
강남구	1천 미만	198
강남구	1천 이상	3417
강동구	1천 미만	1564
강동구	1천 이상	2535
강북구	1천 미만	1738
강북구	1천 이상	354
⋮	⋮	⋮

[Long type]

elong.pivot()

widen.melt()

자치구	1천 미만	1천 이상
강남구	198	3417
강동구	1564	2535
강북구	1738	354
⋮	⋮	⋮

[Wide type]

Long type의 데이터프레임 생성

- 두 범주형 변수 기준으로 집계한 Long type의 데이터프레임을 생성합니다.

```
>>> elong = price.groupby(by = cols).count()['단위금액'].reset_index()
```

- elong의 처음 10행을 출력합니다.

```
>>> elong.head(n = 10) # elong은 25개 자치구별로 '1천 미만'과 '1천 이상'의 행을 가집니다.(50행)
```

- 단위금액의 이름을 거래건수로 변경합니다.

```
>>> elong = elong.rename(columns = {'단위금액': '거래건수'})
```

- elong의 처음 10행을 다시 출력합니다.

```
>>> elong.head(n = 10)
```

Long type을 Wide type으로 변환

- Long type을 Wide type으로 변환합니다.

```
>>> widen = elong.pivot(index = '자치구', # index로 사용할 열이름을 지정합니다.
                        columns = '금액구분', # Wide type의 열이름이 될 Long type의 열이름을 지정합니다.
                        values = '거래건수') # Wide type의 값으로 채울 Long type의 열이름을 지정합니다.
>>> wide = widen.reset_index() # widen의 인덱스를 초기화합니다.
>>> widen.head(n = 10) # widen의 처음 10행을 출력합니다.
                       # widen은 25개 자치구별로 '1천 미만'과 '1천 이상'의 열을 가집니다.(25행)
>>> widen.columns.name = '' # [참고] widen의 컬럼명 이름에 빈 문자열을 할당하면, 인덱스 위에 출력된 '금액구분'을 삭제할 수 있습니다.
```

Wide type을 Long type으로 변환

- Wide type을 Long type으로 변환한 결과를 출력합니다.

```
>>> widen.melt(id_vars = '자치구', # 가장 왼쪽에 ID로 사용할 열이름을 지정합니다.
                value_vars = ['1천 미만', '1천 이상'], # Wide type의 열이름을 리스트로 지정합니다.
                var_name = '금액종류', # value_vars에 지정된 열이름을 원소로 갖는 열이름을 지정합니다.
                value_name = '매매건수') # value_vars에 지정된 열의 값을 원소로 갖는 열이름을 지정합니다.
```

피벗 테이블 생성

- 피벗 테이블은 집계함수를 이용하여 데이터를 요약합니다.

```
>>> pd.pivot_table(data = price, # 데이터프레임을 지정합니다.
                   values = '단위금액', # 피벗 테이블의 값으로 사용될 열이름을 지정합니다.
                   index = '자치구', # 행이름으로 사용될 열이름을 지정합니다.
                   columns = '금액구분', # 열이름으로 사용될 열이름을 지정합니다.
                   aggfunc = np.mean) # 집계함수를 numpy 통계 함수로 지정합니다.
                                      # [참고] 두 개 이상의 함수는 리스트로 지정합니다.
```

데이터프레임 결합concatenate

- 열이름이 같은 두 개의 데이터프레임을 세로 방향으로 결합합니다.

```
>>> pd.concat(objs = [df1, df2]) # df1과 df2를 세로 방향으로 결합한 결과를 반환합니다.
                                 # 열이름이 같은 열끼리 결합됩니다.
>>> pd.concat(objs = [df1, df2], ignore_index = True) # 데이터프레임을 결합한 결과에서
                                                      # 행이름을 초기화합니다.
```

- 열이름이 다른 두 개의 데이터프레임을 세로 방향으로 결합합니다.

```
>>> df2 = df2.rename(columns = {'아파트': '아파트명'}) # df2의 일부 열이름을 변경합니다.
>>> pd.concat(objs = [df1, df2], ignore_index = True) # 열이름이 다른 열은 결측값이 대신
                                                      # 채워집니다.
```

- 행이름이 다른 두 개의 데이터프레임을 가로 방향으로 결합합니다.

```
>>> pd.concat(objs = [df1, df2], axis = 1) # 행이름이 같은 행끼리 결합됩니다.
```

데이터프레임 병합의 시각적 예시

A

ID	V1	V2
a	a1	a2
b	b1	b2
c	c1	c2

B

ID	V3	V4
b	b3	b4
c	c3	c4
d	d3	d4

*# 데이터프레임을 병합할 때 기준 열을 **외래키**foreign key라고 합니다.*
외래키의 값이 일치하는 행을 가로 방향으로 붙입니다.

 # 데이터프레임을 병합하기 전에 두 가지를 확인해야 합니다.
- 외래키의 값이 서로 일치하는가?
- 오른쪽 데이터프레임의 외래키 값에 중복이 있는가?

내부 병합Inner Join

ID	V1	V2	V3	V4
b	b1	b2	b3	b4
c	c1	c2	c3	c4

외부 병합Full Outer Join

ID	V1	V2	V3	V4
a	a1	a2	NA	NA
b	b1	b2	b3	b4
c	c1	c2	c3	c4
d	NA	NA	d3	d4

왼쪽 외부 병합Left Outer Join

ID	V1	V2	V3	V4
a	a1	a2	NA	NA
b	b1	b2	b3	b4
c	c1	c2	c3	c4

외래키 확인 및 전처리preprocessing

- price의 주소와 detail의 지번주소에서 일치하는 원소 개수를 확인합니다.

```
>>> len(set(price['주소']) & set(detail['지번주소']))
```

- 외래키로 사용할 시리즈를 각각 출력합니다.

```
>>> price['주소'].head() # price의 주소를 출력합니다. '서울특별시'로 시작합니다.
>>> detail['지번주소'].head() # detail의 지번주소를 출력합니다. '서울'로 시작합니다.
```

- price의 주소에서 '특별시'를 삭제합니다.

```
>>> price['주소'] = price['주소'].str.replace(pat = '특별시', repl = '')
>>> len(set(price['주소']) & set(detail['지번주소']))
```

[참고] 샘플링 함수 소개

- 관련 라이브러리를 호출합니다.

```
>>> import random
```

- 재현 가능한 결과를 얻기 위해 임의의 수를 생성하기 전에 시드를 고정합니다.

```
>>> random.seed(a = 1)
```

- 1~45의 정수에서 6개를 비복원추출로 샘플링합니다.

```
>>> lotto = random.sample(population = range(1, 46), k = 6)
>>> lotto.sort() # lotto의 원소를 오름차순 정렬합니다.
>>> print(lotto)
```

[참고] 중복 원소 확인 함수 소개

- 1~5의 정수에서 10개를 복원추출하여 시리즈를 생성합니다.

```
>>> random.seed(a = 1)

>>> nums = random.choices(population = range(1, 6), k = 10)

>>> nums = pd.Series(data = nums); nums
```

- **duplicated**() 함수는 원소의 중복 여부를 True 또는 False로 반환합니다.

```
>>> nums.duplicated() # nums에서 순방향으로 중복된 원소를 True로 반환합니다.
                      # [참고] keep 매개변수에 전달되는 인자의 기본값은 'first'입니다.

>>> nums.duplicated(keep = 'last') # nums에서 역방향으로 중복된 원소를 True로 반환합니다.

>>> nums.duplicated(keep = False) # nums에서 중복된 모든 원소를 True로 반환합니다.
```

데이터프레임 중복 원소 확인 및 제거

- detail의 지번주소에서 중복된 건수를 출력합니다.

```
>>> detail['지번주소'].duplicated().sum() # 왼쪽 코드를 실행하면 25가 반환됩니다.
                                          # [참고] keep = False를 추가하면 48이 반환됩니다.
```

- detail의 지번주소가 중복인 모든 행을 선택합니다.

```
>>> dup = detail['지번주소'].duplicated(keep = False) # dup은 True 또는 False를 원소로
                                                      갖는 시리즈입니다.
>>> detail[dup].sort_values(by = ['지번주소']) # dup이 True인 행만 남기고 지번주소 기준으로
                                               오름차순 정렬한 결과를 반환합니다.
```

- detail의 지번주소가 중복일 때 첫 번째 행만 남기고 detail에 재할당합니다.

```
>>> detail = detail.drop_duplicates(subset = ['지번주소'], keep = 'first')
>>> detail.shape[0] # detail의 행 길이가 25개 감소했습니다.(9145 -> 9120)
```

데이터프레임 병합merge

- 두 개의 데이터프레임으로 내부 병합을 실행합니다.

```
>>> pd.merge(left = price, # 왼쪽 데이터프레임을 지정합니다.
              right = detail, # 오른쪽 데이터프레임을 지정합니다.
              how = 'inner', # how 매개변수에 병합 방법을 지정합니다.(기본값: 'inner')
                             # 외부 병합은 'outer', 왼쪽 외부 병합은 'left'를 지정합니다.
              left_on = '주소', # left_on 매개변수에 왼쪽 데이터프레임의 외래키 열이름을 지정합니다.
              right_on = '지번주소') # right_on 매개변수에 오른쪽 데이터프레임의 외래키 열이름을
                                     # 지정합니다.
```

데이터프레임 병합merge(계속)

- detail의 외래키 이름을 변경합니다.

```
>>> detail = detail.rename(columns = {'지번주소': '주소'})
>>> detail.head()
```

- 외래키 이름이 같으면 on 매개변수를 사용하거나, 생략할 수 있습니다.

```
>>> apt = pd.merge(left = price, right = detail, how = 'inner', on = '주소')
```

- apt의 정보를 확인합니다.

```
>>> apt.info()
```

외부 파일로 저장

- 현재 작업경로를 확인합니다.

```
>>> os.getcwd()
```

- apt를 xlsx, csv 파일로 각각 저장합니다. *# csv 파일을 저장할 때 소요되는 시간이 훨씬 적습니다.*

```
>>> apt.to_excel(excel_writer = 'APT_List_Seoul_2020.xlsx', index = None)
>>> apt.to_csv(path_or_buf = 'APT_List_Seoul_2020.csv', index = None)
```

- 현재 작업경로에 포함된 폴더명과 파일명을 출력합니다.

```
>>> os.listdir()
```

데이터 시각화

데이터 시각화 개요

- 탐색적 데이터 분석EDA 과정에서 히스토그램, 상자 수염 그림, 산점도 등 그래프로 일변량 변수의 분포 및 이변량 변수의 관계를 확인함으로써 분석할 데이터에 대한 이해의 폭을 넓힐 수 있습니다.
- 특히 선형 회귀모형을 적합하기 전 목표변수와 입력변수 간 산점도를 그려봄으로써 두 변수 간 직선의 관계를 갖는지 여부를 시각적으로 확인할 수 있습니다.
- 아울러 데이터 분석 결과를 시각적으로 표현함으로써 데이터 분석 과정에 참여하지 않은 사람들에게 분석 결과를 쉽고 빠르게 전달할 수 있습니다.
- 결국 데이터 시각화는 그래프를 이용하여 데이터에 내재되어 있는 패턴을 발굴하고 이를 효과적으로 전달하기 위해 수행하는 과정이라 할 수 있습니다.

데이터 시각화 종류

구분	특징
히스토그램	● 히스토그램은 일변량 연속형 변수의 도수분포표를 시각화한 것입니다. ● 히스토그램은 막대가 서로 붙어 있으며, 막대의 총 면적은 확률 1을 의미합니다.
상자 수염 그림	● 일변량 상자 수염 그림은 연속형 변수의 분포에 사분위수와 이상치를 시각화한 것입니다. ● 이변량 상자 수염 그림은 범주형 변수에 따라 연속형 변수의 분포를 시각화한 것입니다.
막대그래프	● 일변량 막대그래프는 범주형 변수의 빈도수를 시각화한 것입니다. ● 이변량 막대그래프는 범주형 변수에 따라 연속형 변수의 크기를 시각화한 것입니다.
선그래프	● 선그래프는 시간의 흐름에 따라 연속형 변수의 변화를 시각화한 것입니다. ● 따라서 주가 데이터와 같이 시계열 변수를 시각화할 때 사용합니다.
산점도	● 산점도는 이변량 연속형 변수의 선형관계를 시각화한 것입니다. ● 선형 회귀분석에 사용할 입력변수와 목표변수 간 직선의 관계가 있는지 확인합니다.

실습 데이터셋 준비

- 관련 라이브러리를 호출합니다.

```
>>> import os, chardet
>>> import numpy as np
>>> import pandas as pd
```

- 현재 작업경로 확인 및 data 폴더로 작업경로를 변경하고, 파일명을 출력합니다.

```
>>> os.getcwd()
>>> os.chdir(path = '../data')
>>> os.listdir()
```

실습 데이터셋 준비(계속)

- csv 파일의 문자 인코딩 방식을 확인합니다.

```
>>> fileName = 'APT_List_Seoul_2020.csv' # csv 파일명을 fileName에 할당합니다.
>>> raw = open(file = fileName, mode = 'rb').read() # csv 파일을 바이너리로 읽습니다.
>>> chardet.detect(raw[:100]) # csv 파일의 문자 인코딩 방식을 확인합니다.
```

- csv 파일을 읽고 데이터프레임을 생성합니다.

```
>>> apt = pd.read_csv(fileName, parse_dates = ['거래일'])
>>> apt.head() # apt의 처음 5행을 출력합니다.
>>> apt.info() # apt의 정보를 확인합니다.
```

관련 라이브러리 호출

- 관련 라이브러리를 호출합니다.

```
>>> import seaborn as sns  # 고급 시각화 함수를 포함하는 라이브러리입니다.

>>> import matplotlib.pyplot as plt  # 그래프 크기, 제목, 축이름 등을 지정할 때 사용합니다.

>>> import matplotlib.font_manager as fm  # 한글폰트를 지정할 때 사용합니다.

>>> %matplotlib inline  # 주피터 노트북에서 그린 그래프를 웹 브라우저에서 바로 출력하도록 설정합니다.
                        # 하지만 왼쪽 코드를 생략해도 무방합니다.
```

그래픽 옵션 설정

- 그래프의 크기 및 해상도를 설정합니다.

```
>>> plt.rc(group = 'figure', figsize = (6, 6), dpi = 100)
```

- 선 굵기 및 스타일을 설정합니다.

```
>>> plt.rc(group = 'lines', linewidth = 0.5, linestyle = '-')
```

- 한글폰트 및 글자 크기를 설정합니다.

```
>>> plt.rc(group = 'font', family = 'Gamja Flower', size = 10)
```

- 한글폰트에는 마이너스 기호가 없으므로, 축에 출력되지 않도록 설정합니다.

```
>>> plt.rc(group = 'axes', unicode_minus = False)
```

[참고] 한글폰트명 찾는 법

- 현재 사용 중인 컴퓨터에 설치된 한글폰트 목록에서 폰트명으로 탐색합니다.

```
>>> fontList = fm.findSystemFonts(fontext = 'ttf') # 컴퓨터에 설치된 폰트 파일을 탐색하고 리스트로 반환합니다.
>>> fontPath = [font for font in fontList if 'Gamja' in font] # 폰트명으로 한글폰트 파일을 선택합니다.
>>> fontPath.sort() # 한글폰트 파일을 원소로 갖는 리스트를 오름차순으로 정렬합니다.
```

- 반복문으로 한글폰트명을 출력하고, 마음에 드는 폰트명을 선택합니다.

```
>>> for font in fontPath:
        print(fm.FontProperties(fname = font).get_name())
```

[참고] 한글이 네모로 출력되는 문제 해결 방법

- 한글폰트를 설정했음에도 아래 경고와 함께 한글이 네모로 출력될 수 있습니다.
  - 에러 메시지: Font family ['폰트명'] not found. Falling back to DejaVu Sans
- 관련 라이브러리를 호출합니다.

```
>>> import matplotlib, glob  # glob 모듈의 glob() 함수는 조건에 맞는 파일명을 리스트로 반환합니다.
                             # 조건에 정규표현식을 사용할 수 없고, 와일드카드(*, ?)는 지원합니다.
```

- matplotlib 라이브러리의 임시 폴더에 있는 json 파일을 삭제합니다.

```
>>> path = matplotlib.get_cachedir()  # matplotlib 라이브러리 임시 폴더 경로를 path로 생성합니다.
                                      # 폰트 목록을 담고 있는 JSON 파일명을 file로 생성합니다.
>>> file = glob.glob(pathname = f'{path}/fontlist-*.json')[0]
>>> os.remove(path = file)  # matplotlib 라이브러리 임시 폴더에 있는 JSON 파일을 삭제합니다.
                            # 주피터 노트북을 재실행하면 한글폰트가 제대로 설정됩니다.
```

[참고] 그래픽 파라미터 설정 관련 모듈 생성

- 시각화 관련 라이브러리 호출, 그래프 크기 및 한글폰트 설정 등 일련의 과정을 py 파일로 저장하면 필요할 때마다 편리하게 호출할 수 있습니다.
- 새로운 주피터 노트북을 열고, py 파일로 저장할 코드만 복사하여 붙여넣습니다.
- 상단 메뉴에서 File > Download as > Python(.py) 메뉴를 차례로 선택합니다.
- 이번 예제에서는 GraphicSetting.py 파일로 저장합니다.
    - [참고] py 파일을 호출하려면 주피터 노트북 파일과 같은 폴더에 있어야 합니다.
- 관련 모듈을 호출합니다.

```
>>> from GraphicSetting import *
```

히스토그램 계급 생성

- 거래금액의 최솟값과 최댓값을 확인합니다.

```
>>> apt['거래금액'].describe()
```

```
>>> apt['거래금액'].describe()[['min', 'max']]
```

- 히스토그램 계급(막대의 경계)을 설정합니다.

```
>>> bins = np.arange(start = 0, stop = 80, step = 2)  # 최솟값보다 작은 숫자로 시작하고 최댓값보다 큰 숫자로 끝납니다.
```

- 히스토그램 계급을 출력합니다.

```
>>> bins  # 히스토그램 계급은 0 이상 ~ 2 미만, 2 이상 ~ 4 미만과 같이 설정됩니다.
```

히스토그램histogram 그리기

- 거래금액의 분포를 히스토그램으로 시각화합니다.

```
>>> sns.histplot(data = apt, # 데이터프레임을 지정합니다.
                  x = '거래금액', # 히스토그램을 그릴 일변량 연속형 열이름을 지정합니다.
                  bins = bins, # 히스토그램의 계급을 미리 생성했던 배열로 지정합니다.
                               [참고] 막대 개수를 정수로 지정해도 됩니다.
                  color = '0.5', # 막대 채우기 색을 문자열로 지정합니다.
                                 [참고] '0'은 검정이고, '1'은 흰색을 의미합니다.
                  edgecolor = 'black') # 막대 테두리 색을 문자열로 지정합니다.
>>> plt.title(label = '거래금액의 분포'); # 그래프 제목을 지정합니다.
```

[참고] KDE 밀도 곡선 추가

- 히스토그램의 y축을 빈도수count 대신 밀도density로 변경합니다.

```
>>> sns.histplot(data = apt, x = '거래금액', bins = bins, color = '1',
                 stat = 'density') # stat 매개변수의 기본인자는 'count'입니다.
```

- 히스토그램에 KDE 밀도 곡선을 추가합니다.

```
>>> sns.kdeplot(data = apt, x = '거래금액', color = 'red', linewidth = 1.5);
```

[참고] matplotlib CSS Colors

- matplotlib에서 제공되는 CSS Color 목록에서 원하는 색을 고를 수 있습니다.

```
>>> import matplotlib.colors as mcolors
>>> mcolors.CSS4_COLORS  # 148가지 색이름과 Hex Code를 딕셔너리로 출력합니다.
```

히스토그램을 겹쳐서 그리기

- 관심 있는 자치구 3개를 선택하여 top3를 생성합니다.

```
>>> top3 = apt[apt['자치구'].str.contains(pat = '강남|서초|송파')].copy()
>>> top3['거래금액'].describe()[['min', 'max']] # 거래금액의 최솟값과 최댓값을 확인합니다.
>>> bins = np.arange(start = 0, stop = 68, step = 1) # 히스토그램 계급을 설정합니다.
```

- 특정 변수의 값에 따라 히스토그램의 색을 다르게 겹쳐서 그립니다.

```
>>> sns.histplot(data = top3, x = '거래금액', bins = bins,
                 hue = '자치구', palette = 'Dark2');
# hue 매개변수에 지정된 변수에 따라 히스토그램을 각각 그립니다.
# palette 매개변수에 히스토그램의 막대 채우기 색 팔레트를 설정합니다.
```

히스토그램을 나눠서 그리기

- 특정 변수의 값에 따라 히스토그램을 가로 방향으로 나눠서 그립니다.

```
>>> sns.displot(data = top3, x = '거래금액', bins = bins, hue = '자치구',
                palette = 'Dark2', col = '자치구', legend = False);
```

- 특정 변수의 값에 따라 히스토그램을 세로 방향으로 나눠서 그립니다.

```
>>> sns.displot(data = top3, x = '거래금액', bins = bins, hue = '자치구',
                palette = 'Dark2', row = '자치구', legend = False);
```

[참고] 컬러맵(팔레트) 목록 및 색상 확인

- pyplot 모듈에서 제공되는 컬러맵 목록을 출력합니다.

```
>>> dir(plt.cm)
```

- 기본 팔레트의 색을 출력합니다.

```
>>> sns.color_palette(palette = 'tab10')
```

- 관심 있는 팔레트의 색을 출력합니다.

```
>>> sns.color_palette(palette = 'Dark2')
```

- 출력할 색의 개수를 설정합니다.

```
>>> sns.color_palette(palette = 'Dark2', n_colors = 10)
```

[참고] 사용자 팔레트 생성

- 색이름을 리스트로 생성합니다.

```
>>> colors = ['skyblue', 'orange', 'purple']
```

- 색이름 리스트로 사용자 팔레트를 설정합니다.

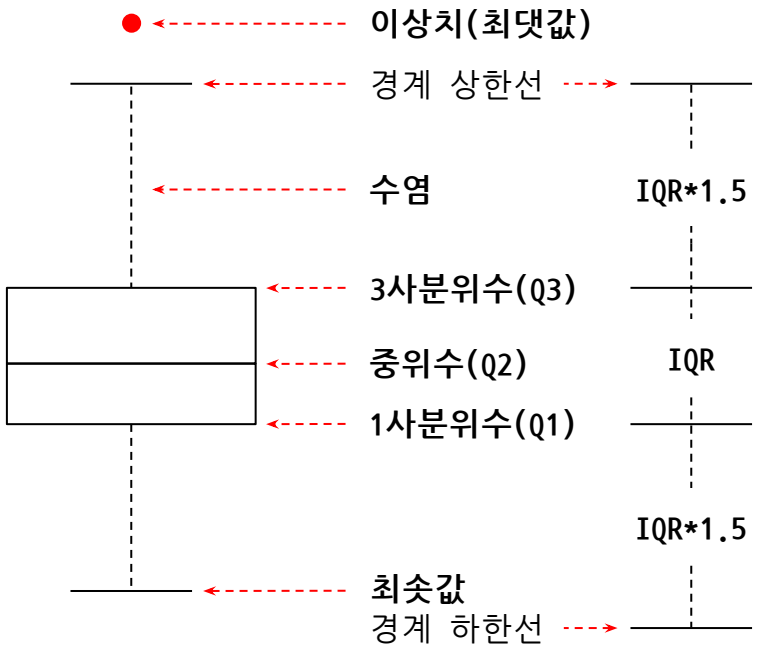
```
>>> myPal = sns.set_palette(palette = colors)
```

- 기존 그래프에 사용자 팔레트를 적용합니다.

```
>>> sns.displot(data = top3, x = '거래금액', bins = bins, hue = '자치구',
                palette = myPal, col = '자치구', legend = False);
```

상자 수염 그림boxplot

- 상자 수염 그림은 연속형 변수의 분포에 사분위수와 이상치를 시각화한 것입니다.
  - 변수의 분포를 세로로 표현한 것입니다.
- 상자 수염 그림을 통해 기술통계량은 물론 이상치도 확인할 수 있습니다.
  - Q1 및 Q3 간 간격은 사분범위입니다.
    - IQR: Interquartile range
  - Q1과 Q3에서 IQR의 1.5배를 미달/초과하는 원소를 이상치로 간주할 수 있습니다.



상자 수염 그림 그리기

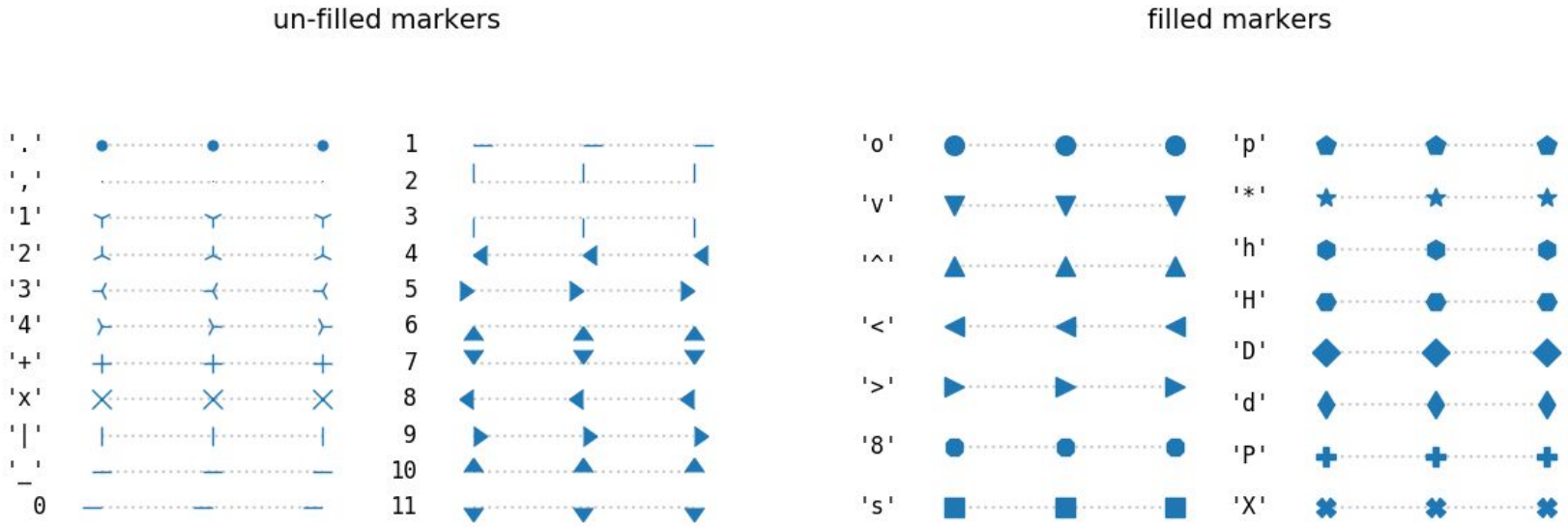
- 이상치의 모양, 크기 및 채우기/테두리 색을 설정합니다.

```
>>> flierprops = {'marker': 'd', # 이상치의 모양을 설정합니다.(기본값: 'd')
                  'markersize': 5, # 이상치의 크기를 설정합니다.(기본값: 5)
                  'markerfacecolor': 'pink', # 이상치의 채우기 색을 설정합니다.
                                             # (기본값: 'black')
                  'markeredgecolor': 'red'} # 이상치의 채우기 색을 설정합니다.
                                            # (기본값: 'black')
```

- 거래금액으로 상자 수염 그림을 그립니다.

```
>>> sns.boxplot(data = apt, y = '거래금액', flierprops = flierprops)
                                # 상자 수염 그림을 세로로 그립니다.
                                # [참고] y 대신 x를 사용하면 상자 수염 그림이 가로로 그려집니다.
```

[참고] marker의 종류



 # 출처: https://matplotlib.org/3.1.0/gallery/lines_bars_and_markers/marker_reference.html

집단별 상자 수염 그림 그리기

- 자치구별 거래금액의 중위수를 오름차순으로 정렬한 sigg를 생성합니다.

```
>>> sigg = apt.groupby(by = ['자치구']).median()['거래금액']; sigg.head()
>>> sigg = sigg.sort_values()
```

- 자치구를 x축, 거래금액을 y축에 놓고 집단별 상자 수염 그림을 그립니다.

```
>>> sns.boxplot(data = apt, x = '자치구', y = '거래금액',
                order = sigg.index) # x축의 순서를 sigg의 인덱스로 지정합니다.
>>> plt.xticks(rotation = 45); # x축 눈금명을 45도로 회전시킵니다.
```

일변량 막대그래프countplot 그리기

- 자치구별 거래금액의 빈도수를 오름차순으로 정렬한 sigg를 생성합니다.

```
>>> sigg = apt.groupby(by = ['자치구']).count()['거래금액']; sigg.head()
>>> sigg = sigg.sort_values(ascending = False)
```

- 자치구별 빈도수로 막대그래프를 그립니다.

```
>>> sns.countplot(data = apt, x = '자치구', order = sigg.index)
>>> plt.xticks(rotation = 45);
```

이변량 막대그래프barplot 그리기

- 자치구별 거래금액의 평균을 내림차순으로 정렬한 sigg를 생성합니다.

```
>>> sigg = apt.groupby(by = ['자치구']).mean()['거래금액']; sigg.head()
>>> sigg = sigg.round(1).sort_values(ascending = False)
```

- 자치구별 거래금액의 평균으로 막대그래프를 그립니다.

```
>>> sns.barplot(data = apt, x = '자치구', y = '거래금액', order = sigg.index,
                estimator = np.mean, ci = None) # 거래금액의 평균을 계산합니다.
                                                # 95% 신뢰구간이 출력되지 않도록 합니다.
>>> plt.ylim(0, 20) # 나중에 텍스트를 추가할 공간을 확보하기 위해 y축 범위를 0~20으로 제한합니다.
>>> plt.xticks(rotation = 45);
```

막대 위에 텍스트 추가

- 막대그래프 위에 거래금액 평균을 텍스트로 추가합니다.

```
>>> sns.barplot(data = apt, x = '자치구', y = '거래금액', order = sigg.index,
                estimator = np.mean, ci = None)
>>> for index, value in enumerate(sigg):
        plt.text(x = index, y = value + 0.2, s = value, fontsize = 10,
                 ha = 'center', va = 'bottom', color = 'black')
>>> plt.ylim(0, 20)
>>> plt.xticks(rotation = 45);
```

*# **enumerate**() 함수는 객체의 정수 인덱스와 값을 각각 반환합니다.*

 # s 매개변수에 텍스트로 출력할 값을 지정합니다.
 # ha 매개변수에는 수평정렬horizontal align 방식을 지정합니다.
 # va 매개변수에는 수직정렬vertical align 방식을 지정합니다.
 # color 매개변수에는 글자 색상을 지정합니다.

선그래프linegraph 그리기

- apt에 거래월을 추가합니다.

```
>>> apt['거래월'] = apt['거래일'].dt.month
```

```
>>> apt.head() # apt의 처음 5행을 출력하면 마지막에 거래월이 추가된 것을 확인할 수 있습니다.
```

- 거래월별 거래금액의 평균으로 선그래프를 그립니다.

```
>>> sns.lineplot(data = apt, x = '거래월', y = '거래금액', ci = None);
```

 # 월별 거래금액 평균의 95% 신뢰구간이 추가됩니다.
ci = None를 추가하면 신뢰구간을 제거합니다.

- 점을 추가한 선그래프를 그립니다.

```
>>> sns.pointplot(data = apt, x = '거래월', y = '거래금액', ci = None);
```

[참고] x축 눈금명 변경

- x축 눈금명을 생성합니다.

```
>>> months = [f'{i}월' for i in range(1, 13)]; months
```

- 선그래프에 x축 눈금 위치와 눈금명을 지정합니다.

```
>>> sns.lineplot(data = apt, x = '거래월', y = '거래금액', ci = None)
>>> plt.xticks(ticks = range(1, 13), labels = months);
```

- 점을 추가한 선그래프는 눈금 시작 위치가 다릅니다.

```
>>> sns.pointplot(data = apt, x = '거래월', y = '거래금액', ci = None)
>>> plt.xticks(ticks = range(0, 12), labels = months);
```

선그래프를 겹쳐서 그리기

- top3에 거래월을 추가합니다.

```
>>> top3['거래월'] = top3['거래일'].dt.month
```

- 자치구별 선그래프를 겹쳐서 그립니다.

```
>>> sns.lineplot(data = top3, x = '거래월', y = '거래금액', ci = None,
                 hue = '자치구', palette = 'Dark2')
>>> plt.xticks(ticks = range(1, 13), labels = months);
```

선그래프를 나눠서 그리기

- 자치구별 선그래프를 나눠서 그립니다.

```
>>> sns.relplot(data = top3, x = '거래월', y = '거래금액', ci = None,
                hue = '자치구', palette = 'Dark2', legend = False,
                col = '자치구', kind = 'line')
# col 매개변수에 지정된 변수의 원소별 선그래프를 가로 방향으로 펼쳐서 그립니다.
  row 매개변수에 지정된 변수의 원소별 선그래프를 세로 방향으로 펼쳐서 그립니다.

# kind 매개변수에 전달되는 인자의 기본값은 'scatter'이므로 생략하면 산점도를 그립니다.

>>> plt.xticks(ticks = range(1, 13), labels = months);
```

산점도scatterplot 그리기

- 전용면적과 거래금액의 관계를 산점도로 그립니다.

```
>>> sns.scatterplot(data = apt, x = '전용면적', y = '거래금액',
# 점의 채우기 색, 테두리 색
  및 선의 두께를 설정합니다.  color = '0.8', edgecolor = 'black', linewidth = 1);
```

- 산점도에 회귀직선을 추가한 그래프를 그립니다.

```
>>> sns.regplot(data = apt, x = '전용면적', y = '거래금액',
                scatter_kws = {'color': '0.8', 'edgecolor': 'black'},
# 점과 회귀직선의
  요소를 설정합니다.
                line_kws = {'color': 'red', 'linewidth': 1.5});
```

산점도를 겹쳐서 또는 나눠서 그리기

- 자치구별 산점도를 겹쳐서 그립니다.

```
>>> sns.scatterplot(data = top3, x = '전용면적', y = '거래금액',
                    edgecolor = 'white', linewidth = 0.5,
                    hue = '자치구', palette = 'Dark2');
```

- 자치구별 산점도를 나눠서 그립니다.

```
>>> sns.relplot(data = top3, x = '전용면적', y = '거래금액',
                hue = '자치구', palette = 'Dark2', legend = False,
                col = '자치구', kind = 'scatter');
```

산점도 행렬scatterplot matrix 그리기

- 여러 개의 산점도를 동시에 시각화하여 변수 간 관계를 빠르게 확인하고자 할 때 산점도 행렬을 그립니다.
  - 주대각에는 변수의 히스토그램, 삼각행렬에는 산점도를 그립니다.
  - 열 개수가 많으면 시간이 오래 걸리고 가독성이 떨어집니다.
- 산점도 행렬에 포함할 숫자 변수명을 리스트로 생성합니다.(4~5개가 적당)

```
>>> cols = ['자치구', '거래금액', '전용면적', '층', '세대수']
```

- 산점도 행렬을 그립니다.

```
>>> sns.pairplot(data = top3[cols], hue = '자치구', palette = 'Dark2');
```

산점도 행렬에서 하나의 행만 출력

- x축 변수와 y축 변수를 설정하면 산점도 행렬이 간결해집니다.

```
>>> sns.pairplot(data = top3,
                 x_vars = ['전용면적', '층', '세대수'], # x축에 놓을 열이름을 리스트로 지정합니다.
                 y_vars = ['거래금액'], # y축에 놓을 열이름을 리스트로 지정합니다.
                 hue = '자치구',
                 palette = 'Dark2');
```

[부록] Python 라이브러리 공식 문서

- 이번 강의에서 소개해드린 주요 라이브러리 공식 문서를 소개합니다.
  - numpy: https://numpy.org/doc/stable/numpy-user.pdf
  - pandas: https://pandas.pydata.org/docs/user_guide/index.html#user-guide
  - dfply: https://github.com/kieferk/dfply
  - seaborn: https://seaborn.pydata.org/tutorial.html
  - matplotlib: https://matplotlib.org/tutorials/index.html

End of Document