

Python Basic

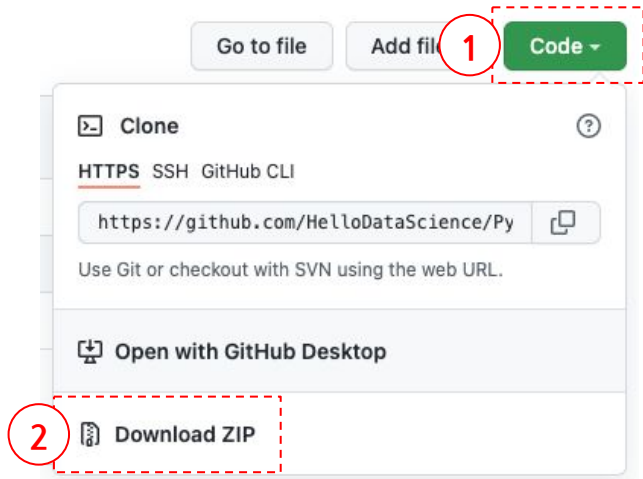
Data Analysis

강의 내용

- 프로그래밍 시작
- 자료형
- 자료구조
- 제어문
- 사용자 정의 함수
- numpy 라이브러리
- pandas 라이브러리
- 데이터 입출력
- 데이터 전처리
- 데이터 시각화

실습 데이터셋 내려받기

- 크롬에서 깃허브 저장소로 접속합니다. [주의] 크롬 아니면 에러 발생 가능!
 - URL: `https://github.com/HelloDataScience/PythonBasic`
- 초록색 Code와 Download ZIP을 차례대로 클릭하면 zip 파일을 다운로드 폴더에 저장합니다.
- zip 파일을 적당한 위치(예: 문서 폴더)에서 압축을 풀고 code와 data 폴더를 확인합니다.
 - code: Jupyter Notebook 파일을 포함하는 폴더입니다.
 - data: 실습 데이터 파일을 포함하는 폴더입니다.



프로그래밍 시작

Python이란?

- Python은 네덜란드 출신의 컴퓨터 프로그래머이자 구글, 드롭박스를 거쳐 현재는 MS 개발자인 **귀도 반 로섬** Guido Van Rossum이 1991년에 발표한 프로그래밍 언어입니다.
 - 1989년 12월, 귀도는 취미가 될만한 프로젝트를 찾고 있었고 마침 크리스마스 휴일이라 사무실 문을 닫으면서 집에 있는 컴퓨터로 Python 개발에 착수하였다고 전해집니다.
- Python이라는 이름은 당시 귀도가 좋아하던 코미디 쇼였던 "**Monty Python's Flying Circus**"에서 따온 것입니다.
 - Python(피톤)은 그리스 신화에서 나오는 큰 뱀입니다.
 - Python 로고는 뱀 모양입니다.
 - Python의 버전은 2와 3이 있습니다.



Python의 특징

- Python은 인간다운 언어이므로 사람이 생각하는 방식으로 프로그래밍합니다.
 - 영어를 모국어로 하는 사람들에게 해당합니다. 최소한 한글은 아니니까요.
- Python은 문법이 쉬워 빠르게 배울 수 있습니다.
 - 프로그래밍에 익숙하다면 일주일이면 충분하겠지만, 초심자에게는 절대로 아닙니다.
- Python은 오픈소스이므로 무료고, 수많은 라이브러리를 사용할 수 있습니다.
 - 라이브러리 설치 에러가 종종 발생하고, 일관성이 부족한 것도 오픈소스의 특징입니다.
- Python은 간결하고, 들여쓰기 규칙으로 코드 가독성을 높일 수 있습니다.

출처: 점프 투 파이썬(<https://wikidocs.net/6>)에서 일부 발췌

Python의 특징

Life is too short, you need Python!

기본 구문: 코드 입력 및 실행

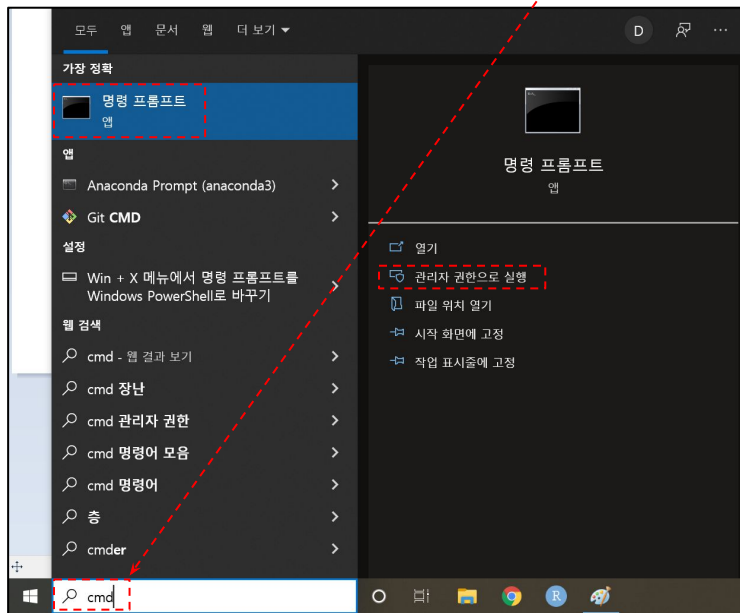
- Python을 실행하면 콘솔이 열리고, 콘솔 왼쪽에 `>>>` 기호(프롬프트)가 있습니다. 프롬프트 오른쪽에서 커서^{cursor}가 깜빡거립니다.
- Python은 스크립트 언어이므로 콘솔에서 코드를 입력하고 [enter] 키를 눌러 실행하면 코드 실행 결과를 코드 아래에 출력합니다.[대화식 모드]
- 여러 줄 코드를 입력하고 실행하려면 IDLE 편집기를 이용합니다.[스크립트 모드]
 - IDLE 편집기에서 [enter] 키를 누르면 코드를 실행하지 않고 커서를 아래로 옮깁니다.
 - 실행할 코드를 py 파일로 저장하고 상단 메뉴에서 Run → Run Module 버튼을 차례대로 클릭하면 py 파일을 실행합니다.

이번 강의에서는 JupyterLab을 이용한 라이브 코딩 실습 방식으로 진행합니다.

Windows에서 CMD 실행

- Windows 작업 표시줄에 있는 돋보기 메뉴에서 **cmd**를 입력하고 실행합니다.

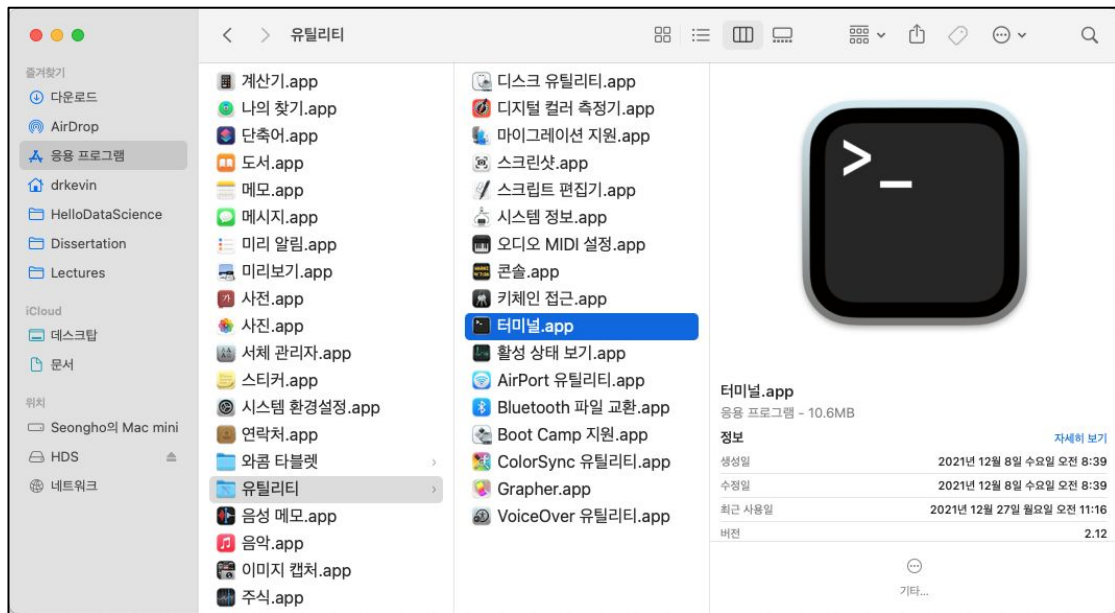
명령 프롬프트에서 마우스
오른쪽 버튼을 클릭합니다.



관리자 권한으로 실행을
선택합니다.

MacOS에서 Terminal 실행

- 파인더 > 응용 프로그램 > 유틸리티 폴더에서 터미널.app을 실행합니다.



Jupyter Notebook 실행

- CMD(Windows) 또는 Terminal(MacOS)에서 아래 코드를 실행하면 기본 브라우저에서 새 탭이 열리고, Jupyter Notebook을 실행합니다.
[참고] Jupyter Notebook을 크롬에서 실행하는 것을 추천합니다.

```

명령 프롬프트
Microsoft Windows [Version 10.0.19043.1415]
(c) Microsoft Corporation. All rights reserved.
C:\Users\User>jupyter notebook
  
```

```

hdsceokevin — zsh — 80x24
(Venv) hdsceokevin@Seonghoui-Macmini ~ % jupyter lab

# [참고] 위 코드는 JupyterLab을 실행합니다.
  
```

[참고] Jupyter Notebook 실행 경로를 아래와 같이 추가할 수 있습니다.
% jupyter notebook --notebook-dir=경로명

Jupyter Notebook 메인화면

The screenshot shows the Jupyter Notebook main interface. At the top, there's a 'jupyter' logo and 'Quit' and 'Logout' buttons. Below that are tabs for 'Files', 'Running', and 'Clusters'. A message says 'Select items to perform actions on them.' Below this is a file browser showing a list of folders: Applications, Desktop, Documents, Downloads, Movies, Music, Pictures, and Public. A red dashed arrow points from the text '# Jupyter Notebook의 시작 위치는 User 폴더입니다.' to the '/' icon in the file browser. Another red dashed arrow points from the text '# 새 Python Jupyter Notebook을 열려면 New → Python 3를 차례대로 선택합니다.' to the 'New' button and the 'Python 3' option in the dropdown menu. The dropdown menu also shows 'R', 'Text File', 'Folder', and 'Terminal' under the 'Other:' section.

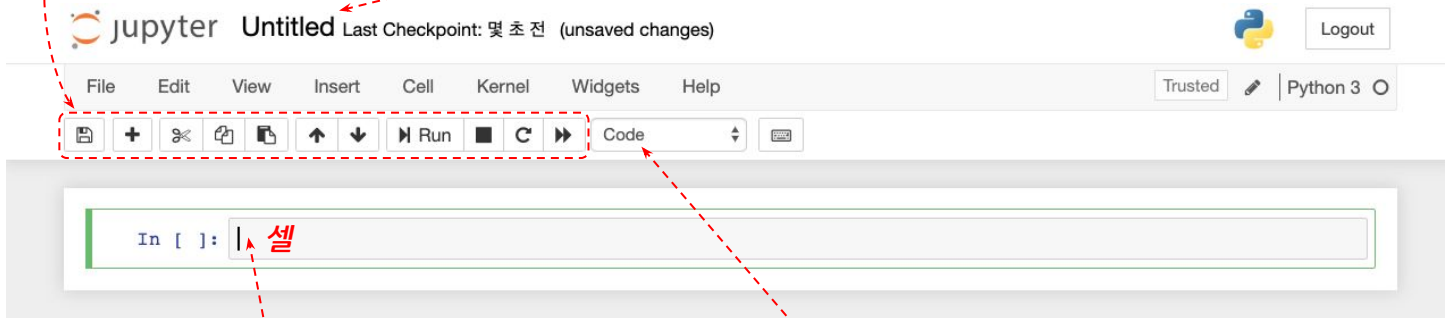
Jupyter Notebook의 시작 위치는 User 폴더입니다.

새 Python Jupyter Notebook을 열려면 New → Python 3를 차례대로 선택합니다.

Jupyter Notebook 메뉴

저장, 셀^{Cell} 추가, 잘라내기, 복사, 붙여넣기 및 코드 실행 메뉴입니다.

Untitled를 클릭하면 **Rename Notebook** 팝업이 뜨는데 Jupyter Notebook의 파일명을 변경할 수 있습니다.



코드와 주석을 셀에 입력합니다.
[참고] 주석은 # 기호로 시작합니다.

코드를 실행하려면 ▶ 버튼을 클릭하거나 [shift] + [enter] 키를 누릅니다.

셀은 편집모드와 명령모드가 있습니다.
편집모드일 때 코드를 입력할 수 있습니다.

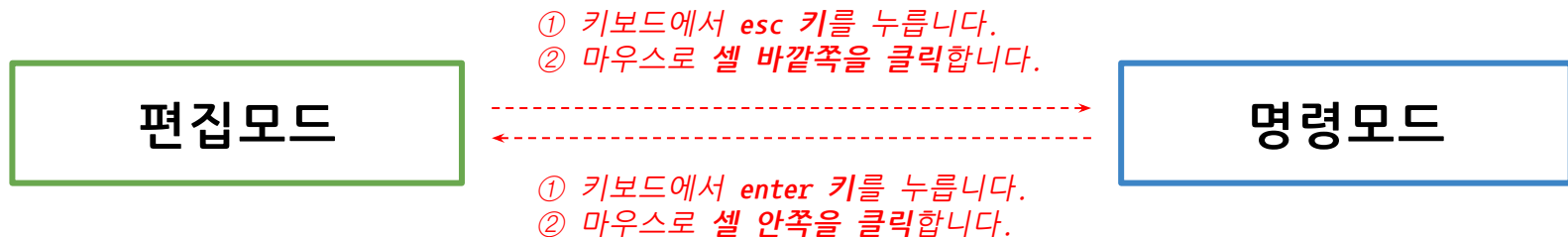
셀을 세 가지 형태^{type}로 변경할 수 있습니다.

- **Code**는 주석과 코드를 입력할 수 있습니다.
- **Markdown**은 마크다운 문서로 작성할 수 있습니다.
- **Raw**는 코드를 원형으로 보관할 때 사용합니다.

[참고] 마크다운은 텍스트 기반의 마크업 언어이며 HTML 또는 PDF로 쉽게 변환할 수 있습니다.

Jupyter Notebook 셀 모드

- 편집모드는 셀 테두리 색이 초록색으로 활성화된 상태입니다.
 - 편집모드일 때, 셀에 코드와 주석을 입력하고 실행합니다.
- 명령모드는 셀 테두리 색이 파란색으로 비활성화된 상태입니다.
 - 명령모드일 때, 단축키로 셀 관련 작업을 실행합니다.



JupyterLab 메인화면

File Edit View Run Kernel Tabs Settings Help

Filter files by name

Name	Last Modified
/	
Applications	19 days ago
Desktop	35 minutes ago
Documents	11 minutes ago
Downloads	6 hours ago
Movies	4 days ago
Music	19 days ago
Pictures	17 days ago
Public	19 days ago

JupyterLab 시작 위치는 User 폴더입니다.

Launcher

Notebook

Python 3

새 Jupyter Notebook 파일을 열려면 왼쪽 아이콘을 클릭합니다.

Console

Python 3

Other

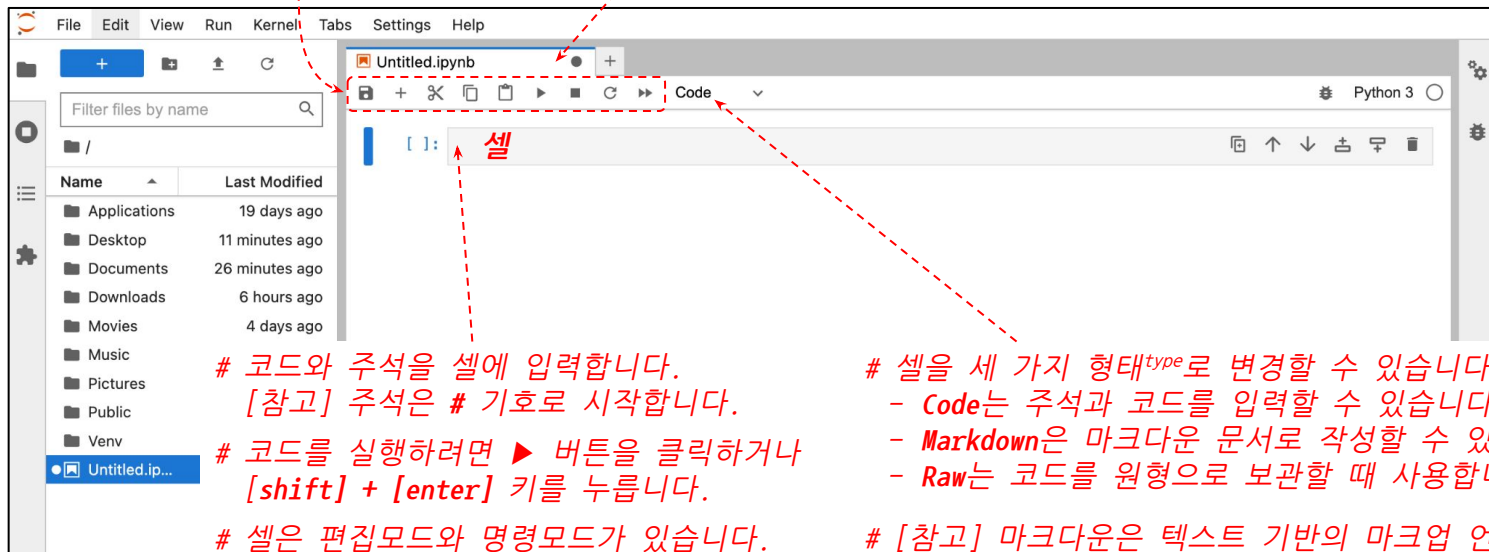
Terminal Text File Markdown File Python File Show Contextual Help

Simple 0 0 Launcher

JupyterLab에서 Notebook 메뉴

저장, 셀^{Cell} 추가, 잘라내기, 복사, 붙여넣기 및 코드 실행 메뉴입니다.

탭에서 마우스 오른쪽 버튼을 클릭하면 메뉴 팝업이 뜨는데 Rename Notebook을 선택하면 이름을 변경할 수 있습니다.



코드와 주석을 셀에 입력합니다.
[참고] 주석은 # 기호로 시작합니다.

코드를 실행하려면 ▶ 버튼을 클릭하거나 [shift] + [enter] 키를 누릅니다.

셀은 편집모드와 명령모드가 있습니다.
편집모드일 때 코드를 입력할 수 있습니다.

셀을 세 가지 형태^{type}로 변경할 수 있습니다.
- Code는 주석과 코드를 입력할 수 있습니다.
- Markdown은 마크다운 문서로 작성할 수 있습니다.
- Raw는 코드를 원형으로 보관할 때 사용합니다.

[참고] 마크다운은 텍스트 기반의 마크업 언어이며 HTML 또는 PDF로 쉽게 변환할 수 있습니다.

JupyterLab에서 Notebook 셀 모드

- 편집모드는 셀이 활성화된 상태이며, 셀에 코드와 주석을 입력하고 실행합니다.
- 명령모드는 셀이 비활성화된 상태이며, 단축키로 셀 관련 작업을 실행합니다.

편집모드



- ① 키보드에서 **esc** 키를 누릅니다.
- ② 마우스로 셀 바깥쪽을 클릭합니다.

- ① 키보드에서 **enter** 키를 누릅니다.
- ② 마우스로 셀 안쪽을 클릭합니다.

명령모드



여러 셀 복사/붙여넣기, 셀 위/아래로 이동, 셀 위/아래 추가 및 셀 삭제 메뉴입니다.

Jupyter Notebook 주요 단축키(명령모드에서 실행)

단축키	동작	단축키	동작
[y]	셀을 Code로 변환	[a], [b]	현재 셀 위, 아래로 셀 추가
[m]	셀을 Markdown으로 변환	[x]	선택한 셀 잘라내기
[r]	셀을 Raw로 변환	[c]	선택한 셀 복사
[1] ~ [6]	셀을 Headings 1~6으로 변환	[v]	선택한 셀 붙여넣기
[shift] + [enter]	코드 실행 후 아래 셀로 이동	[d] + [d]	선택한 셀 삭제
[ctrl] + [enter]	코드 실행 후 현재 셀에 멈춤	[z]	삭제한 셀 되돌리기(undo)
[alt] + [enter]	코드 실행 후 아래에 셀 추가	[f]	패턴 찾기 및 바꾸기
[↑], [↓]	현재 셀을 위, 아래로 이동	[l]	셀마다 행 숫자 보이기/감추기
[shift] + [↑], [↓]	현재 셀부터 위, 아래로 확장	[h]	단축키 목록 보이기

Jupyter Notebook 매직 명령어

- 매직 명령어^{magic commands}는 Jupyter Notebook에서 사용할 수 있는 명령어입니다.

명령어	동작	명령어	동작
%lsmagic	사용 가능한 매직 명령어 출력	%precision	실수를 소수점 최대한 길게 출력
%pwd	현재 작업 경로 출력	%precision 3	실수를 소수점 셋째 자리까지 출력
%cd '경로'	작업 경로 변경	%precision %i	실수에서 정수 부분만 출력
%ls	현재 작업 경로의 파일명 출력	%precision %e	실수를 과학적 표기법으로 출력
%who_ls	전역 변수 목록을 리스트로 출력	%time	코드 실행 소요시간 출력
%whos	전역 변수 목록을 표로 출력	%timeit	코드 실행 평균 소요시간 출력
%reset	전역 변수 일괄 삭제	%matplotlib	matplotlib 라이브러리 관련 설정

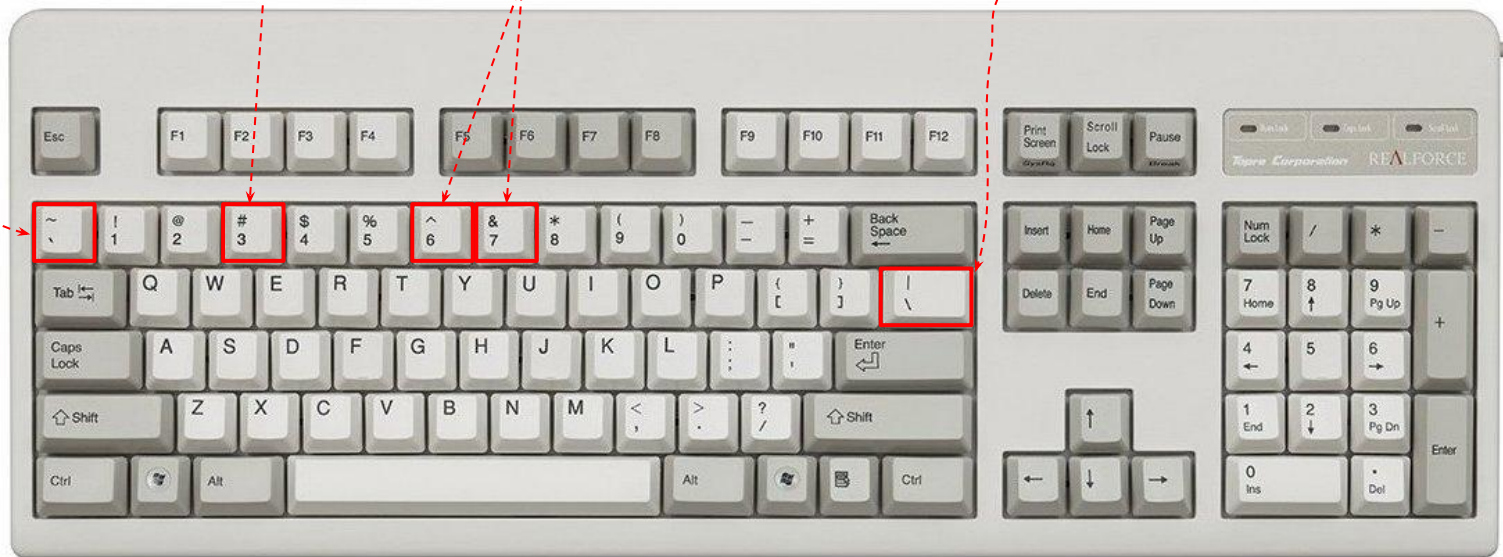
[참고] 키보드의 주요 키^{key} 위치

[~] 틸데tilde
[`] 백틱backtick

[#] 샵sharp

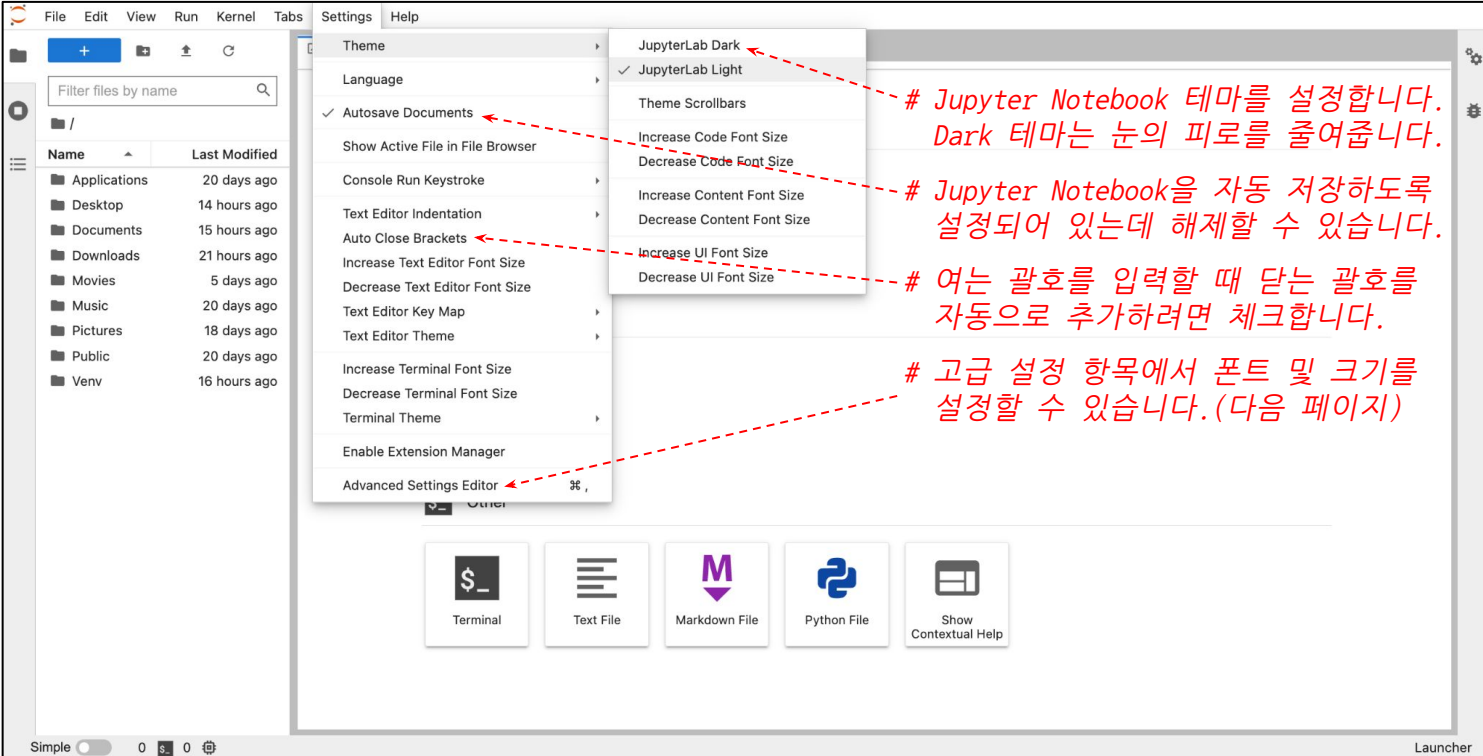
[^] 캐럿caret
[&] 앰퍼샌드ampersand

[|] 바bar
[\\] 백슬래시backslash



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

JupyterLab 환경설정



The screenshot shows the JupyterLab interface with the Settings menu open. Red dashed arrows point from Korean text annotations to specific settings:

- JupyterLab Dark**: # Jupyter Notebook 테마를 설정합니다. Dark 테마는 눈의 피로를 줄여줍니다.
- Autosave Documents**: # Jupyter Notebook을 자동 저장하도록 설정되어 있는데 해제할 수 있습니다.
- Auto Close Brackets**: # 여는 괄호를 입력할 때 닫는 괄호를 자동으로 추가하려면 체크합니다.
- Advanced Settings Editor**: # 고급 설정 항목에서 폰트 및 크기를 설정할 수 있습니다. (다음 페이지)

The interface also shows a file browser on the left, a terminal at the bottom, and various icons for opening different file types (Terminal, Text File, Markdown File, Python File, Show Contextual Help).

JupyterLab 환경설정 (계속)

Settings

Search...

MODIFIED

- Extension Manager
- Keyboard Shortcuts
- Notebook**
- Theme

SETTINGS

- Cell Toolbar
- Code Console**
- CodeMirror
- Command Palette
- CSV Viewer
- Debugger
- Document Manager
- Document Search
- File Browser
- File Browser Widget
- HTML Viewer
- JupyterLab Shell

JSON Settings Editor

Notebook

Notebook settings.

Restore to Defaults

Code Cell Configuration

The configuration for all code cells.

☐ Auto Closing Brackets

Cursor Blinking Rate

530

Half-period in milliseconds used for cursor blinking. The default blink rate is 530ms. By setting this to zero, blinking can be disabled. A negative value hides the cursor entirely.

Font Family

NanumGothicCoding

코드 기본 폰트를 설정합니다.

Default:

Font Size

14

폰트 기본 크기를 설정합니다.

Default:

Line Height

☐ Show Line Numbers

Line Wrap

off

☒ Match Brackets

☐ Read Only

☒ Insert Spaces

변수와 객체

- Python 프로그래밍을 할 때 변수^{variable}와 객체^{object}의 개념을 알아야 합니다.
 - 변수에 어떤 값을 할당^{assign}하면, 값으로 객체를 생성하고 변수는 객체를 가리킵니다.
 - 변수가 가리키는 객체는 바뀔 수 있습니다.
 - 값을 변수에 할당하지 않으면, 값이 바뀔 때마다 매번 직접 코딩해야 합니다.
 - 하지만 변수에 할당하면 값이 바뀌더라도 기존 코드를 재사용할 수 있습니다.

```
>>> 1 + 2
>>> 2 + 2
>>> 3 + 2
```

값을 직접 코딩



```
>>> a = 1
>>> a = 2
>>> a = 3
```

값을 변수에 할당



```
>>> a + 2
>>> a + 2
>>> a + 2
```

기존 코드 재사용

변수명

- 변수명은 알파벳과 숫자, 밑줄^{underscore}을 조합하여 만듭니다.
 - 변수명은 알파벳 또는 밑줄로 시작해야 하며, 숫자로 시작하면 에러가 발생합니다.
 - 변수명은 알파벳 대소문자를 구분합니다.
 - 변수명에 한글을 사용하는 것은 추천하지 않습니다.
- 변수명으로 예약어^{reserved keywords}를 사용할 수 없습니다.


```
>>> help('keywords') # Python 예약어를 확인합니다.
```
- 내장 함수명도 피하는 것이 좋습니다. # 아래 링크에서 71개 내장 함수 목록을 확인할 수 있습니다.
 - 링크: <https://docs.python.org/3/library/functions.html>

객체

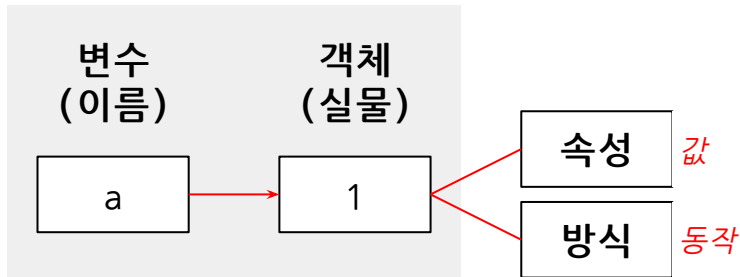
- Python에서 변수에 어떤 값을 할당하면 값의 자료형에 따라 객체를 생성합니다. 객체는 메모리에 할당되며 변수는 객체가 할당된 메모리 주소를 저장합니다.
 - 변수는 객체가 할당된 메모리 주소로 객체를 연결하므로 객체를 바꾸면 주소도 바뀝니다.
- Jupyter Notebook에서 아래 코드를 실행하여 직접 확인해보겠습니다.

```
>>> a = 1 # a에 정수 1을 할당합니다. 정수 1은 객체로 메모리에 할당됩니다.
        [참고] 등호(=)는 할당 연산자입니다.
```

```
>>> id(a) # a가 가리키는 메모리 주소를 출력합니다.
```

```
>>> a = 2 # a에 정수 2를 할당합니다.
```

```
>>> id(a) # a가 가리키는 메모리 주소가 바뀌었습니다.
```



객체의 속성과 방식

- 객체는 클래스^{class}로 생성합니다. # [참고] 많은 책에서 클래스를 붕어빵틀에 비유하여 설명합니다. 객체는 같은 붕어빵틀에서 만든 붕어빵이 됩니다.
 - 클래스는 데이터를 다룰 때 필요한 여러 속성^{attribute}과 방식^{method}을 정의한 것입니다.
 - 클래스로 생성한 객체를 인스턴스^{instance} 객체라고 합니다.
 - Python 사용자가 필요한 클래스를 생성할 수 있지만, 이 강의에서는 다루지 않습니다.
 - 사전에 정의된 클래스로 객체를 생성하고 객체의 속성과 방식을 활용하는 방법을 설명합니다.
- **type()** 함수는 변수가 가리키는 객체를 생성한 클래스를 반환합니다.
 - 변수에 할당하는 값에 따라 객체의 클래스가 정해집니다.
 - 같은 클래스로 생성한 객체는 같은 속성과 방식을 갖습니다.

객체의 속성과 방식 확인

- 변수에 어떤 값을 할당하여 객체를 생성하고, 변수의 클래스를 확인합니다.

```
>>> a = 10 # a에 정수 10을 할당합니다.
```

```
>>> a # 변수명만 입력하고 실행합니다. 변수가 가리키는 객체의 값을 출력합니다.
```

```
>>> print(a) # print() 함수로 a를 출력합니다.  
[참고] print() 함수를 실행한 결과는 변수명을 실행한 결과와 다를 수 있습니다.
```

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

- dir() 함수는 변수가 갖는 속성과 방식을 리스트로 반환합니다.

```
>>> dir(a) # a에 많은 속성과 방식이 있지만 알파벳으로 시작하는 것만 사용합니다.  
[참고] 밑줄로 시작하는 속성과 방식은 특별한 기능을 갖습니다.
```

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

객체의 속성과 방식에 접근

- 객체의 속성^{attribute} 또는 방식^{method}에 접근하려면 변수명 뒤에 마침표(온점)를 찍고 접근하려는 속성 또는 방식을 덧붙입니다.

`Object.attribute`

`Object.method()`

- 객체의 속성 또는 방식에 접근하는 예시 코드입니다.

```
>>> a.numerator # numerator은 속성입니다. 따라서 뒤에 빈 괄호를 추가하면 아래 메시지를 출력합니다.  
                TypeError: 'int' object is not callable
```

```
>>> a.bit_length() # bit.length는 방식입니다. 따라서 뒤에 빈 괄호를 생략하면 아래 메시지를 출력합니다.  
                  <function int.bit_length()>
```

이 강의안에서는 객체의 방식^{method}도 편의상 함수라고 호칭하겠습니다.

객체의 생성 및 교환

- a와 b에 같은 문자열을 할당하고 a와 b를 출력합니다.

```
>>> a = b = 'Language'
```

```
>>> print(a); print(b) # [참고] 여러 줄 코드 사이에 세미콜론을 추가하면 한 줄로 작성할 수 있습니다.
```

- a와 b에 다른 문자열을 할당하고 a와 b를 출력합니다.

```
>>> a, b = 'Python', 'R' # [참고] 함수 실행 결과 2개 이상의 값을 반환할 때 변수를 각각 설정합니다.
```

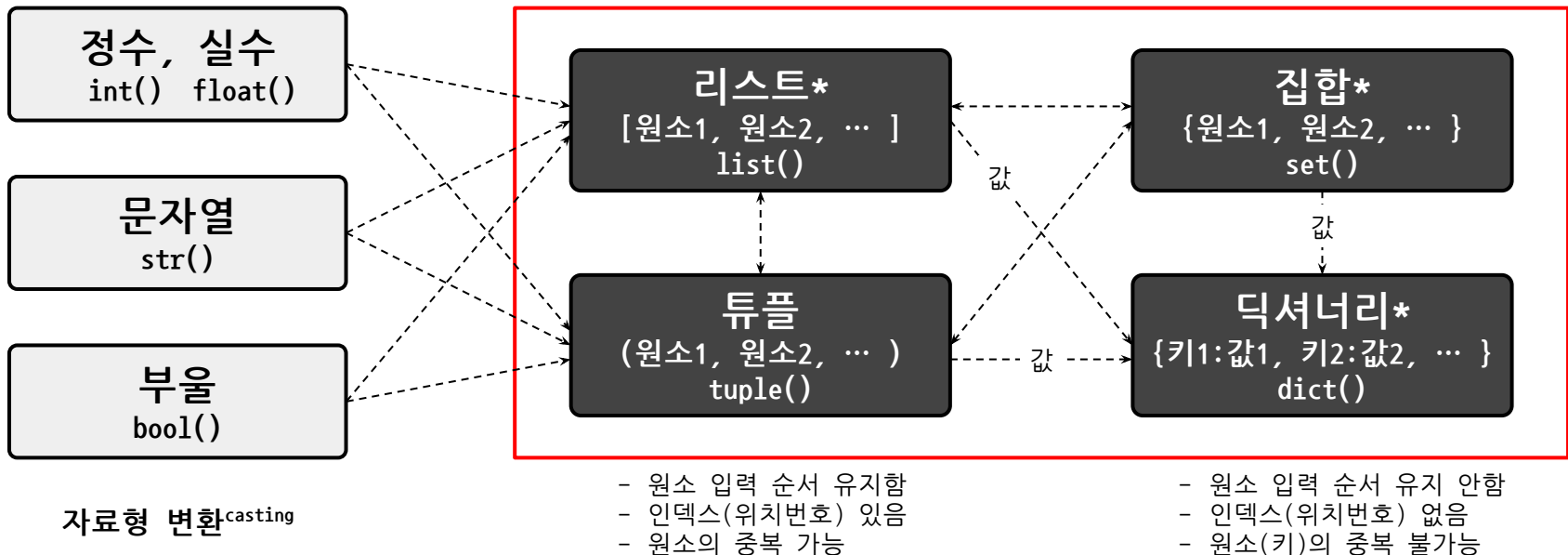
```
>>> print(a); print(b) # a는 'Python', b는 'R'을 출력합니다.
```

자료형

Python 자료형과 자료구조

자료형: 원소^{Base}

자료구조: 여러 원소를 담는 그릇^{Container}



*가 붙은 자료구조는 mutable이며 원소를 추가, 삭제 및 변경할 수 있습니다.

수: 정수, 실수

- 정수^{integer}는 소수점이 없는 자료형입니다.

```
>>> a = 3 # a에 정수 3을 할당합니다.
```

```
>>> a # a를 출력합니다.
```

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

- 실수^{float}는 소수점이 있는 자료형입니다.

```
>>> b = 3.0 # b에 실수 3.0을 할당합니다.
```

```
>>> b # b를 출력합니다.
```

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


산술 연산자

- 정수/실수로 산술 연산을 실행합니다.

연산자	상세 내용
$a + b$	<ul style="list-style-type: none"> a와 b를 더합니다. # [참고] $+a$는 단항 덧셈 연산자로 양의 부호를 의미합니다.
$a - b$	<ul style="list-style-type: none"> a에서 b를 뺍니다. # [참고] $-a$는 단항 뺄셈 연산자로 음의 부호를 의미합니다.
$a * b$	<ul style="list-style-type: none"> a와 b를 곱합니다.
$a ** b$	<ul style="list-style-type: none"> a를 b로 거듭제곱합니다.
a / b	<ul style="list-style-type: none"> a를 b로 나눕니다.
$a \% b$	<ul style="list-style-type: none"> a를 b로 나눈 나머지를 반환합니다.
$a // b$	<ul style="list-style-type: none"> a를 b로 나눈 (소수점 이하를 절사한) 몫을 반환합니다.

산술 연산자

- 두 변수로 산술 연산을 실행합니다.

>>> a + b # a와 b를 더합니다. 둘 다 정수면 정수를 반환하고 실수가 섞여 있으면 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 반환합니다. 정수 간 나눗셈도 마찬가지로입니다.

>>> a % b # a를 b로 나눈 나머지를 반환합니다.

>>> a // b # a를 b로 나눈 (소수점 이하를 절사한) 몫을 반환합니다.

할당 연산자

- 값 또는 산술 연산 결과를 변수에 할당합니다.

연산자	상세 내용
$a = b$	<ul style="list-style-type: none"> a에 b를 할당합니다.
$a += b$	<ul style="list-style-type: none"> a에 b를 더하고 a에 재할당합니다.
$a -= b$	<ul style="list-style-type: none"> a에서 b를 빼고 a에 재할당합니다.
$a *= b$	<ul style="list-style-type: none"> a에 b를 곱하고 a에 재할당합니다.
$a **= b$	<ul style="list-style-type: none"> a를 b로 거듭제곱하고 a에 재할당합니다.
$a /= b$	<ul style="list-style-type: none"> a를 b로 나누고 a에 재할당합니다.
$a \% = b$	<ul style="list-style-type: none"> a를 b로 나눈 나머지를 a에 재할당합니다.
$a //= b$	<ul style="list-style-type: none"> a를 b로 나눈 몫을 a에 재할당합니다.

할당 연산자

- 정수형 변수로 할당 연산을 실행합니다.

```
>>> a = 5; a # a에 정수 5를 할당하고 a를 출력합니다.
```

```
>>> a += 1; a # a에 1을 더하고 a에 재할당한 다음 a를 출력합니다.
```

```
>>> a -= 2; a # a에서 2를 빼고 a에 재할당한 다음 a를 출력합니다.
```

```
>>> b *= 3; b # b에 3을 곱하고 b에 재할당한 다음 b를 출력합니다.
```

```
>>> b /= 4; b # b를 4로 나누고 b에 재할당한 다음 b를 출력합니다.
```

비교 연산자

- 값의 크기를 비교하여 True 또는 False를 반환합니다.

연산자	상세 내용
$a > b$	• a가 b보다 크면 True, 작거나 같으면 False를 반환합니다.
$a \geq b$	• a가 b보다 크거나 같으면 True, 작으면 False를 반환합니다.
$a < b$	• a가 b보다 작으면 True, 크거나 같으면 False를 반환합니다.
$a \leq b$	• a가 b보다 작거나 같으면 True, 크면 False를 반환합니다.
$a == b$	• a가 b와 같으면 True, 다르면 False를 반환합니다.
$a != b$	• a가 b와 다르면 True, 같으면 False를 반환합니다.

[참고] True 또는 False를 진리값^{truth value}라고 합니다.

비교 연산자

- 변수와 상수로 비교 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, 4가 아니면 False를 반환합니다. [주의] 등호(=)를 2번 사용합니다.
```

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

논리 연산자

- 진리값 사이에서 논리곱, 논리합 또는 논리부정 연산을 실행합니다.

연산자	상세 내용
and	• [논리곱] 양쪽 진리값이 모두 True면 True, 아니면 False를 반환합니다.
or	• [논리합] 양쪽 진리값 중 하나라도 True면 True, 모두 False이면 False를 반환합니다.
not	• [논리부정] True를 False로, False를 True로 반전합니다.

[논리곱]

True	and	True	→	True
True		False		False
False		True		False
False		False		False

[논리합]

True	or	True	→	True
True		False		True
False		True		True
False		False		False

[논리부정]

not	True	→	False
	False		True

[참고] 연산자 우선순위

- Python 연산자의 우선순위^{precedence}를 정리한 표입니다.

연산자	상세 내용	연산자	상세 내용
()	• 괄호로 묶기	^	• 비트 연산자 배타적 논리합
**	• 거듭제곱		• 비트 연산자 논리합
~x	• 비트 연산자 논리부정	in, not in, <, <=, >, >=, ==, !=	• 멤버 연산자 • 비교 연산자
+x, -x	• 단항 덧셈, 단항 뺄셈	not	• 논리 연산자 논리부정
*, /, //, %	• 곱셈, 나눗셈, 몫, 나머지	and	• 논리 연산자 논리곱
+, -	• 덧셈, 뺄셈	or	• 논리 연산자 논리합
&	• 비트 연산자 논리곱		

높음



낮음

[참고] 괄호와 거듭제곱 사이에 자료구조(리스트, 튜플, 딕셔너리, 집합)의 생성, 함수, 슬라이싱, 인덱싱 및 객체.속성에 대한 우선순위를 적용합니다. 아울러 마지막에 람다 표현식에 대한 우선순위를 적용합니다.

논리 연산자

- 비교 연산을 실행하여 True 또는 False를 반환합니다.

```
>>> print(a > 3) # [참고] 같은 셀에 두 개 이상의 결과를 함께 출력하려면 print() 함수를 실행해야 합니다.
```

```
>>> print(b > 3)
```

- 비교 연산 결과로 논리 연산을 실행합니다.

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

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

```
>>> not (a > 3 and b > 3) # [논리부정] not 뒤에 오는 진리값을 반전합니다.  
[주의] 괄호로 감싸지 않으면 not 연산자는 바로 뒤 a > 3을 반전합니다.
```

```
>>> not (a > 3 or b > 3)
```

문자열

- 문자열^{string}은 여러 문자/숫자를 순서대로 나열하고 따옴표로 감싼 자료형입니다.

```
>>> str1 = 'Life is short, ' # str1에 문자열을 할당합니다. 홑따옴표를 사용했습니다.
```

```
>>> str1 # str1을 출력합니다.
```

```
>>> print(str1) # print() 함수를 실행하면 따옴표 없이 글자만 출력합니다.
```

```
>>> type(str1) # str1의 클래스를 확인합니다. str1의 클래스는 str입니다.
```

```
>>> len(str1) # str1의 글자수를 반환합니다.
```

```
>>> str2 = "you need Python!" # str2에 문자열을 할당합니다. 겹따옴표를 사용했습니다.
```

```
>>> str2 # str2를 출력합니다.
```

[참고] 따옴표 사용법

- 문자열을 생성하는 다양한 따옴표 사용법을 소개합니다.

```
>>> 'Monty Python's Flying Circus' # 홀따옴표로 감싼 문자열 안에 홀따옴표를 추가할 수 없습니다.  
[주의] 왼쪽 코드를 실행하면 에러가 발생합니다.
```

```
>>> "Monty Python's Flying Circus" # 문자열 안에 홀따옴표를 추가하려면 문자열을 홀따옴표 대신에  
겹따옴표로 감싸야 합니다.
```

```
>>> 'Monty Python\'s Flying Circus' # 홀따옴표로 감싼 문자열 안 홀따옴표 앞에 역슬래시(\) 기호를  
추가합니다. [참고] \ 기호는 정규표현식의 이스케이프입니다.
```

```
>>> 'I\'m saying "You need Python!" now.' # 두 가지 경우를 모두 포함하는 예제입니다.
```

```
>>> "I'm saying \"You need Python!\" now." # 문자열을 겹따옴표로 감쌌다면 문자열 안 겹따옴표  
앞에 \ 기호를 추가합니다.
```

```
>>> '''Hello! # 여러 줄의 문자열을 생성하려면 홀따옴표 또는 겹따옴표를 세 번 반복합니다.  
예) '''문자열''', """문자열"""
```

```
Good to see you.''' # 왼쪽 코드를 실행하면 '!'와 'G' 사이에 '\n'을 출력합니다.  
[참고] '\n'은 줄바꿈입니다.
```

문자열 연산자

- + 연산자는 두 문자열을 결합합니다.

```
>>> str1 + str2 # [주의] 수와 문자열을 섞으면 에러가 발생합니다.
```

- * 연산자는 왼쪽 문자열을 오른쪽에 지정한 정수만큼 반복합니다.

```
>>> str2 * 3 # [주의] 실수를 지정하면 에러가 발생합니다.
```

- 멤버 연산자 in은 오른쪽 문자열에 왼쪽 문자열이 있으면 True, 없으면 False를 반환합니다. not in은 반대로 동작합니다.

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

```
>>> 'a' not in str1 # str1에 문자열 'a'가 없으면 True, 있으면 False를 반환합니다.
```

문자열 인덱싱

- 문자열 인덱싱^{indexing}은 인덱스(위치번호)로 문자를 선택합니다.

인덱스	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
문자	아	버	지	가		안	방	에		들	어	가	신	다	.

- 문자열에 대괄호를 추가하고 대괄호 안에 선택할 문자의 인덱스를 지정합니다.

```
>>> sen = '아버지가 안방에 들어가신다.' # 실습할 문자열을 생성합니다.
```

```
>>> sen[0] # sen의 0번 인덱스(첫 번째) 문자를 선택합니다. 왼쪽 코드를 실행하면 '아'를 반환합니다.
           [주의] Python 인덱스는 0부터 시작합니다!
```

```
>>> sen[1] # sen의 1번 인덱스(두 번째) 문자를 선택합니다. 왼쪽 코드를 실행하면 '버'를 반환합니다.
```

```
>>> sen[-1] # sen의 -1번 인덱스(마지막) 문자를 선택합니다. 인덱스 앞에 마이너스 부호를 추가하면 마지막에서
            거꾸로 시작합니다. [참고] -0은 0과 같습니다!
```

문자열 슬라이싱

- 문자열 슬라이싱^{slicing}은 시작과 끝 인덱스를 콜론(:)으로 연결한 슬라이스^{slice}로 연속한 문자열을 선택합니다.(시작:끝+1) # [주의] 끝 인덱스(콜론 오른쪽 정수)를 포함하지 않으므로 끝 인덱스에 1을 더한 값을 지정해야 합니다.

```
>>> sen[0] + sen[1] + sen[2] # sen의 0~2번 인덱스 문자를 결합합니다.
```

```
>>> sen[0:3] # sen의 0번 인덱스(첫 번째)부터 3-1번 인덱스(세 번째)까지 연속한 문자를 선택합니다.
```

```
>>> sen[:3] # 콜론 앞 정수를 생략하면 첫 번째 문자부터 선택합니다.
```

```
>>> sen[9:] # 콜론 뒤 정수를 생략하면 마지막 문자까지 선택합니다.
```

```
>>> sen[:] # 빈 콜론을 입력하면 처음부터 끝까지 연속한 문자를 선택합니다.  
[주의] 대괄호 안에 아무것도 입력하지 않으면 에러가 발생합니다.
```

문자열 공백 제거 및 변경

- 문자열의 속성과 방식으로 문자열을 전처리합니다.

```
>>> sen = '\n 나의 살던 고향은 꽃피는 산골 \t' # 문자열 양쪽에 공백을 추가합니다.  
[참고] '\n'은 줄바꿈, '\t'는 탭입입니다.
```

```
>>> sen # sen을 출력하면 문자열 양쪽에 공백이 보입니다.
```

```
>>> print(sen) # print() 함수는 따옴표와 공백 없이 글자만 출력하므로 문자열의 공백을 확인할 수 없습니다.
```

```
>>> dir(sen) # sen의 속성과 방식 목록을 출력합니다.
```

```
>>> sen = sen.strip(); sen # sen에서 양쪽 공백을 제거하고 sen에 재할당합니다.  
[참고] lstrip() 함수는 왼쪽,rstrip() 함수는 오른쪽 공백을 제거합니다.
```

```
>>> sen.replace('산골', '공원') # sen에서 '산골'을 '공원'으로 바꾼 결과를 반환합니다.  
[참고] 세 번째 인수에 변경할 횟수를 정수로 지정할 수 있습니다.
```

문자열 인덱스 확인

- 관심 있는 문자(열)의 개수와 인덱스를 확인합니다.

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

```
>>> sen.index(' ') # sen에서 공백이 처음 나오는 인덱스를 반환합니다.  
[참고] index() 함수에 탐색 범위(시작과 끝 인덱스)를 설정할 수 있습니다.
```

```
>>> sen.index(' ', 3, len(sen)) # 탐색 범위(3번 인덱스부터 마지막)를 지정하고 공백을 탐색합니다.
```

```
>>> sen.index('산골') # 두 글자 이상인 문자열을 지정하면 첫 번째 글자의 인덱스를 반환합니다.
```

```
>>> sen.index('공원') # sen에 없는 문자열을 지정하면 에러가 발생합니다.
```

```
>>> sen.find('공원') # find() 함수는 index() 함수와 같은 기능을 수행하지만 에러 대신 -1을 반환합니다.
```


문자열 분리 및 결합

- 문자열을 분리하거나 결합합니다.

```
>>> sen.split('은') # sen을 구분자 '은'으로 분리합니다. [참고] 문자열을 분리하면 리스트로 반환합니다.
```

```
>>> strs = sen.split() # sen을 공백으로 분리하고 strs에 할당합니다.
```

```
>>> strs # strs를 출력합니다. 대괄호 안에 여러 문자열이 있습니다.
```

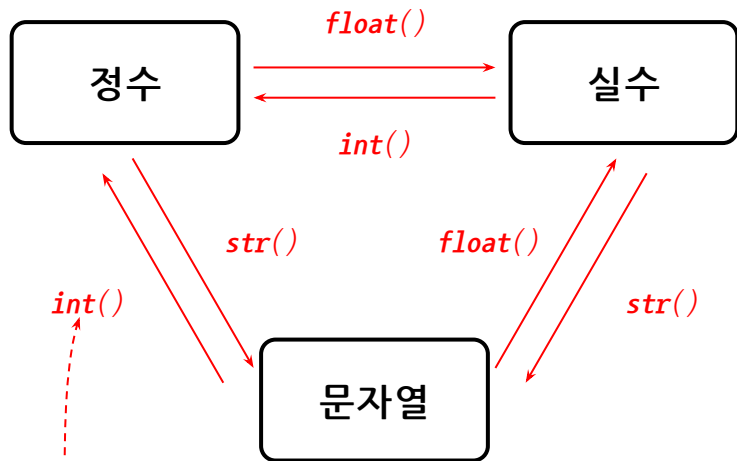
```
>>> type(strs) # strs의 클래스를 확인합니다. strs의 클래스는 list입니다.
```

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

```
>>> '-'.join(strs) # 구분자에 따라 실행 결과가 달라집니다.
```

자료형 변환

- 자료형 변환^{casting} 관계도입니다.
 - 클래스 함수로 자료형을 변환합니다.



[주의] 소수점 있는 문자열은 정수로 변환할 수 없으므로 실수로 변환한 다음 정수로 변환합니다.

- 자료형을 변환합니다.

```
>>> a = '1.2'; a # a에 문자열 '1.2'를 할당합니다.
```

```
>>> type(a) # a의 클래스는 str입니다.
```

```
>>> b = float(a); b # a를 실수로 변환하고 b에 할당합니다.
```

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

```
>>> c = int(a) # [주의] 소수점 있는 문자열을 정수로 변환할 수 없습니다.
```

```
>>> c = int(b); c # b를 정수로 변환하고 c에 할당합니다.
```

```
>>> type(c) # c의 클래스는 int입니다.
```

문자열 포매팅: 포맷 코드

- 문자열 포매팅은 문자열에서 원하는 위치에 포맷 코드를 추가하고, 변수에 할당된 값을 포맷 코드로 전달하여 문자열에 대입합니다.

%s(문자열)	%d(정수)	%f(실수)	%('%' 출력)
---------	--------	--------	-----------

- 문자열 결합과 포매팅 코드를 비교합니다.

```
>>> name = '홍길동'; rate = 3.2 # name과 rate에 적당한 값을 할당합니다.
```

```
>>> name + ' 고객님의 이자율은 ' + str(rate) + ' %입니다.' # 여러 문자열을 결합합니다.
```

```
>>> '%s 고객님의 이자율은 %f %입니다.' %(name, rate) # name과 rate에 할당된 값을 포맷 코드(%s와 %f)에 대입합니다.
```

```
>>> '%s 고객님의 이자율은 %.2f %입니다.' %(name, rate) # 실수는 소수점 자리수를 설정할 수 있습니다.
```

문자열 포매팅: format(), f-문자열

- format() 함수는 포맷 코드 대신 중괄호를 사용합니다.

```
>>> '{0} 고객님의 이자율은 {1} %입니다.'.format(name, rate) # 중괄호 안에 인덱스 또는  
# 변수명을 지정합니다.
```

```
>>> '{} 고객님의 이자율은 {} %입니다.'.format(name, rate) # 인덱스를 생략하면 변수값을  
# 순서대로 대입합니다.
```

```
>>> '{} 고객님의 이자율은 {:.2f} %입니다.'.format(name, rate)  
# [주의] 실수의 소수점 자리수를 지정하려면 % 대신 콜론을 추가합니다.
```

- Python 3.6에서 도입한 f-문자열을 사용하면 코드 가독성을 높일 수 있습니다.

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

f-문자열은 따옴표 앞에 f를 추가합니다. 만약 f를 추가하지 않으면 중괄호 안 내용을 문자 그대로 출력합니다.

부울

- 부울^{bool}은 True 또는 False를 표현하는 자료형입니다.

```
>>> a = True # a에 True를 할당합니다. [주의] 따옴표로 감싸면 문자열이 됩니다.
```

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

```
>>> True + False # True는 정수 1, False는 정수 0이므로 진리값을 더하면 True 개수를 반환합니다.
```

- 어떤 객체에 값 또는 원소가 있으면 True, 없으면 False를 반환합니다.

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

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

자료구조

리스트

- 리스트^{List}는 수, 문자열 또는 리스트 등 다양한 원소를 갖는 자료구조입니다.

```
>>> a = []; a # 빈 리스트를 생성합니다. 리스트를 생성할 때 대괄호 안에 원소를 콤마로 나열합니다.  
[참고] a = list()를 실행한 결과와 같습니다.
```

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

```
>>> b = [1, 5, 3, 3]; b # 리스트는 원소를 입력한 순서를 유지하며, 원소의 중복을 허용합니다.
```

```
>>> len(b) # 리스트의 길이(원소 개수)를 반환합니다. [참고] 문자열은 글자수를 반환합니다.
```

```
>>> c = [1, 2.0, '3']; c # 리스트는 다양한 자료형을 원소로 가질 수 있습니다.
```

```
>>> [a, b, c] # 리스트는 리스트를 원소로 가질 수 있습니다.
```

리스트 연산자

- + 연산자는 두 리스트를 결합합니다.

```
>>> b + c # b와 c를 결합하고 하나의 리스트로 반환합니다.
```

- * 연산자는 왼쪽 리스트를 오른쪽에 지정한 정수만큼 반복합니다.

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

- 멤버 연산자 **in**은 오른쪽 리스트에 왼쪽 원소가 있으면 True, 없으면 False를 반환합니다. **not in**은 반대로 동작합니다.

```
>>> 1 in b # b에 정수 1이 있으면 True, 없으면 False를 반환합니다.
```

```
>>> 1 not in b # b에 정수 1이 없으면 True, 있으면 False를 반환합니다.
```


리스트 인덱싱

- 리스트 인덱싱^{indexing}은 인덱스(위치번호)로 원소를 선택합니다.

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

- 리스트에 대괄호를 추가하고 대괄호 안에 선택할 원소의 인덱스를 지정합니다.

```
>>> a = [1, 2, '3', '4', ['가', '나', '다']] # 실습할 리스트를 생성합니다.
```

```
>>> a[0] + a[1] # a의 0번과 1번 인덱스 원소를 더합니다.
```

```
>>> a[2] + a[3] # a의 2번과 3번 인덱스 원소를 결합합니다.
```

```
>>> a[4][0] # a의 4번 인덱스 원소는 리스트이므로 대괄호를 이어서 사용하면 리스트의 원소를 반환합니다.
```

리스트 슬라이싱

- 리스트 슬라이싱^{slicing}은 시작과 끝 인덱스를 콜론(:)으로 연결한 슬라이스^{slice}로 연속한 원소를 선택합니다. (시작:끝+1) # [주의] 끝 인덱스(콜론 오른쪽 정수)를 포함하지 않으므로 끝 인덱스에 1을 더한 값을 지정해야 합니다.
- 리스트 슬라이싱은 실행 결과를 항상 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 4번~마지막 원소를 선택하고 리스트로 반환합니다.
```

[참고] 리스트 인덱싱 및 슬라이싱 관련 주의사항

- 리스트 인덱싱 및 슬라이싱과 관련하여 주의해야 할 내용을 정리한 것입니다.

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

```
>>> a[:] # 빈 콜론을 지정하면 리스트의 전체 원소를 반환합니다.
```

```
>>> a[0] # 대괄호 안에 인덱스를 스칼라(길이가 1인 값)로 지정하면 해당 원소를 본래 자료형으로 반환합니다.
```

```
>>> a[0:1] # 대괄호 안에 콜론을 사용한 슬라이스를 지정하면 항상 리스트로 반환합니다.
```

```
>>> a[[0, 2]] # 대괄호 안에 리스트를 지정하면 에러가 발생합니다.  
[주의] 대괄호 안에 인덱스 스칼라 또는 콜론으로 연결한 슬라이스만 지정할 수 있습니다.
```

```
>>> [a[0], a[2]] # 리스트의 0번과 2번 인덱스 원소를 선택하고 리스트로 반환합니다.
```

range() 함수 사용법

- range() 함수는 연속한 정수를 반환합니다. # [주의] range() 함수는 정수만 지정할 수 있습니다! 실수를 지정하면 에러가 발생합니다.

```
>>> a = range(5); a # 0부터 4까지 연속한 정수를 a에 할당하고 a를 출력합니다.
```

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

```
>>> a = list(a); a # a를 리스트로 변환하여 a에 재할당하고 a를 출력합니다.  
[참고] list()는 괄호 안 변수를 리스트로 변환하는 클래스 함수입니다.
```

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

```
>>> list(range(1, 6)) # 0이 아닌 정수로 시작하려면 range() 함수에 시작과 끝+1 정수를 지정합니다.
```

```
>>> list(range(1, 11, 2)) # 간격을 설정하려면 range() 함수에 시작, 끝+1 및 간격을 지정합니다.
```

이중 콜론 연산자

- 이중 콜론 연산자는 슬라이스에 간격을 설정한 것입니다.(시작:끝+1:간격)

```
>>> a = list(range(1, 8)); a # 1부터 7까지 연속한 정수를 원소로 갖는 리스트를 a에 할당합니다.
```

```
>>> a[1:5:2] # 1~4번 인덱스 원소에서 두 칸 간격으로 원소를 선택하고 리스트로 반환합니다.
```

```
>>> a[:5:2] # 0~4번 인덱스 원소에서 두 칸 간격으로 원소를 선택하고 리스트로 반환합니다.
```

```
>>> a[1::2] # 1번 인덱스 원소부터 마지막 원소까지 두 칸 간격으로 원소를 선택하고 리스트로 반환합니다.
```

```
>>> a[::-2] # 처음부터 마지막 원소까지 두 칸 간격으로 원소를 선택하고 리스트로 반환합니다.
```

```
>>> a[::-1] # 전체 원소를 역순으로 반환합니다. [주의] 내림차순 정렬이 아닙니다.  
[참고] 간격에 -2를 지정하면 역순으로 변경하고 두 칸 간격으로 원소를 선택합니다.
```

리스트 원소 추가 및 삽입

- **append()** 함수는 지정한 값 그대로 리스트의 마지막 원소로 추가합니다.

```
>>> a.append(5); a # 정수 5를 a의 마지막 원소로 추가합니다.
```

```
>>> a.append([6, 7]); a # 리스트 [6, 7]을 a의 마지막 원소로 추가합니다.
```

- **extend()** 함수는 리스트만 입력받고 리스트 원소를 마지막 원소로 추가합니다.

```
>>> a.extend([8, 9]); a # 정수 8, 9를 a의 마지막 원소로 추가합니다.
```

- **insert()** 함수는 지정한 인덱스 앞에 값을 삽입합니다.(인덱스, 값)

```
>>> a.insert(1, [6, 7]); a # a의 1번 인덱스에 리스트 [6, 7]을 삽입합니다.
```

[참고] **append()**, **extend()**, **insert()** 함수는 실행 결과를 리스트에 반영하므로 리스트에 재할당하지 않습니다.

리스트 원소 삭제

- `remove()` 함수는 지정한 값을 (여러 개 있어도) 한 번만 삭제합니다.

```
>>> a.remove([6, 7]); a # a에서 원소 [6, 7]을 삭제합니다.
```

```
>>> a.remove([6, 7]); a # a에서 원소 [6, 7]을 한 번 더 삭제합니다.
```

```
>>> a.remove([6, 7]) # a에 원소 [6, 7]이 없으므로 에러가 발생합니다.  
[참고] remove() 함수에 없는 원소를 지정하면 에러가 발생합니다.
```

- `pop()` 함수는 지정한 인덱스 원소를 출력하고 해당 원소를 삭제합니다.

```
>>> a.pop(6); a # a의 6번 인덱스 원소를 출력하고 해당 원소를 삭제합니다.
```

```
>>> a.pop(); a # a의 마지막 원소를 출력하고 해당 원소를 삭제합니다.  
[참고] pop() 함수에 인덱스를 생략하면 마지막 원소를 제거합니다.
```

[참고] `remove()`, `pop()` 함수는 실행 결과를 리스트에 반영하므로 리스트에 재할당하지 않습니다.

리스트 원소 변경

- 인덱스와 슬라이싱을 이용하여 원하는 위치의 원소를 변경합니다.

```
>>> a[1] = 6; a # a의 1번 인덱스 원소를 정수 6으로 변경합니다.
```

```
>>> a[3:5] = [7, 8]; a # a의 3~4번 인덱스 원소를 리스트 [7, 8]의 원소로 변경합니다.  
[참고] 왼쪽 리스트 원소 개수와 오른쪽 리스트 원소 개수가 달라도 됩니다.
```

```
>>> a[2] = [3, 4]; a # a의 2번 인덱스 원소를 리스트 [3, 4]로 변경합니다.
```


리스트 원소 확인

- `count()` 함수는 지정한 원소 개수(없으면 0)를 반환합니다.

```
>>> a.count(6) # a에서 원소 6의 개수를 반환합니다.
```

```
>>> a.count(3) # a에 원소 3이 없으므로 0을 반환합니다.
```

- `index()` 함수는 지정한 원소의 첫 번째 인덱스(없으면 에러)를 반환합니다.

```
>>> a.index(6) # a에서 원소 6이 처음 나오는 인덱스를 반환합니다.
```

```
>>> a.index(3) # a에 원소 3이 없으므로 에러가 발생합니다.
```

```
>>> a.find(3) # 리스트에 find() 함수가 없으므로 에러가 발생합니다. 에러 메시지는 아래와 같습니다.  
AttributeError: 'list' object has no attribute 'find'
```

리스트 원소 정렬

- `sort()` 함수는 리스트 원소를 오름차순/내림차순 정렬합니다.

```
>>> a.sort() # 리스트 원소에 자료형이 섞여 있으면 원소를 정렬할 수 없으므로 에러가 발생합니다.  
[참고] sort() 함수는 리스트 원소의 클래스가 모두 같을 때 정상 실행됩니다.
```

```
>>> a.remove([3, 4]) # a에서 원소([3, 4])를 삭제합니다.
```

```
>>> a.sort(); a # a의 원소를 오름차순 정렬합니다.  
[참고] sort() 함수는 실행 결과를 리스트에 반영하므로 리스트에 재할당하지 않습니다.
```

```
>>> a.sort(reverse = True); a # a의 원소를 내림차순 정렬합니다.  
[참고] reverse 매개변수에 전달하는 인수의 기본값은 False입니다.
```

- `reverse()` 함수는 리스트 원소를 역순으로 변경합니다.

```
>>> c.reverse(); c # [참고] reverse() 함수는 리스트 원소의 클래스가 달라도 에러가 발생하지 않습니다.  
reverse() 함수도 실행 결과를 리스트에 반영합니다.
```

[참고] 함수/방식의 Docstring 사용법

- 함수 괄호에 커서를 놓고 [shift] + [tab] 키를 누르면 Docstring을 출력합니다.

```
In [ ]: a.sort()
```

```
Signature: a.sort(*, key=None, reverse=False)
Docstring:
Sort the list in ascending order and return None.
```

```
Signature: a.sort(*, key=None, reverse=False)
Docstring:
Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them,
ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.
Type: builtin_function_or_method
```

^ 버튼을 클릭하면 맨 아래에 Docstring을 출력합니다.
[tab] 키를 한 번 더 누르면 제자리에서 팝업을 펼칩니다.

reverse는 플래그 변수이고, 인수 기본값은 False입니다.
아래에 reverse 관련 설명이 있습니다.

[참고] 플래그 변수는 특정 동작의 수행 여부를 불리언 (1비트) 값을 갖습니다.

[참고] 함수 뒤에 괄호 없이 물음표를 추가하고 실행하면 같은 결과를 얻습니다.

튜플

- 튜플^{Tuple}은 원소를 변경할 수 없는 리스트입니다.

```
>>> t = (1, 2.0, '3'); t
```

튜플을 생성합니다. 튜플을 생성할 때 소괄호 안에 원소를 콤마로 나열합니다.
[참고] 여러 값을 소괄호로 묶지 않아도 튜플로 패키징^{packing}합니다.

```
>>> type(t)
```

t의 클래스를 확인합니다. t의 클래스는 tuple입니다.

```
>>> tuple(c)
```

tuple() 클래스 함수는 리스트를 튜플로 변환합니다.

```
>>> list(t)
```

list() 클래스 함수는 튜플을 리스트로 변환합니다.

```
>>> t.append(4)
```

튜플에 append() 함수가 없으므로 에러가 발생합니다. 에러 메시지는 아래와 같습니다.
AttributeError: 'tuple' object has no attribute 'append'

```
>>> t.remove(1)
```

튜플에 remove() 함수가 없으므로 에러가 발생합니다. 에러 메시지는 아래와 같습니다.
AttributeError: 'tuple' object has no attribute 'remove'

```
>>> t[2] = 3
```

튜플은 원소를 할당할 수 없으므로 에러가 발생합니다. 에러 메시지는 아래와 같습니다.
TypeError: 'tuple' object does not support item assignment

딕셔너리

- 딕셔너리^{Dictionary}는 키^{key}와 값^{value}을 콜론으로 결합한 원소를 갖는 자료구조입니다.
 - 딕셔너리를 생성할 때 중괄호 안에 'key': value를 콤마로 연결합니다.
 - 키에 스칼라(수/문자열), 값에 스칼라, 리스트, 튜플 또는 딕셔너리를 지정합니다.
- 딕셔너리는 원소의 순서를 유지하지 않습니다. # [참고] Python 3.6 이후로 원소의 순서를 유지하는 `OrderedDict` 클래스와 동일하게 동작합니다.
- 딕셔너리는 키로 인덱싱합니다. # [참고] 딕셔너리에는 정수 인덱스가 없으므로 정수 인덱싱이 불가능합니다. 만약 키를 정수로 지정하면 비슷하게 코딩할 수 있습니다.
 - 딕셔너리 원소를 추가, 삭제 및 변경할 때에도 키 인덱싱을 사용합니다.
- 딕셔너리는 키의 중복을 허용하지 않습니다. 따라서 같은 키에 다른 값을 여러 번 할당하면 가장 마지막에 할당한 값으로 업데이트합니다.

딕셔너리 생성

- 딕셔너리를 생성하고 클래스를 확인합니다.

```
>>> d = {'item': 'pants', 'size': ['S', 'M', 'L']} # [주의] 키가 문자열일 때 따옴표를  
# 감싸지 않으면 변수로 인식합니다.
```

```
>>> d # d를 출력합니다.
```

```
>>> type(d) # d의 클래스를 확인합니다.  
# d의 클래스는 dict입니다.
```

키	'item'	'size'
	'pants'	['S', 'M', 'L']

- 같은 값을 원소로 갖는 리스트와 비교합니다.

```
>>> a = ['pants', ['S', 'M', 'L']]
```

```
>>> a
```

인덱스	0	1
	'pants'	['S', 'M', 'L']

[참고] dict() 클래스 함수 사용법

- dict() 클래스 함수는 키를 따옴표로 감싸지 않고 콜론 대신 등호를 사용합니다.

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

[주의] 시각화 코드에 딕셔너리를 지정할 때 **dict()** 함수를 자주 사용합니다.

- 키를 따옴표로 감싸면 에러가 발생합니다.

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

- 등호 대신 콜론을 사용하면 에러가 발생합니다.

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

- 빈 딕셔너리를 생성합니다.

```
>>> dict()
```

딕셔너리 연산자와 인덱싱

- 멤버 연산자 `in`은 딕셔너리에 키가 있으면 `True`, 없으면 `False`를 반환합니다.

```
>>> 'item' in d # d에 'item'인 키가 있으면 True, 없으면 False를 반환합니다.
```

```
>>> 'shop' in d # d에 'shop'인 키가 있으면 True, 없으면 False를 반환합니다.
```

- 딕셔너리는 키로 인덱싱합니다.

```
>>> d['item'] # d에서 키가 'item'인 값을 반환합니다.
```

```
>>> d['shop'] # d에서 키가 'shop'인 값을 반환합니다.  
[주의] 딕셔너리에 없는 키를 입력하면 에러가 발생합니다.
```


딕셔너리 원소 추가, 삭제 및 변경

- 딕셔너리에 원소를 추가합니다.

```
>>> d['shop'] = 'A1'; d # d에 키가 'shop', 값이 'A1'인 원소를 추가합니다.
```

- 딕셔너리의 원소를 삭제합니다.

```
>>> del d['item']; d # d에서 키가 'item'인 원소를 삭제합니다.  
[참고] del문은 예약어keyword이고, 변수 또는 변수의 원소를 삭제합니다.
```

- 딕셔너리의 원소를 변경합니다.

```
>>> d['shop'] = 'A2'; d # d에서 키가 'shop'인 값을 'A2'로 변경합니다.
```

[참고] 딕셔너리 키와 값 반환

- 딕셔너리의 키와 값을 반환합니다.

```
>>> d.keys() # keys() 함수는 딕셔너리의 키를 dict_keys 클래스로 반환합니다.
```

```
>>> d.values() # values() 함수는 딕셔너리의 값을 dict_values 클래스로 반환합니다.
```

```
>>> d.items() # items() 함수는 딕셔너리의 키와 값의 쌍을 튜플로 묶어서 dict_items 클래스로 반환합니다.
```

- 반복문으로 딕셔너리의 키와 값을 차례대로 출력합니다.

```
>>> for item in d.items():
```

```
    print(item) # 변수 item은 d의 키와 값의 쌍을 원소로 갖는 튜플입니다.
```

집합

- 집합^{Set}은 원소의 중복을 허용하지 않고 순서도 없는 자료구조입니다.

```
>>> s = {3, 1, 2, 1}; s
```

집합을 생성합니다. 집합을 생성할 때 중괄호 안에 원소를 콤마로 나열합니다.
[참고] 집합은 중복 원소를 제거하고 오름차순 정렬합니다.

```
>>> type(s)
```

s의 클래스를 확인합니다. s의 클래스는 set입니다.

```
>>> set(b)
```

set() 클래스 함수는 리스트를 집합으로 변환합니다.

```
>>> list(s)
```

list() 클래스 함수는 집합을 리스트로 변환합니다.

```
>>> s & set(b)
```

두 집합의 교집합을 반환합니다. [참고] s.intersection(set(b))를 실행한 결과가 같습니다.

```
>>> s | set(b)
```

두 집합의 합집합을 반환합니다. [참고] s.union(set(b))를 실행한 결과가 같습니다.

```
>>> s - set(b)
```

두 집합의 차집합을 반환합니다. [참고] s.difference(set(b))를 실행한 결과가 같습니다.

[참고] 집합 원소 생성 기준

- 리스트를 집합으로 변환할 때 중복 원소는 아래 기준으로 제거합니다.
 - 정수와 실수는 앞선 자료형을 적용합니다. ex) $\{1, 1.0, 2.0, 2\} \rightarrow \{1, 2.0\}$
 - 문자열은 정수/실수와 별개로 존재합니다. ex) $\{1, '1'\} \rightarrow \{1, '1'\}$
- 집합 원소의 정렬 순서는 다음과 같습니다.
 - 정수, 문자열, 실수 순서로 정렬합니다. ex) $\{1.0, 1, '1'\} \rightarrow \{'1', 1.0\}$
 - 교집합은 오른쪽 집합의 자료형을 따릅니다. ex) $\{1\} \& \{1.0\} \rightarrow \{1.0\}$
 - 합집합은 왼쪽 집합의 자료형을 따릅니다. ex) $\{1\} \mid \{1.0\} \rightarrow \{1\}$
 - 차집합은 왼쪽 집합의 자료형을 따릅니다. ex) $\{1, 2\} - \{1.0\} \rightarrow \{2\}$

제어문: 조건문

if 조건문 기본 구조

- if 조건문은 지정한 조건 만족 여부에 따라 실행할 코드를 분기합니다.

```
>>> if 조건1:    # if문에 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 조건이 있으면 elif문을 사용하고, 추가 조건이 없으면 생략합니다.
                  # elif문만 단독으로 실행할 수 없고, if문 다음에 여러 번 추가할 수 있습니다.
    코드B        # 조건2를 만족하면 코드B를 실행하고 조건문을 종료합니다.
else:            # 모든 조건을 만족하지 않을 때 마지막으로 실행할 코드가 있으면 else문을 사용합니다.
                  # else문도 단독으로 실행할 수 없고, 생략하거나 마지막에 한 번 추가합니다.
    코드C        # 모든 조건을 만족하지 않으면 코드C를 실행하고 조건문을 종료합니다.

# [중요] if 조건문에 지정하는 조건 코드는 실행 결과로 True 또는 False를 하나만 반환해야 합니다!
# 만약 진리값을 두 개 이상 반환하면 에러가 발생합니다.
```

[참고] 탭 vs 공백

- 조건문, 반복문 또는 사용자 정의 함수의 콜론 다음에 [enter] 키를 누르면 탭 들여쓰기를 지원합니다.
- Python은 탭 크기를 공백 4칸으로 설정합니다.

```
>>> if 조건1:
```

```
    코드A # 공백 4칸과 탭 1번을 섞어 쓰지 않는 것이 좋습니다.
```

```
elif 조건2:
```

```
    코드B # [참고] IDE마다 탭 크기가 다르므로 공백 대신 탭을  
        사용하지 않는 것이 좋습니다.
```

Jupyter Notebook은 탭을 공백 4칸으로 자동 변경합니다.

When using spacebar
instead of tabs



콘솔에서 문자열 입력

- 전역 변수 `global variable` 목록을 출력합니다.

```
>>> %whos # %whos를 실행하면 전역 변수 목록을 출력합니다.  
[참고] Jupyter Notebook을 열면 전역 변수 목록은 비어있습니다.
```

- 콘솔에서 값을 입력받고 변수에 문자열로 할당합니다.

```
>>> score = input('점수: ') # 콘솔에 '점수: '를 출력하고 커서를 깜빡이다가 사용자가 입력한 값을  
                           score에 할당하고 전역 변수 목록에 추가합니다.
```

```
>>> score # score를 출력합니다. score는 따옴표로 감싼 문자열입니다.  
[참고] 전역 변수 목록에 없는 변수를 출력하려고 하면 에러가 발생합니다.
```

- `score`를 실수로 변환합니다.

```
>>> score = float(score)
```

```
>>> score # score를 출력합니다. score는 따옴표 없이 소수점이 추가된 실수입니다.
```


if 조건문 실습

- 조건문을 실행하고 점수에 따라 합격 여부를 출력합니다.

```
>>> if score >= 90:
```

```
    print('합격') # 첫 번째 조건을 만족하면 왼쪽 코드를 실행하고 조건문을 종료합니다.
```

```
elif score >= 80:
```

```
    print('재검사') # 두 번째 조건을 만족하면 왼쪽 코드를 실행하고 조건문을 종료합니다.
```

```
else:
```

```
    print('불합격') # 모든 조건을 만족하지 않으면 왼쪽 코드를 실행하고 종료합니다.
```

제어문: 반복문

for 반복문 기본 구조

- 반복문은 여러 줄의 코드에서 일부 값을 바꾸면서 반복 실행할 때 사용합니다.
 - for 반복문과 while 반복문이 있으며 상황에 따라 알맞는 반복문을 선택합니다.
- for 반복문은 반복 실행할 범위가 정해져 있을 때 사용합니다.

>>> for 변수 in 리스트: *# 변수는 리스트 원소를 처음부터 입력받아 콜론 아래 코드를 반복 실행합니다.
리스트 대신 튜플, 딕셔너리 또는 문자열을 지정할 수 있습니다.*

코드A

코드B

코드C

왼쪽 코드에는 for 반복문에서 설정한 변수를 포함하는 경우가 일반적입니다.

for 반복문 실습

- 중식당 메뉴로 리스트를 생성하고 반복문으로 리스트 원소를 차례대로 출력합니다.

```
>>> menu = ['짜장면', '탕수육', '깐풍기', '짬뽕', '전가복', '삭스핀']
```

```
>>> for item in menu:
```

```
    print(item) # [주의] 콜론 아래 코드에서 변수를 출력하려면 반드시 print() 함수를 사용해야 합니다!
```

- 여러 객체를 결합한 문자열을 출력합니다.

```
>>> for item in menu:
```

```
    print(item, '시킬까요?', sep = ' ', end = '\n')
```

print() 함수는 여러 객체를 *sep* 매개변수에 지정한 구분자로 결합한 문자열을 출력합니다.(기본값: ' ')
[참고] *end* 매개변수에는 마지막에 추가할 문자열을 지정합니다.(기본값: '\n')

반복문 안에 조건문 추가

- 반복문 안에 조건문을 추가하여 코드를 제어합니다.

```
>>> for item in menu:
```

```
    print(item, '시킬까요?')
```

```
    if item in ['짜장면', '짬뽕']: # 변수 item이 '짜장면' 또는 '짬뽕'이면 아래 코드를 실행
                                   합니다.
```

```
        print('-> 요리부터 주문합시다!')
```

```
    print('-> 다음 메뉴는 뭔가요?\n') # 조건문과 상관없이 왼쪽 코드를 항상 실행합니다.
```

반복문 제어: continue

- 반복문에서 continue를 만나면 아래 코드를 실행하지 않고 처음으로 되돌아갑니다.

```
>>> for item in menu:
```

```
    print(item, '시킬까요?')
```

```
    if item in ['짜장면', '짬뽕']:
```

```
        continue # 반복문 실행 도중 continue를 만나면 처음으로 되돌아갑니다.
```

```
    print('-> 요리부터 주문합시다!') # continue는 왼쪽 코드를 실행하지 않습니다.
```

```
print('-> 다음 메뉴는 뭔가요?\n') # 변수 item이 '짜장면' 또는 '짬뽕'이면 실행하지 않습니다.
```

반복문 제어: break

- 반복문에서 break를 만나면 반복문을 중단합니다.

```
>>> for item in menu:
```

```
    print(item, '시킬까요?')
```

```
    if item in ['전가복', '삭스핀']:
```

```
        break # 반복문 실행 도중 break를 만나면 반복문을 중단합니다.
```

```
    print('-> 요리부터 주문합시다!') # break 때문에 왼쪽 코드를 실행할 수 없습니다.
```

```
print('-> 다음 메뉴는 뭔가요?\n') # 변수 item이 '전가복' 또는 '삭스핀'이면 반복문을 중단합니다.
```

[참고] 중첩 for 반복문

- 두 리스트의 원소로 가능한 모든 경우의 수를 반복 실행합니다.(구구단 만들기)

```
>>> for i in range(2, 10): # 변수 i는 2를 입력받아 안쪽 반복문을 실행합니다.  
                           # 안쪽 반복문을 완료하면 변수 i는 다음 원소를 입력받습니다.
```

```
    print(f'*** {i}단 ***') # 변수 i에 할당한 값을 대입합니다.
```

```
        for j in range(1, 10): # 변수 j는 1~9의 값을 차례로 입력받아 안쪽 반복문을 실행합니다.
```

```
            print(f'{i} * {j} = {i*j}')
```

```
        print() # 안쪽 반복문이 끝나고 변수 i가 다음 값을 받기 전에 줄바꿈을 실행합니다.
```

```
>>> print('반복문 실행 완료!') # 반복문이 끝나면 왼쪽 코드를 실행합니다.
```


반복문 실행 결과로 리스트 생성

- 반복문을 실행할 범위로 리스트를 생성합니다.

```
>>> nums = list(range(1, 11)) # 정수 1~10을 원소로 갖는 리스트를 생성합니다.
```

```
>>> nums
```

- 빈 리스트를 미리 생성하고 반복문으로 생성한 값을 리스트 원소로 추가합니다.

```
>>> sqrs = [] # for 반복문을 실행하기 전에 값을 저장할 빈 리스트를 미리 생성합니다.
```

```
>>> for num in nums: # nums의 각 원소를 변수 num에 할당하고 아래 코드를 반복 실행합니다.
```

```
    sqrs.append(num ** 2) # nums을 제공한 값을 sqrs의 마지막 원소로 추가합니다.  
    [주의] sqrs를 미리 생성하지 않으면 에러가 발생합니다.
```

```
>>> sqrs # sqrs를 출력합니다. 1~10의 정수를 제공한 값을 원소로 갖습니다.
```

반복문 실행 결과로 리스트 생성(계속)

- nums에서 짝수만 제공한 값을 원소로 갖는 리스트를 생성합니다.

```
>>> sqrs = [] # sqrs를 다시 빈 리스트로 생성합니다.
```

```
>>> for num in nums:
```

```
    if num % 2 == 0: # num이 짝수일 때 아래 코드를 실행합니다.
```

```
        sqrs.append(num ** 2) # num을 제공한 값을 sqrs의 마지막 원소로 추가합니다.
```

```
>>> sqrs # sqrs를 출력합니다. 짝수를 제공한 값을 원소로 갖습니다.
```

리스트 컴프리헨션

- 리스트 컴프리헨션^{Comprehension}은 반복문을 대괄호 안에서 실행하고 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.
```

- 반복문 뒤에 조건문을 추가하면 조건을 만족하는 리스트 원소만 남길 수 있습니다.

```
>>> [num ** 2 for num in nums if num % 2 == 0] # nums에서 짝수만 남겨서 제공한 값을 원소로  
갖는 리스트를 생성합니다.
```

- 중첩 for 반복문을 리스트 컴프리헨션으로 실행합니다.

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

zip() 함수 사용법

- `zip()` 함수는 지정한 두 객체에서 같은 인덱스 원소 쌍을 튜플로 반환합니다.

```
>>> for i in zip(range(2, 10), range(1, 10)): # [참고] zip() 함수에 리스트, 튜플, 딕셔너리,
                                             집합 등 다양한 객체를 지정할 수 있습니다.
    print(i) # 변수 i는 두 객체에서 같은 인덱스 원소 쌍을 튜플로 받습니다.
              [참고] zip() 함수에 딕셔너리를 지정하면 키를 출력합니다.
              # 두 객체의 길이(원소 개수)가 다르면 원소 개수가 적은 객체를 기준으로 동작합니다.
```

- 두 객체에서 같은 인덱스 원소를 변수 `i`와 `j`로 받아서 반복문을 실행합니다.

```
>>> for i, j in zip(range(2, 10), range(1, 10)): # [참고] 튜플 원소를 분리하여 개별 변수로
                                                  받도록 튜플을 언패킹unpacking 합니다.
    print(f'{i} * {j} = {i*j}')
```

enumerate() 함수 사용법

- `enumerate()` 함수는 지정한 객체의 인덱스와 원소 쌍을 튜플로 반환합니다.

```
>>> for i in enumerate(nums): # [참고] enumerate() 함수에 리스트, 튜플, 딕셔너리, 집합 등 다양한  
                               객체를 지정할 수 있습니다.  
    print(i) # 변수 i는 nums의 인덱스와 원소 쌍을 튜플로 받습니다.  
             [참고] enumerate() 함수에 딕셔너리를 지정하면 키를 출력합니다.
```

- 객체의 인덱스와 원소를 변수 `i`와 `v`로 받아서 반복문을 실행합니다.

```
>>> for i, v in enumerate(nums):  
    print(f'{i}번 인덱스 원소는 {v}입니다.')
```

반복문에서 에러 발생

- 반복문 실행 도중 에러가 발생하면 반복문을 중단합니다.

```
>>> for num in nums:
```

```
    if num % 3 == 0:
```

```
        num = str(num) # 변수 num이 3의 배수면 문자열로 변환합니다.
```

```
    print(num ** 2) # 변수 num이 3의 배수면 에러가 발생하고 for 반복문을 중단합니다.  
                    [참고] 문자열을 제공할 수 없으므로 TypeError가 발생합니다.
```

예외 처리

- 코드 실행 중 에러가 발생할 때 다른 코드를 실행하려면 예외 처리를 추가합니다.

```
>>> try: # try문 아래에 여러 줄의 실행 코드를 추가합니다.
```

코드A

except 에러 종류: # 코드A 실행 도중 에러가 발생하면 대신 실행할 코드를 except문에 추가합니다.
[참고] except문에 에러 종류를 명시하면 해당 에러에 대해 아래 코드를 실행합니다.

코드B # [참고] 코드B에 **pass**를 지정하면 아무것도 실행하지 않고 통과합니다.

finally: # 코드 에러와 상관 없이 항상 마지막으로 실행할 코드가 있으면 finally문에 추가합니다.
[참고] finally문은 생략할 수 있습니다.

코드C

반복문에 예외 처리 추가

- 반복문에 예외 처리를 추가하면 도중에 에러가 발생해도 끝까지 실행합니다.

```
>>> for num in nums:
```

```
    if num % 3 == 0:
```

```
        num = str(num)
```

```
    try:
```

```
        print(num ** 2) # 변수 num이 3의 배수면 문자열로 변환하므로 에러가 발생합니다.
```

```
    except Exception as e: # 에러가 발생하면 변수 e에 에러 메시지를 할당합니다.
```

```
        print(e) # 에러가 발생하면 에러 메시지를 출력합니다.
```


while 반복문 기본 구조

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

```
>>> while 조건: # 조건이 True면 아래 코드를 반복 실행합니다.
```

```
    코드A
```

```
    코드B
```

```
    코드C
```

왼쪽 코드에 while 반복문 조건을 False로 만드는 증감식 또는 조건문 + break를 추가합니다.
그렇지 않으면 while 반복문을 무한 반복합니다!

while 반복문 실습

- for 반복문에서 사용했던 변수 `i`를 출력합니다.

```
>>> print(i) # 현재 i는 nums의 마지막 인덱스인 정수 9를 가리키고 있습니다.
```

- while 반복문을 실행하여 변수 `i`에서 1씩 차감한 값을 출력합니다.

```
>>> while i > 0: # 지정한 조건이 False가 될 때까지 아래 코드를 반복 실행합니다.
```

```
    print(i)
```

```
    i -= 1 # 왼쪽 코드는 반복문을 실행하면서 i에서 1씩 차감하는 증감식입니다.  
           [주의] 뺄셈 대신 덧셈 연산자를 사용하면 while 반복문을 무한 실행합니다.
```

while 반복문 실습(계속)

- while 반복문에 조건문을 추가하고 break를 만나면 반복문을 중단합니다.

```
>>> while True: # while 반복문에 조건 대신 True를 지정하면 반복문을 무한 반복합니다.  
                코드 실행을 멈추려면 상단 메뉴에서 네모 버튼(interrupt)을 클릭합니다!
```

```
    i += 1
```



```
    if i >= 10000:
```

```
        break
```

- 변수 i를 출력합니다.

```
>>> print(i)
```

사용자 정의 함수

사용자 정의 함수의 필요성

- 당장 실행되는 코딩에 집중하면 같은 코드를 여러 번 중복할 가능성이 있습니다.
 - 예를 들어 체질량지수^{BMI} 계산 코드를 10번 복사하여 값만 바꿔서 코딩했다고 가정합시다.
- 코드를 중복 사용하면 코드가 길어지고 사소한 수정 작업도 어려울 수 있습니다.
 - 중복 코드만 50줄입니다.
 - hgt를 height로 바꾸려면 30번 수정해야 합니다.
- 중복 코드를 함수로 만들면 쉽게 해결할 수 있습니다.
 - 중복 코드를 한 줄 함수로 변경하면 코드가 짧아집니다.
 - 수정사항이 발생하면 함수 코드만 한 번 수정합니다.

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

[참고] 스파게티 코드

- 프로그래밍 결과 코드가 복잡하게 엉켜 있는 상태를 스파게티 코드라고 합니다.
- 스파게티 코드는 정상적으로 동작하지만 코드를 읽고 동작을 파악하기 어렵다는 특징이 있습니다.
- 만약 코드 실행 도중에 에러가 발생하면 에러를 빠르게 수정하기 어렵습니다.
- 따라서 가독성 높은 코딩을 위해 중복을 피하고 주석을 추가하는 것이 좋습니다.



사용자 정의 함수 기본 구조

- 체질량지수를 반환하는 사용자 정의 함수를 생성합니다.

```
>>> def BMI(height, weight): # 함수명과 매개변수를 정의합니다.
```

```
    bmi = weight / (height/100) ** 2
```

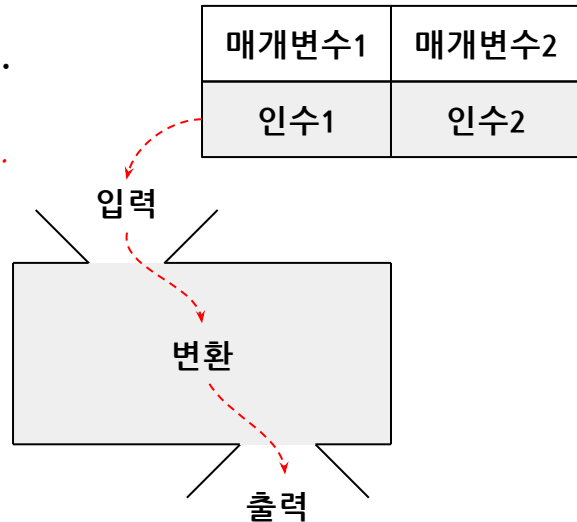
```
    return bmi # 함수가 반환할 값을 설정합니다.
```

- 사용자 정의 함수를 실행합니다.

```
>>> BMI(175, 65) # 매개변수 없이 인수만 지정하여 실행합니다.
```

```
>>> BMI(height = 175, weight = 65) # 매개변수와 인수를 등호로 연결하여 실행합니다.
```

```
>>> BMI(weight = 65, height = 175) # 매개변수를 추가하면 인수의 순서를 변경할 수 있습니다.  
[참고] height는 매개변수parameter, 175는 인수argument입니다.
```



인수의 기본값 설정 및 독스트링 추가

- 사용자 정의 함수에서 인수의 기본값을 설정하고 독스트링을 추가합니다.

```
>>> def BMI(height = 175, weight = 65):
```

```
    ...
```

```
    This function returns BMI from height(cm) and weight(kg).
```

```
    ''' # [참고] 함수 관련 설명을 Docstring으로 추가할 수 있습니다.
```

```
    return weight / (height/100) ** 2 # 중간에 변수를 생성하지 않고 반환할 수 있습니다.
```

- 사용자 정의 함수를 실행합니다.

```
>>> BMI() # 사용자 정의 함수를 실행할 때 인수를 생략하면 기본값을 적용합니다.
```


람다 표현식 사용법

- 람다^{lambda} 표현식은 한 줄 코드로 함수를 만들 때 사용합니다.

- 아래는 BMI 함수를 람다 표현식으로 바꾼 코드입니다.
- 람다 표현식에서도 인수의 기본값을 설정할 수 있습니다.

```
>>> BMI2 = lambda height, weight: weight / (height/100) ** 2
```

콜론 뒤에 오는 코드를 실행하고 반환합니다.

- 람다 표현식으로 정의한 함수를 실행합니다.

```
>>> BMI2(height = 190, weight = 85)
```

- 람다 표현식을 괄호로 감싸면 함수처럼 실행할 수 있으므로 **익명함수**라 합니다.

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

사용자 정의 함수 주의사항

- 함수를 정의할 때 입력받을 매개변수를 나열합니다.

```
>>> def printValues1(a, b, c):  
    for i in (a, b, c):  
        print(i)
```

- 함수에서 정의한 매개변수에 인수를 누락하면 에러가 발생합니다.

```
>>> printValues1(1, 2, 3) # 함수에서 정의한 매개변수 개수만큼 인수를 지정합니다.
```

```
>>> printValues1(1, 2) # 인수를 누락하면 에러가 발생합니다.
```

여러 인수를 튜플로 받는 함수

- 함수를 정의할 때 매개변수 대신 ***args**를 지정합니다.

```
>>> def printValues2(*args):  
    for i in args: # [참고] args는 튜플로 전달받습니다.  
        print(i)
```

- 함수의 괄호 안에 원하는 개수만큼 인수를 지정할 수 있습니다.

```
>>> printValues2(1, 2, 3)  
  
>>> printValues2(1, 2) # 인수를 2개 지정해도 에러가 발생하지 않습니다.
```

여러 인수를 딕셔너리로 받는 함수

- 함수를 정의할 때 매개변수 대신 ****kwargs**를 지정합니다.

```
>>> def printValues3(**kwargs):
```

```
    for i, v in kwargs.items(): # [참고] kwargs는 딕셔너리로 전달받는데 딕셔너리의 items()  
                                함수는 키와 값의 쌍을 튜플로 묶어서 리스트로 반환합니다.
```

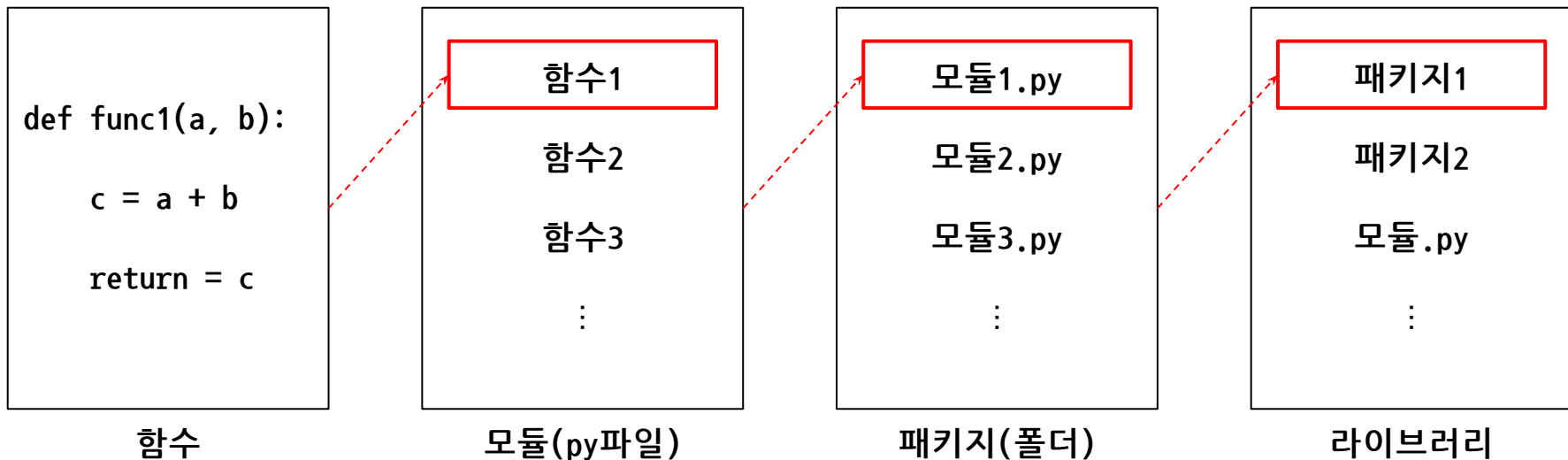
```
        print(f'{i}은(는) {v}입니다.')
```

- 함수의 괄호 안에 **dict()** 클래스 함수처럼 키와 값을 등호로 연결하여 지정합니다.

```
>>> printValues3(메뉴 = '순대국')
```

```
>>> printValues3(메뉴 = '순대국', 가격 = 9000) # 키와 값의 쌍을 여러 개 지정할 수 있습니다.
```

함수, 모듈, 패키지, 라이브러리의 관계



사용자 정의 함수 모듈 생성

- 사용자 정의 함수 코드를 모듈(py 파일)로 저장하면 필요할 때마다 호출할 수 있으므로 편리합니다.
- Anaconda 메인에서 Text File을 열고, 모듈(py 파일)로 저장할 사용자 정의 함수 (예를 들어 printValues3) 코드를 붙여넣습니다.
- 상단 메뉴에서 File → Save Text As를 클릭하고 텍스트 파일을 저장합니다.
 - 파일명을 myFuncs.py로 저장합니다. # [주의] py 파일을 Jupyter Notebook 파일과 같은 폴더에 저장해야 호출할 수 있습니다.
- Jupyter Notebook으로 돌아와 메뉴에서 Kernel → Restart Kernel and Clear All Outputs를 클릭하고 Jupyter Notebook을 초기화합니다.

사용자 정의 함수 모듈 호출

- 사용자 정의 함수 모듈을 호출하는 세 가지 방법을 소개합니다.

```
>>> import myFuncs # myFuncs 모듈을 호출합니다.  
# 함수는 myFuncs 모듈에 속하므로 함수명 앞에 myFuncs를 추가해야 합니다.
```

```
>>> myFuncs.printValues3(메뉴 = '순대국', 가격 = 9000) # Jupyter Notebook을 초기화하고  
# 두 번째 방법을 실행합니다.
```

```
>>> import myFuncs as mf # myFuncs 모듈을 mf라는 가명으로 호출합니다.  
# 함수는 mf 모듈에 속하므로 함수명 앞에 mf를 추가해야 합니다.
```

```
>>> mf.printValues3(메뉴 = '순대국', 가격 = 9000) # Jupyter Notebook을 초기화하고 세 번째  
# 방법을 실행합니다.
```

```
>>> from myFuncs import printValues3 # myFuncs 모듈에서 일부 함수만 호출합니다.  
# 함수명 앞에 모듈명을 추가할 필요가 없습니다.
```

```
>>> printValues3(메뉴 = '순대국', 가격 = 9000)
```

[참고] 다양한 괄호 사용법

- 소괄호, 중괄호, 대괄호 사용법에 대해 표로 정리한 것입니다.

구분	상세내용
소괄호 ()	<ul style="list-style-type: none"> - 튜플을 생성할 때 소괄호 안에 원소를 콤마로 연결합니다. - 함수 또는 객체의 방식^{method} 뒤에 추가해야 합니다.
중괄호 { }	<ul style="list-style-type: none"> - 딕셔너리 또는 집합을 생성할 때 중괄호 안에 원소를 콤마로 연결합니다. - f-문자열에서 변수를 중괄호로 감싸주어야 합니다.
대괄호 []	<ul style="list-style-type: none"> - 리스트를 생성할 때 대괄호 안에 원소를 콤마로 연결합니다. - 객체를 인덱싱할 때 대괄호 안에 정수 스칼라, 슬라이스를 지정합니다.

numpy 라이브러리

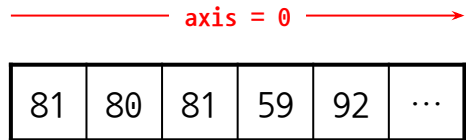
numpy 라이브러리

- numpy 라이브러리 함수로 다음과 같은 작업을 수행할 수 있습니다.
 - 1~n차원의 배열^{ndarray}을 생성합니다.
 - 배열의 차원^{dimension}을 확인하고 형태^{shape}를 변경합니다.
 - 1차원 배열(벡터) 및 2차원 배열(행렬) 연산을 실행합니다.[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.

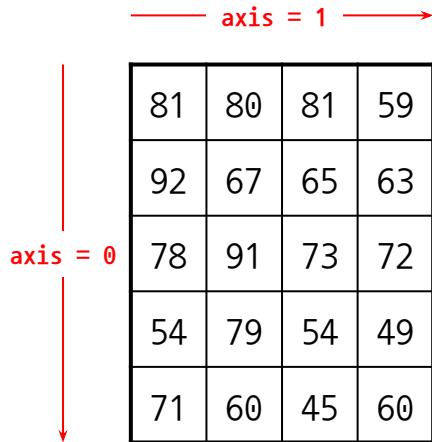
```
>>> import numpy as np # numpy 라이브러리를 호출하고 np로 사용하도록 설정합니다.
```

[참고] 배열의 시각적 예시

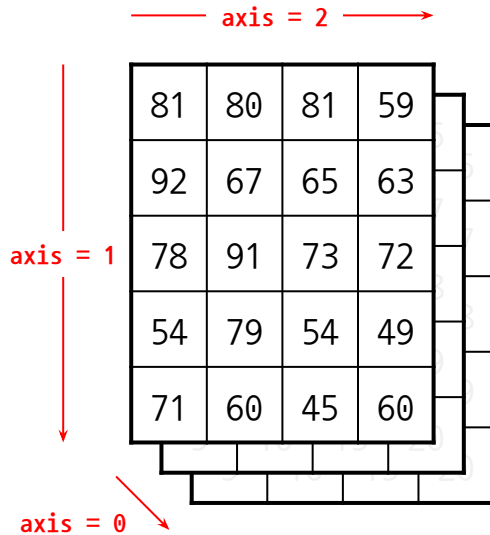
[1차원 배열]



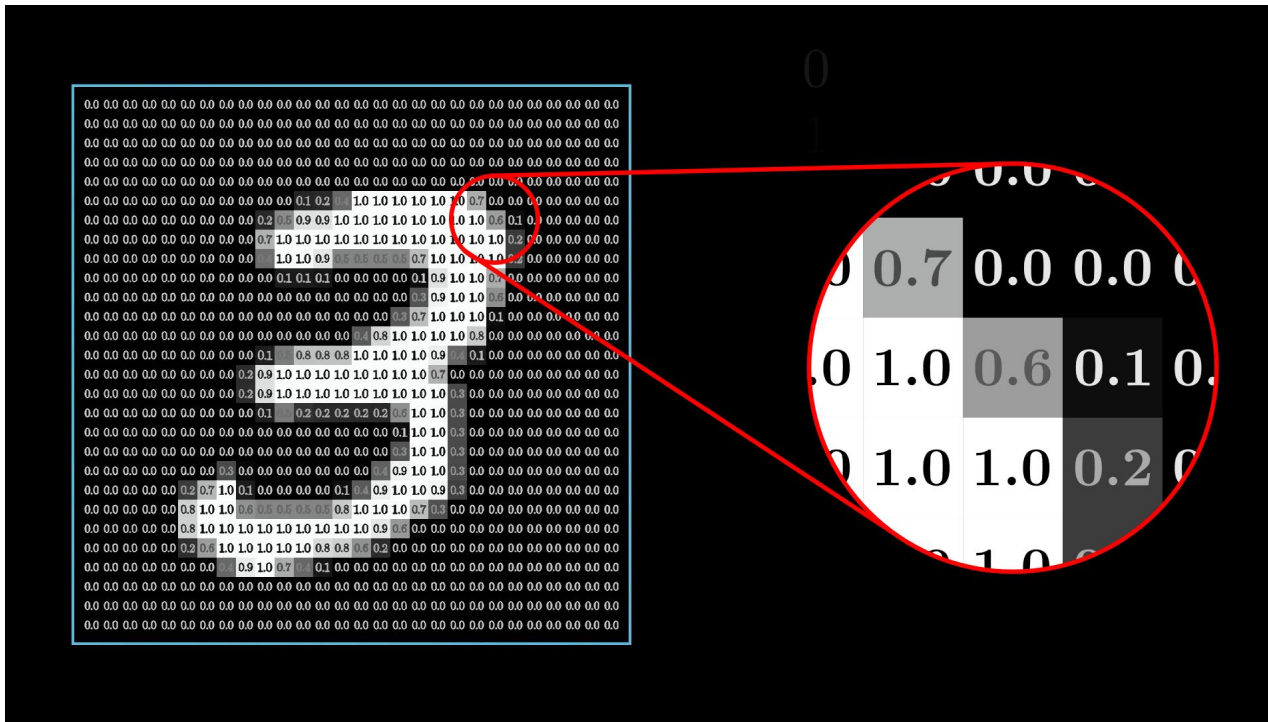
[2차원 배열]



[3차원 배열]

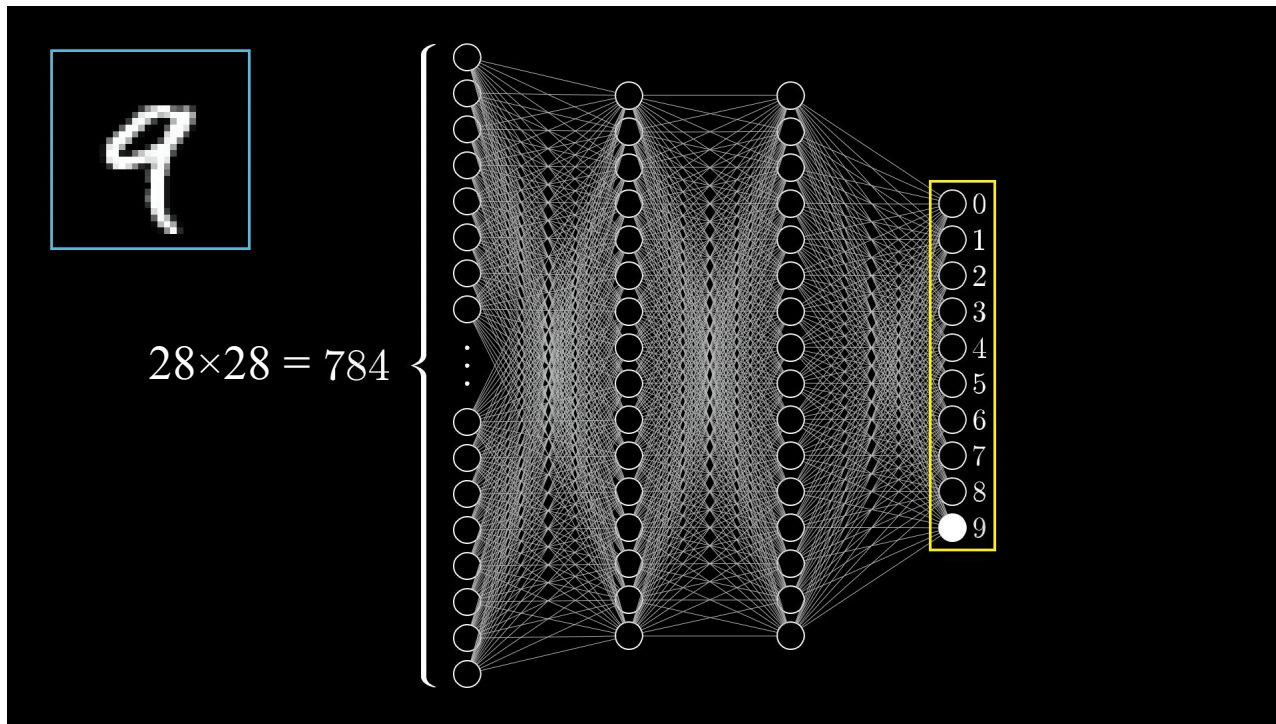


[참고] 흑백 이미지를 행렬로 표현



출처: <https://3b1b-posts.us-east-1.linodeobjects.com//content/lessons/2017/neural-networks/pixel-values.png>

[참고] 딥러닝을 활용한 손글씨 분류



출처: <https://3b1b-posts.us-east-1.linodeobjects.com//content/lessons/2017/neural-networks/output-layer.png>

1차원 배열 생성

- 1차원 배열을 생성하고 클래스를 확인합니다.

```
>>> ar1 = np.array(object = [1, 2, 3]) # 리스트로 1차원 배열을 생성합니다.
```

```
>>> ar1 # ar1을 출력합니다. 원소를 가로 방향으로 출력합니다.
```

```
>>> type(ar1) # ar1의 클래스를 확인합니다. ar1의 클래스는 numpy.ndarray입니다.
```

- 1차원 배열의 형태, 원소 개수 및 원소 자료형을 확인합니다.

```
>>> ar1.shape # ar1의 형태(행 개수, 열 개수)를 확인합니다.
```

```
>>> ar1.size # ar1의 원소 개수를 확인합니다.
```

```
>>> ar1.dtype # ar1의 원소 자료형을 확인합니다. ar1의 원소 자료형은 numpy.int64입니다.  
[참고] Windows에서는 정수를 numpy.int32로 생성합니다.
```

1차원 배열의 원소 자료형 변환

- 배열은 모든 원소 자료형이 같아지도록 자료형을 자동 변환합니다.

```
>>> a = [1, 2.0, '3'] # 다양한 자료형을 원소로 갖는 리스트를 생성합니다.
```

```
>>> ar1 = np.array(object = a) # 리스트로 배열을 생성하면 원소 자료형을 자동 변환합니다.  
[참고] 정수 < 실수 < 문자열 방향으로 변환합니다.
```

```
>>> ar1 # ar1을 출력합니다. ar1의 원소 자료형은 'U32'입니다.  
[참고] 'U32'는 little-endian 32 character string이며, 'U'는 Unicode string입니다.
```

- 배열의 원소 자료형을 변환할 때 **astype()** 함수를 사용합니다.

```
>>> ar1.astype(float) # ar1의 원소 자료형을 실수로 변환합니다.
```

```
>>> ar1.astype(int) # ar1의 원소 자료형을 정수로 변환하려고 하면 '2.0' 때문에 에러가 발생합니다.
```

```
>>> ar1.astype(float).astype(int) # ar1의 원소 자료형을 실수로 변환하면 여전히 배열이므로 이어서  
정수로 변환할 수 있습니다.
```

[참고] 배열을 생성할 때 원소 자료형 지정

- 배열을 생성할 때 원소 자료형을 지정할 수 있습니다.

```
>>> np.array(object = a, dtype = float) # 배열의 원소 자료형을 실수로 지정합니다.
```

```
>>> np.array(object = a, dtype = int) # 배열의 원소 자료형을 정수로 지정합니다.
```

```
>>> np.array(object = a, dtype = str) # 배열의 원소 자료형을 문자열로 지정합니다.
```

```
>>> np.array(object = a, dtype = object) # 배열의 원소 자료형을 객체로 지정합니다.  
# 리스트의 원소 자료형을 그대로 유지합니다.
```

- [참고] **pandas** 라이브러리에서 1차원 자료구조인 시리즈^{Series}를 생성할 때 원소에 문자열을 포함하고 있으면 시리즈의 원소 자료형을 object로 자동 변환합니다.
 - 따라서 시리즈는 원소 자료형이 섞여 있을 수 있으므로 주의해야 합니다.

간격이 일정한 배열 생성

- 간격이 일정한 실수를 원소로 갖는 1차원 배열을 생성합니다.

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

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

```
>>> np.arange(start = 1, stop = 11, step = 2) # 1부터 10까지 홀수인 정수를 반환합니다.
```

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

- 지정한 개수만큼 원소를 갖는 1차원 배열을 생성합니다.

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

```
>>> np.linspace(start = 80, stop = 75, num = 31) # 80부터 75까지 원소 개수가 31개인 실수를 반환합니다.
```

원소를 반복한 배열 생성

- 배열의 전체 원소를 두 번 반복합니다.

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

- 배열의 각 원소를 두 번씩 반복합니다.

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

- 배열 원소별로 반복할 횟수를 지정합니다.

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

```
>>> np.repeat(a = ar1, repeats = range(3, 0, -1)) # 반복 횟수를 range() 함수로 지정할 수  
있습니다.
```

1차원 배열 인덱싱 및 슬라이싱

- 배열 인덱싱은 인덱스(위치번호)로 원소를 선택합니다.

인덱스	0	1	2	3	4	5
원소	1	3	5	7	9	11

- 배열에 대괄호를 추가하고 대괄호 안에 선택할 원소의 인덱스를 지정합니다.

```
>>> ar1 = np.arange(start = 1, stop = 12, step = 2) # 1부터 11까지 홀수인 정수를 원소로 갖는 1차원 배열을 생성합니다.
```

```
>>> ar1[0] # ar1의 0번 인덱스(첫 번째) 원소를 선택합니다.
```

```
>>> ar1[1] # ar1의 1번 인덱스(두 번째) 원소를 선택합니다.
```

```
>>> ar1[-1] # ar1의 -1번 인덱스(마지막) 원소를 선택합니다.
```

1차원 배열 인덱싱 및 슬라이싱(계속)

- 배열 슬라이싱은 슬라이스를 지정하여 연속한 원소를 선택합니다.

```
>>> ar1[:3] # ar1의 0~2번 인덱스 원소를 선택합니다.
```

```
>>> ar1[3:] # ar1의 3번 인덱스 원소부터 마지막 원소까지 선택합니다.
```

- 대괄호 안에 정수를 원소로 갖는 리스트를 지정하면 해당 원소를 선택합니다.

```
>>> ar1[[3, 2, 1]] # 정수 스칼라 또는 슬라이스 대신 리스트를 지정하는 배열 인덱싱이 가능합니다.  
[주의] 리스트는 배열 인덱싱을 실행할 수 없습니다!
```

```
>>> ar1[[3, 3, 3]] # 리스트 원소를 반복하면 같은 원소를 여러 번 선택합니다.
```

2차원 배열 생성

- 같은 길이의 리스트를 원소로 갖는 리스트로 2차원 배열을 생성합니다.

```
>>> ar2 = np.array(object = [[1, 2, 3], [4, 5, 6]])
```

```
>>> ar2 # ar2를 출력합니다. 전체 원소를 2행 3열로 배치합니다.
```

```
>>> type(ar2) # ar2의 클래스를 확인합니다. ar2의 클래스는 numpy.ndarray입니다.
```

- 2차원 배열의 형태, 원소 개수 및 원소 자료형을 확인합니다.

```
>>> ar2.shape # ar2의 형태(행 개수, 열 개수)를 확인합니다.
```

```
>>> ar2.size # ar2의 원소 개수를 확인합니다.
```

```
>>> ar2.dtype # ar2의 원소 자료형을 확인합니다. ar2의 원소 자료형은 numpy.int64입니다.
```

배열의 재구조화

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

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

```
>>> ar1.reshape(4, 3) # ar1을 4행 3열인 2차원 배열로 변환합니다. 원소 입력 방향은 가로입니다.  
[주의] 1차원 배열 원소 개수의 약수만 행 또는 열 개수로 설정할 수 있습니다.
```

```
>>> ar1.reshape(4, 3, order = 'F') # order = 'F'를 추가하면 원소 입력 방향을 세로로 변경합니다.  
[참고] order 매개변수에 전달하는 인수의 기본값은 'C'입니다.
```

- 2차원 배열 인덱싱 및 슬라이싱 실습을 위해 `ar2`를 생성합니다.

```
>>> ar2 = ar1.reshape(4, -1) # ar1을 4행 3열인 2차원 배열로 변환하고 ar2에 할당합니다.  
[참고] 열 위치에 -1을 지정하면 열 개수를 자동 계산합니다.
```

```
>>> ar2 # ar2는 4행 3열인 2차원 배열입니다.
```

```
>>> ar2.flatten() # 2차원 배열을 1차원 배열로 변환합니다.  
[참고] order 매개변수를 추가할 수 있습니다.(기본값: 'C')
```

[참고] 배열의 결합

- 원소 개수가 같은 1차원 배열을 가로 방향으로 쌓은 2차원 배열을 생성합니다.

```
>>> np.column_stack(tup = (ar1, ar1)) # [주의] 함수의 괄호 안에 1차원 배열을 튜플 또는 리스트로  
# 지정해야 합니다!
```

- 원소 개수가 같은 1차원 배열을 세로 방향으로 쌓은 2차원 배열을 생성합니다.

```
>>> np.row_stack(tup = (ar1, ar1))
```

- 두 개 이상의 1차원 배열을 일렬로 결합하여 커다란 1차원 배열을 생성합니다.

```
>>> np.concatenate((ar1, ar1)) # [참고] 리스트를 + 연산자로 결합하는 것과 같은 동작입니다.
```

2차원 배열 인덱싱 및 슬라이싱

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

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

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

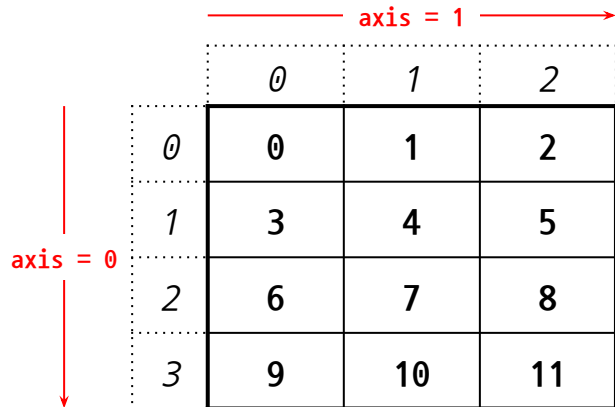
- 2차원 배열의 슬라이싱을 실행합니다.

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

```
>>> ar2[:, 1:3] # ar2의 전체 행 2~3열 원소를 선택합니다.  
[참고] 전체 행을 선택하려면 빈 콜론을 추가합니다.
```

- 리스트를 사용한 배열 인덱싱도 가능합니다.

```
>>> ar2[:, [2, 1]] # ar2의 일부 열 순서를 변경합니다.  
[주의] 행과 열 순서를 함께 변경할 수 없습니다.
```



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

배열 산술 연산

- 원소 개수가 3인 1차원 배열과 3행 1열의 2차원 배열을 생성합니다.

```
>>> ar1 = np.array(object = range(3)); ar1
```

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

- 배열의 산술 연산을 실행할 때 차원이 같아지도록 브로드캐스팅^{broadcasting}합니다.

```
>>> ar1 + 1 # ar1의 각 원소에 1을 더합니다.
```

```
>>> ar2 * 2 # ar2의 각 원소에 2를 곱합니다.
```

```
>>> ar1 + ar2 # ar1과 ar2를 더합니다.
```

[참고] 배열 브로드캐스팅

- 두 배열의 원소 개수가 다르면 원소 개수가 작은 배열을 확장합니다.

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} \quad \# \text{ [참고] 원소 개수가 같거나 1이어야 합니다!}$$

$(3,)$
 $(1,)$
 $(3,)$

$$\begin{array}{|c|} \hline 0 \\ \hline 1 \\ \hline 2 \\ \hline \end{array} \times \begin{array}{|c|} \hline 2 \\ \hline 2 \\ \hline 2 \\ \hline \end{array} = \begin{array}{|c|} \hline 0 \\ \hline 2 \\ \hline 4 \\ \hline \end{array}$$

$(3,1)$
 $(1,1)$
 $(3,1)$

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline 2 & 2 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 1 & 2 & 3 \\ \hline 2 & 3 & 4 \\ \hline \end{array}$$

$(1,3)$
 $(3,1)$
 $(3,3)$

[참고] 두 배열의 차원을 열 → 행 순서로 확인합니다.

[참고] 수학 관련 값과 함수

- 데이터 분석 과정에서 자주 사용하는 수학 관련 값과 함수를 정리한 표입니다.

구분	상세 내용	구분	상세 내용	구분	상세 내용
np.e	자연상수(2.7182)	np.sqrt()	제곱근	np.min()	최솟값
np.pi	파이(3.141592)	np.power()	거듭제곱	np.max()	최댓값
np.nan	결측값	np.exp()	자연상수 거듭제곱	np.mean()	평균
np.inf	무한대	np.log()	자연로그	np.median()	중위수
np.floor()	소수점 버림	np.sum()	모든 원소 덧셈	np.var()	분산
np.ceil()	소수점 올림	np.prod()	모든 원소 곱셈	np.std()	표준편차
np.round()	반올림	np.diff()	원소 차이	np.size()	원소 개수

pandas 라이브러리

pandas 라이브러리

- pandas 라이브러리의 함수로 다음과 같은 작업을 실행합니다.
 - 1차원 시리즈^{Series}: 원소를 세로 방향으로 정렬한 1차원 자료구조이며, 데이터프레임의 원소(열벡터)입니다.
 - 2차원 데이터프레임^{DataFrame}: 1차원 시리즈를 원소로 갖는 2차원 자료구조입니다.
 - R의 데이터프레임에서 유래한 것으로 알려져 있습니다.
 - 3차원 패널^{Panel}: 시간의 흐름에 따라 여러 데이터프레임을 중첩한 것입니다.
 - 동일 관찰 대상이 시간의 흐름에 따라 어떻게 바뀌는지 확인하는 분석이 가능합니다.
- pandas 라이브러리를 호출합니다.

```
>>> import pandas as pd
```

pandas 라이브러리를 호출하고 pd로 사용하도록 설정합니다.
[참고] numpy 라이브러리도 함께 호출하는 것이 좋습니다.

데이터프레임의 시각적 예시

- 데이터프레임은 행^{row}과 열^{column}을 갖는 2차원 자료구조입니다.
 - 열별 자료형이 다를 수 있으며, 행 또는 열을 하나만 선택하면 시리즈로 반환합니다.
 - 2차원 배열은 전체 원소 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] 데이터 척도의 종류

범주형

데이터

명목척도 nominal scale

- 이름으로 구분하는 것에 의미 부여
 - 국가명, 학교명, 혈액형 등
- 수치가 아니므로 사칙연산 불가
- 등호, 빈도수와 백분율 계산

서열척도 ordinal scale

- 명목척도에 높낮이(서열) 부여
 - 학력, 등급, 직급 등
- 수치가 아니므로 사칙연산 불가
- 부등호/등호, 빈도수와 백분율 계산

연속형

데이터

등간척도 interval scale

- 높낮이에 등간격으로 수치 부여
- 절대 영점 없음(ex. 0℃)
 - 온도, 리커트 5점 척도 등
- 가감연산(+, -)과 평균 계산

비율척도 ratio scale

- 등간척도에 0과 비율 부여
- 절대 영점 있음(ex. 0km)
 - 길이, 무게, 점수 등
- 사칙연산(+, -, ×, ÷)과 평균 계산

[참고] 데이터 척도의 특징

척도	동일 여부	크기 비교	차이 계산	비율 관계
명목척도	○	×	×	×
서열척도	○	○	×	×
등간척도	○	○	○	×
비율척도	○	○	○	○

[참고] 데이터프레임의 명칭

열

Column

{

변수 Variable

특성 Feature

속성 Attribute

열 크기: p

	X ₁	X ₂	...	X _p
1				
2				
3				
⋮				
n				

행

Row

{

관측값 Observation

레코드 Record

사례 Case, Instance

행 크기: n

셀

Cell

셀 값 value 을 컬럼값, 특성값 또는 속성값이라고 합니다.

시리즈 생성

- 시리즈는 1차원 배열, 리스트 또는 딕셔너리로 생성하고 인덱스를 갖습니다.
- 1차원 배열로 시리즈를 생성하고 클래스를 확인합니다.

```
>>> ar1 = np.arange(start = 1, stop = 6, step = 2)
```

```
>>> ar1 # ar1을 출력합니다. 원소를 가로 방향으로 출력합니다.
```

```
>>> sr = pd.Series(data = ar1) # 1차원 배열로 시리즈를 생성합니다.
```

```
>>> sr # sr을 출력합니다. 시리즈는 인덱스index와 값value을 세로로 출력하고 맨 아래에 원소 자료형dtype을  
[참고] 시리즈를 생성할 때 인덱스를 지정하지 않으면 정수 0부터 시작하는 인덱스를 자동으로 적용합니다.
```

```
>>> type(sr) # sr의 클래스를 확인합니다. sr의 클래스는 pandas.core.series.Series입니다.
```

[참고] 시리즈를 생성하는 다른 방법

- 다양한 자료형을 원소로 갖는 리스트로 시리즈를 생성합니다. # 시리즈를 생성할 때 인덱스를 지정할 수 있습니다.

```
>>> sr1 = pd.Series(data = [1, 2.0, '3'], index = ['a', 'b', 'c'])
```

```
>>> sr1 # sr1을 출력합니다. sr1의 원소 자료형은 object이며 원소별 자료형이 다를 수 있습니다.  
[참고] 문자열을 따옴표 없이 출력합니다.
```

```
>>> for i in sr1: # 반복문으로 sr1의 원소별 클래스를 출력합니다.
```

```
    print(type(i)) # 원소별 자료형이 제각각입니다.
```

- 딕셔너리로 시리즈를 생성합니다.

```
>>> pd.Series(data = {'a': 1, 'b': 3, 'c': 5}) # [주의] 딕셔너리의 키를 인덱스로 자동 적용  
하므로 다른 인덱스를 지정할 수 없습니다!
```

시리즈 확인

- 시리즈의 형태, 원소 개수 및 원소 자료형을 확인합니다.

```
>>> sr.shape # sr의 형태(행 개수, 열 개수)를 확인합니다.
```

```
>>> sr.size # sr의 원소 개수를 확인합니다.
```

```
>>> sr.dtype # sr의 원소 자료형을 확인합니다. sr의 원소 자료형은 numpy.int64입니다.  
[참고] Windows에서는 정수를 numpy.int32로 생성합니다.
```

- 시리즈의 값(원소)과 인덱스를 확인합니다.

```
>>> sr.values # sr의 값(원소)을 확인합니다. sr의 값은 numpy.ndarray입니다.
```

```
>>> sr.index # sr의 인덱스를 확인합니다.  
[참고] 시리즈를 생성할 때 인덱스를 지정하지 않으면 정수 0부터 시작합니다.
```

시리즈 인덱스 변경

- 시리즈 인덱스의 클래스를 확인합니다.

```
>>> type(sr.index) # sr 인덱스의 클래스는 pandas.core.indexes.range.RangeIndex입니다.
```

- 시리즈 인덱스를 정수로 변경합니다.

```
>>> sr.index = [1, 2, 3]; sr # [참고] 시리즈 index 속성에 원소 개수와 같은 길이의 리스트를 할당하면  
# 리스트 원소로 인덱스를 변경합니다. 아울러 중복 인덱스를 허용합니다.
```

```
>>> type(sr.index) # sr 인덱스의 클래스는 pandas.core.indexes.numeric.Int64Index입니다.
```

- 시리즈 인덱스를 문자열로 변경합니다.

```
>>> sr.index = ['a', 'b', 'c']; sr # [참고] 시리즈 인덱스에 다양한 자료형을 원소로 갖는 리스트를  
# 할당할 수 있습니다. 예를 들어 [1, 1.0, '1']도 가능합니다.
```

```
>>> type(sr.index) # sr 인덱스의 클래스는 pandas.core.indexes.base.Index입니다.
```

[참고] 시리즈 인덱스의 이해

- 시리즈 인덱스는 다음과 같은 특징을 갖습니다.
 - 시리즈를 생성할 때 인덱스를 지정하지 않으면 정수 0부터 시작합니다.
 - 시리즈 인덱스를 정수, 실수 또는 문자열로 변경할 수 있습니다.
 - 시리즈 인덱스는 중복을 허용합니다.
 - 시리즈에서 일부 원소를 선택하면 기존 인덱스를 유지합니다.
- 시리즈 인덱스는 정수 대신 행이름으로 인식하는 것이 좋습니다.
 - 시리즈를 인덱싱하는 방법은 기존 자료구조(예를 들어 리스트 또는 배열)에서 사용했던 정수 인덱스를 활용하는 방식과 행이름을 지정하는 방식이 있습니다.

시리즈 인덱싱 및 슬라이싱: iloc 인덱서

- iloc^{integer location} 인덱서는 정수 인덱스로 원소를 선택합니다.

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

- 리스트 또는 배열로 원소를 선택하는 방식을 **배열 인덱싱** Array Indexing이라고 합니다.

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

```
>>> sr.iloc[[0, 2]] # 연속하지 않는 인덱스를 함께 선택하려면 반드시 리스트로 지정해야 합니다.  
# 왼쪽 코드를 실행하면 sr의 0, 2번 인덱스 원소를 시리즈로 반환합니다.
```

- 정수 인덱스로 슬라이싱하면 연속한 원소를 시리즈로 반환합니다.

```
>>> sr.iloc[0:2] # sr의 0~1번 인덱스 원소를 시리즈로 반환합니다.  
[참고] iloc 인덱서는 끝(콜론 오른쪽) 인덱스를 포함하지 않습니다.
```

시리즈 인덱싱 및 슬라이싱: loc 인덱서

- `loclocation` 인덱서는 행이름으로 원소를 선택합니다.

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

- 행이름을 리스트로 지정하면 해당 원소를 선택하여 시리즈로 반환합니다.

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

```
>>> sr.loc[['a', 'c']] # 연속하지 않는 행이름을 함께 선택하려면 반드시 리스트로 지정해야 합니다.  
# 왼쪽 코드를 실행하면 sr의 행이름이 'a'와 'c'인 원소를 시리즈로 반환합니다.
```

- 행이름으로 슬라이싱하면 연속한 원소를 시리즈로 반환합니다.

```
>>> sr.loc['a':'c'] # sr의 행이름이 'a', 'b' 및 'c'인 원소를 시리즈로 반환합니다.  
[참고] loc 인덱서는 끝(콜론 오른쪽) 행이름을 포함합니다!
```


[참고] 인덱서를 반드시 사용해야 하나?

- 인덱스가 정수 1부터 시작하는 시리즈를 생성하고 인덱싱합니다.

```
>>> sr2 = pd.Series(data = range(3), index = range(1, 4)); sr2
```

```
>>> sr2[0] # sr2의 행이름에 정수 0이 없으므로 에러가 발생합니다.
```

```
>>> sr2.loc[1] # loc 인덱서로 sr2에서 행이름이 정수 1인 원소를 선택합니다.
```

- 중복 인덱스를 갖는 시리즈를 생성하고 인덱싱합니다.

```
>>> sr3 = pd.Series(data = range(3), index = np.tile(1, 3)); sr3
```

```
>>> sr3.loc[1] # loc 인덱서로 sr3에서 행이름이 정수 1인 원소를 선택합니다.
```

```
>>> sr3.iloc[0] # iloc 인덱서로 sr3에서 0번 인덱스(첫 번째) 원소를 선택합니다.
```

시리즈의 불리언 인덱싱

- 시리즈로 비교 연산을 실행합니다.

```
>>> sr >= 3 # 왼쪽 코드를 실행하면 원소가 True 또는 False인 시리즈를 반환합니다.
```

- 조건을 만족하는 원소를 선택하는 방식을 불리언 인덱싱^{Boolean Indexing}이라고 합니다.

```
>>> sr.loc[sr >= 3] # 대괄호 안에 비교 연산 코드를 지정하면 True에 해당하는 원소를 선택합니다.  
[주의] iloc 인덱서를 사용하면 에러가 발생합니다.
```

- 두 개 이상의 조건을 고려하려면 비트 연산자를 추가합니다.

```
>>> sr.loc[sr >= 3 & sr < 5] # [주의] 왼쪽 코드는 정수 3과 시리즈 sr의 논리곱 연산을 실행하므로  
에러가 발생합니다.
```

```
>>> sr.loc[(sr >= 3) & (sr < 5)] # [주의] 비트 연산자 양 옆 코드를 반드시 소괄호로 감싸야 합니다.
```

[참고] 비트 연산자

- 시리즈에서 불리언 인덱싱을 할 때 논리 연산자 대신 비트 연산자를 사용합니다.
 - [중요] 개별 조건을 반드시 소괄호로 감싸야 합니다.
- 비트 연산자는 정수를 2진수(비트)로 연산한 결과를 반환합니다.

연산자	상세 내용
&	<ul style="list-style-type: none"> [논리곱] 대응하는 비트가 모두 1이면 1, 아니면 0을 반환합니다.(and 연산)
	<ul style="list-style-type: none"> [논리합] 대응하는 비트가 하나라도 1이면 1, 아니면 0을 반환합니다.(or 연산)
^	<ul style="list-style-type: none"> [배타적 논리합] 대응하는 비트가 다르면 1, 같으면 0을 반환합니다.(xor 연산)
~	<ul style="list-style-type: none"> [논리부정] 비트가 1이면 0으로, 0이면 1로 반전합니다.(not 연산)

비트 연산자 사용법

- 부울형 시리즈 사이에 논리 연산자를 사용하면 에러가 발생합니다.

```
>>> sr >= 3 and sr < 5
```

```
>>> sr.iloc[0] >= 3 and sr.iloc[0] < 5
```

논리 연산자는 진리값 사이에서 정상적으로 동작합니다.
따라서 **numpy**, **pandas**에서는 사용할 수 없습니다.

- 비트 연산자로 논리곱, 논리합, 배타적 논리합 또는 논리부정 연산을 실행합니다.

```
>>> (sr >= 3) & (sr < 5)
```

대응하는 원소가 모두 True면 True, 아니면 False를 반환합니다.[논리곱]

```
>>> (sr >= 3) | (sr < 5)
```

대응하는 원소 중 하나라도 True면 True, 아니면 False를 반환합니다.[논리합]

```
>>> (sr >= 3) ^ (sr < 5)
```

대응하는 원소가 다르면 True, 같으면 False를 반환합니다.[배타적 논리합]

```
>>> ~ (sr >= 3)
```

True를 False로, False를 True로 반전합니다.[논리부정]

얕은 복사 vs 깊은 복사

- 얕은 복사한 시리즈 원소를 변경하면 기존 시리즈 원소에 반영합니다.

```
>>> sr4 = sr1; sr4 # sr1을 얕은 복사한 sr4를 생성합니다.
```

```
>>> sr4.iloc[0] = 2; sr4 # sr4의 첫 번째 원소를 2로 변경합니다.
```

```
>>> sr1 # sr1의 첫 번째 원소도 바뀌었습니다. [참고] 두 변수가 같은 객체를 가리키고 있기 때문입니다.
```

- 깊은 복사한 시리즈는 기존 시리즈와 별개이므로 변경 사항을 반영하지 않습니다.

```
>>> sr5 = sr1.copy(); sr5 # sr1을 깊은 복사한 sr5를 생성합니다.  
[참고] copy() 함수의 deep 매개변수에 전달하는 인수 기본값은 True입니다.
```

```
>>> sr5.iloc[0] = 1; sr5 # sr5의 첫 번째 원소를 1로 변경합니다.
```

```
>>> sr1 # sr1의 첫 번째 원소는 바뀌지 않았습니다. [참고] 두 변수가 가리키는 객체가 다르기 때문입니다.
```

데이터프레임 생성

- 데이터프레임은 2차원 배열, 리스트 또는 딕셔너리로 생성하며 인덱스(행이름)와 컬럼명(열이름)을 갖습니다.
- 2차원 배열로 데이터프레임을 생성하고 클래스를 확인합니다.

```
>>> ar2 = np.arange(start = 1, stop = 7).reshape(3, -1)
```

```
>>> ar2 # ar2를 출력합니다. 원소를 가로 방향으로 입력한 3행 2열의 2차원 배열을 출력합니다.
```

```
>>> df = pd.DataFrame(data = ar2) # 2차원 배열로 데이터프레임을 생성합니다.
```

```
>>> df # df를 출력합니다. 데이터프레임은 왼쪽에 인덱스(행이름)와 위쪽에 컬럼명(열이름)을 출력합니다.  
[참고] 데이터프레임을 생성할 때 인덱스와 컬럼명을 지정하지 않으면 정수 인덱스로 자동 적용합니다.
```

```
>>> type(df) # df의 클래스를 확인합니다. df의 클래스는 pandas.core.frame.DataFrame입니다.
```

[참고] 데이터프레임을 생성하는 다른 방법

- 데이터프레임을 생성할 때 인덱스와 컬럼명을 리스트로 지정할 수 있습니다.

```
>>> df = pd.DataFrame(data = ar2, index = range(1, 4), columns = ['A', 'B'])  
# 인덱스와 컬럼명을 리스트로 지정합니다.  
>>> df  
[참고] range() 함수로 지정할 수 있습니다.
```

- 딕셔너리로 데이터프레임을 생성합니다.

```
>>> pd.DataFrame(data = {'A': [1, 2], 'B': [3, 4]}) # 딕셔너리의 키를 열이름으로 적용하고  
값을 세로로 입력합니다.
```

- 딕셔너리를 원소로 갖는 리스트로 데이터프레임을 생성합니다.

```
>>> pd.DataFrame(data = [{'A': 1, 'B': 2}, # 딕셔너리의 키를 열이름으로 적용하고 값을 가로로  
입력합니다.  
{'A': 3, 'B': 4}])
```

데이터프레임 확인

- 데이터프레임의 다양한 정보를 확인합니다.

```
>>> df.shape # df의 형태(행 개수, 열 개수)를 확인합니다.
```

```
>>> df.dtypes # df의 열별 자료형을 확인합니다. [참고] 데이터프레임의 원소는 시리즈입니다.
```

```
>>> df.values # df의 셀 값을 확인합니다.
```

```
>>> df.index # df의 인덱스(행이름)를 확인합니다.
```

```
>>> df.columns # df의 컬럼명(열이름)을 확인합니다.
```

```
>>> df.info() # df의 정보를 확인합니다.(행 개수, 열 개수, 행이름, 열이름, 결측값 아닌 원소 개수 및 자료형)
```


데이터프레임 인덱싱 및 슬라이싱: iloc 인덱서

- iloc 인덱서에 정수 인덱스를 지정하여 행과 열을 선택합니다.

```
>>> df.iloc[0] # 대괄호 안에 정수 인덱스를 스칼라로 지정하면 해당 행을 시리즈로 반환합니다.  
[참고] 데이터프레임은 대괄호 안에逗를 생략하면 행을 선택합니다.
```

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

```
>>> df.iloc[0:2] # 대괄호 안에 슬라이스를 지정하면 해당 행을 데이터프레임으로 반환합니다.  
[주의] 마지막 인덱스를 포함하지 않습니다.
```

```
>>> df.iloc[0:2, :] # 대괄호 안에逗를 추가하고逗 뒤에 선택할 열의 정수 인덱스를 지정합니다.  
[참고] 전체 열을 선택하려면逗 오른쪽에 빈 콜론을 추가합니다.
```

```
>>> df.iloc[0:2, 0:2] # 행 인덱스가 0~1인 행과 열 인덱스가 0~1인 열을 데이터프레임으로 반환합니다.
```

```
>>> df.iloc[:, [1, 0]] # 행은 전체, 열은 리스트로 지정한 인덱스 순서로 데이터프레임을 반환합니다.
```

데이터프레임 인덱싱 및 슬라이싱: loc 인덱서

- loc 인덱서에 행이름과 열이름을 지정하여 행과 열을 선택합니다.

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

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

```
>>> df.loc[1:2] # 대괄호 안에 슬라이스를 지정하면 해당 행을 데이터프레임으로 반환합니다.  
[주의] 마지막 행이름을 포함합니다.
```

```
>>> df.loc[1:2, :] # 대괄호 안에 콤마를 추가하고 콤마 뒤에 선택할 열이름을 지정합니다.  
[참고] 전체 열을 선택하려면 콤마 오른쪽에 빈 콜론을 추가합니다.
```

```
>>> df.loc[1:2, 'A':'B'] # 행이름이 1~2인 행과 열이름이 A~B인 열을 데이터프레임으로 반환합니다.
```

```
>>> df.loc[:, ['B', 'A']] # 행은 전체, 열은 리스트로 지정한 열이름 순서로 데이터프레임을 반환합니다.
```

[참고] 인덱싱 방법에 따른 결과 비교

- 리스트, 배열, 시리즈 또는 데이터프레임은 대괄호 안에 입력하는 값에 따라 반환되는 클래스가 달라지므로 아래 내용을 반드시 숙지하시기 바랍니다!

구분	[스칼라]	[슬라이스]	[리스트]
리스트	원소	리스트	불가능
배열	원소	배열	배열
시리즈	원소	시리즈	시리즈
데이터프레임	시리즈	데이터프레임	데이터프레임

[참고] 인덱서 없는 인덱싱 결과 비교

- 시리즈와 데이터프레임에서 인덱서 없이 인덱싱할 때 주의해야 할 사항입니다.

시리즈		데이터프레임	
[행이름 스칼라]	원소를 반환	[열이름 스칼라]	열을 시리즈로 반환
[행이름 슬라이스]	원하는 결과를 얻을 수 없음	[열이름 슬라이스]	에러 발생
[행이름 리스트]	원소를 시리즈로 반환	[열이름 리스트]	열을 데이터프레임으로 반환
[부울형 시리즈]	원소를 시리즈로 반환	[부울형 시리즈]	행을 데이터프레임으로 반환

- 데이터프레임에서 조건을 만족하는 행을 필터링한 결과에서 일부 열을 선택하려면 반드시 인덱서를 사용해야 합니다. 슬라이싱 할 때에도 인덱서가 필요합니다.

데이터 입출력

작업 경로

- 외부 파일을 입출력할 때 함수에 '경로/파일명.확장자' 형태의 문자열을 인수로 추가하는데, 만약 파일의 경로가 상당히 길면 코딩이 번거로울 수 있습니다.
 - 만약 현재 작업 경로 `working directory`에 있는 파일이라면 경로를 생략할 수 있습니다.
 - 작업 경로는 현재 작업 중인 Jupyter Notebook 파일에 설정된 파일 탐색 경로입니다.
- 현재 작업 경로는 아래와 같습니다.

Windows	"C:\Users\계정명\Documents\PythonBasic-main\code"
MacOS	"/Users/계정명/Documents/PythonBasic-main/code"

[주의] 경로 구분자로 Windows는 역슬래시(\), MacOS는 슬래시(/)를 사용합니다. 만약 작업 경로를 변경하는 함수에 역슬래시가 하나인 경로를 지정하면 에러가 발생하므로 역슬래시를 추가하거나 슬래시로 변경해야 합니다.

절대 경로 vs 상대 경로

- 절대 경로는 경로의 시작과 끝을 모두 표기한 경로입니다.
 - 절대 경로는 현재 사용 중인 컴퓨터에서 유일한 경로입니다.
 - 작업 경로를 절대 경로로 설정하면 현재 작업 경로와 상관 없이 항상 결과가 같습니다.
 - 절대 경로의 길이가 상당히 긴 경우라면 코딩할 때 사용하기 불편합니다.
- 상대 경로는 현재 경로에서의 상대적인 위치를 의미합니다.
 - 상대 경로의 '.'은 현재 폴더, '..'은 상위 폴더를 의미합니다.
 - 상대 경로는 몇 글자만으로도 작업 경로를 설정할 수 있으므로 코딩할 때 편리합니다.
 - 그런데 현재 작업 경로에 따라 실행 결과가 달라지므로 주의를 기울여야 합니다.

작업 경로 확인 및 변경

- 관련 라이브러리를 호출합니다.

```
>>> import os # os 라이브러리는 작업 경로 설정과 관련한 함수를 포함하고 있습니다.
```

- 현재 작업 경로를 확인합니다.

```
>>> os.getcwd() # 현재 작업 경로는 code 폴더입니다.  
[참고] Jupyter Notebook 파일이 있는 폴더로 현재 작업 경로를 자동 설정합니다.
```

- 데이터 파일이 있는 폴더로 작업 경로를 변경합니다.(절대 경로 vs 상대 경로)

```
>>> os.chdir(path = 'C:\\Users\\계정명\\Documents\\PythonBasic-main\\data')
```

```
>>> os.chdir(path = '../data') # 이번 강의에서는 상대 경로를 사용하여 작업 경로를 변경합니다.
```


작업 경로에 있는 폴더명과 파일명 확인

- 현재 작업 경로에 있는 폴더명과 파일명을 출력합니다.

```
>>> os.listdir() # 현재 작업 경로에 있는 폴더명과 파일명을 리스트로 반환합니다.
```

```
>>> os.listdir(path = '../code') # [참고] path 매개변수에 경로명을 문자열로 지정하면 해당 경로에  
# 있는 폴더명과 파일명을 리스트로 반환합니다.
```

- 현재 작업 경로를 출력했을 때 오름차순 정렬하고 싶으면 아래 코드를 참고하세요.

```
>>> files = os.listdir() # [참고] Windows는 파일명과 폴더명을 오름차순 정렬하여 리스트로 반환하지만  
# MacOS는 문자열을 정렬하지 않고 반환하는 경우가 있습니다.
```

```
>>> files.sort() # files는 리스트이므로, 리스트의 원소를 오름차순 정렬하려면 sort() 함수를 실행합니다.  
# [참고] sort() 함수는 실행 결과를 리스트에 반영하므로 재할당하지 않아도 됩니다.
```

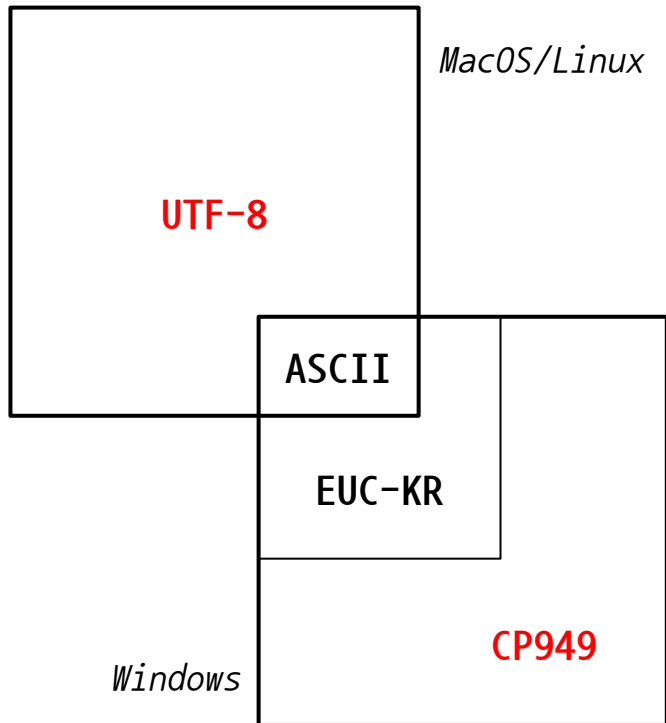
```
>>> files # files를 출력하면 오름차순 정렬되어 있습니다. 외부 파일을 읽을 때 파일명을 직접 타이핑하지 말고  
# 출력된 파일명을 복사하여 붙여넣기 하면 타이핑 에러가 발생하는 것을 막을 수 있습니다.
```

[참고] 한글 인코딩의 이해

- 컴퓨터는 사람이 사용하는 자연어^{Natural Language} 문자를 이해할 수 없습니다.
 - 자연어 문자를 컴퓨터가 이해할 수 있는 코드로 변환하는 것을 인코딩^{Encoding}이라고 하며, 반대로 컴퓨터가 이해하는 코드를 자연어로 변환한 것을 디코딩^{Decoding}이라고 합니다.
 - 초창기에는 자연어를 2진수 코드로 변환하였지만, 나중에 16진수로 발전했습니다.
- 컴퓨터 운영체제별로 기본 한글 인코딩 방식이 다릅니다.

운영체제	인코딩 방식	16진수 코드	자연어 문자
 	CP949 (EUC-KR) UTF-8	0xB0A1 U+AC00	

[참고] 한글 인코딩 방식 관계도



구분	설명
ASCII	<ul style="list-style-type: none"> 미국에서 개발한 대표적인 영문 인코딩 방식입니다. 알파벳, 숫자, 기호 등 128개 문자를 포함합니다.
UTF-8	<ul style="list-style-type: none"> 유니코드에 기반한 인코딩 방식입니다.(가변 길이) 한글 완성형/조합형 모두 포함합니다.(국제 표준)
Unicode	<ul style="list-style-type: none"> 전 세계 모든 문자를 포함하는 인코딩 방식입니다. 한 글자를 나타내기 위해 4 bytes를 사용합니다.
EUC-KR	<ul style="list-style-type: none"> 한글 초기 완성형 인코딩 방식입니다.(2350자) 조합형은 다른 문자 체계와 호환할 수 없습니다.
CP949	<ul style="list-style-type: none"> 8822자를 추가한 통합 완성형 인코딩 방식입니다. Windows 점유율 높은 한국에서 사실상 표준입니다.

새 텍스트 파일 생성

- 쓰기모드로 텍스트 파일을 엽니다.

```
>>> file = open(file = 'test.txt', mode = 'w', encoding = 'UTF-8')  
# 현재 작업 경로에 있는 'test.txt' 파일을 쓰기모드로 엽니다. 파일이 없으면 새로 생성합니다.  
[참고] Windows 사용자는 encoding을 추가하지 않으면 'CP949'를 자동 적용합니다.
```

- 텍스트 파일에 문자열을 입력합니다.

```
>>> for i in range(1, 6): # 반복문을 실행하여 여러 문자열을 텍스트 파일에 입력합니다.  
  
    file.write(f'{i} 페이지 수집!\n')
```

- 텍스트 파일을 닫습니다.

```
>>> file.close() # 텍스트 파일을 닫으면 텍스트 파일을 저장합니다.
```

기존 텍스트 파일에 문자열 추가

- 추가모드로 텍스트 파일을 엽니다.

```
>>> file = open(file = 'test.txt', mode = 'a', encoding = 'UTF-8')
```

현재 작업 경로에 있는 'test.txt' 파일을 추가모드로 엽니다. 파일이 없으면 새로 생성합니다.

- 텍스트 파일에 문자열을 입력합니다.

```
>>> for i in range(11, 16):
```

```
    file.write(f'{i} 페이지 수집!\n')
```

- 텍스트 파일을 닫습니다.

```
>>> file.close()
```

텍스트 파일을 문자열로 읽기

- 문자열 읽기모드로 텍스트 파일을 엽니다.

```
>>> file = open(file = 'test.txt', mode = 'r', encoding = 'UTF-8')
```

현재 작업 경로에 있는 'test.txt' 파일을 문자열 읽기모드로 엽니다. 파일이 없으면 에러가 발생합니다.

- 텍스트 파일을 읽고 출력합니다.

```
>>> text = file.read(); text
```

read() 함수는 file을 사람이 읽을 수 있는 문자열`str`로 반환합니다.
[참고] readlines() 함수는 리스트로 반환합니다.

```
>>> type(text)
```

text의 클래스를 확인합니다. text의 클래스는 str입니다.
[참고] str은 사람이 읽을 수 있는 문자열입니다.

```
>>> text.encode(encoding = 'UTF-8')
```

문자열을 인코딩하면 바이너리 코드로 변환합니다.

텍스트 파일을 바이너리로 읽기

- 바이너리 읽기모드로 텍스트 파일을 엽니다.

```
>>> file = open(file = 'test.txt', mode = 'rb')
```

현재 작업 경로에 있는 'test.txt' 파일을 바이너리 읽기모드로 엽니다. 파일이 없으면 에러가 발생합니다.
[참고] 바이너리 모드로 읽을 때 인코딩 방식을 추가할 수 없습니다.

- 텍스트 파일을 읽고 출력합니다.

```
>>> text = file.read(); text
```

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

```
>>> text.decode(encoding = 'UTF-8') # 바이너리 코드를 디코딩하면 문자열로 변환합니다.
```

[주의] Windows에서 'test.txt' 파일을 생성할 때 **encoding = 'UTF-8'**를 추가하지 않았다면 반드시 'EUC-KR' 또는 'CP949'를 지정해야 합니다.

관련 라이브러리 호출

- 관련 라이브러리를 설치합니다.

```
>>> !pip install chardet # chardet 라이브러리를 설치합니다. 왼쪽 코드는 한 번만 실행합니다.  
[참고] Jupyter Notebook에서 라이브러리를 설치하려면 느낌표를 추가합니다.
```

```
>>> !pip install joblib # joblib 라이브러리를 설치합니다.
```

- 관련 라이브러리를 호출합니다.

```
>>> import chardet # 텍스트 파일의 문자열 인코딩 방식을 확인하는 함수를 포함하고 있습니다.
```

```
>>> import numpy as np
```

```
>>> import pandas as pd
```

```
>>> import joblib # Python 객체를 압축 파일로 저장하고 호출하는 함수를 포함하고 있습니다.
```


xlsx 파일 입출력

- xlsx 파일을 읽고 데이터프레임을 생성합니다.

```
>>> df = pd.read_excel(io = 'KBO_Hitters_2021.xlsx', # [참고] 만약 에러가 발생했다면
                        openpyxl 라이브러리를 설치하세요.
                        sheet_name = 'Hitters', # 시트 인덱스 또는 시트명 하나만 지정합니다.
                                                (기본값: 0)
                        skiprows = 4) # 열이름 위로 생략할 행 개수를 지정합니다.
```

- df를 출력합니다.

```
>>> df # [참고] 데이터프레임의 행과 열 개수가 많으면 모든 행과 열을 출력하지 않고 결과 창 크기에 따라 출력할
        행과 열 개수를 자동 계산하며, 생략하는 행과 열 사이에 ... 기호를 대신 출력합니다.
```

- 데이터프레임을 xlsx 파일로 저장합니다.

index = None을 추가하면 xlsx 파일을 저장할 때
인덱스를 삭제합니다.

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

[참고] 데이터프레임 행/열 최대 출력 옵션 변경

- 최대 행 출력 옵션을 확인하고 변경합니다.

```
>>> pd.get_option('display.max_rows') # 최대 행 출력 개수를 출력합니다.(기본값: 60)
[참고] 최대 열 출력 개수는 max_columns를 사용합니다.
```

```
>>> pd.set_option('display.max_rows', None) # 최대 행 출력 개수에 None을 할당하면 전체 행을
출력합니다.
```

- df를 출력하면 전체 행을 출력합니다.

```
>>> df # [참고] 전체 행을 출력하도록 옵션을 변경하면 Jupyter Notebook이 느려지거나 다운될 수 있습니다.
```

- 최대 행 출력 옵션을 초기화합니다.

```
>>> pd.reset_option('display.max_rows') # [참고] 전체 옵션을 초기화하려면 'all'을 지정합니다.
```

```
>>> pd.options.display.max_rows # 왼쪽 코드를 실행하면 최대 행 출력 개수를 출력합니다.
```

데이터프레임 미리보기

- 데이터프레임의 처음 5행을 출력합니다.

```
>>> df.head() # [참고] n 매개변수에 입력하는 개수만큼 출력합니다. (기본값: 5)
```

```
>>> df.head(n = 10) # 데이터프레임의 처음 10행을 출력합니다.
```

- 데이터프레임의 마지막 5행을 출력합니다.

```
>>> df.tail()
```

- 데이터프레임을 무작위로 1행을 추출합니다.

```
>>> df.sample() # [참고] n 매개변수에 입력하는 개수만큼 출력합니다. (기본값: None)  
                random_state 매개변수에 같은 정수를 지정하면 항상 같은 결과를 반환합니다.
```

[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	$(\text{안타} - \text{홈런}) \div (\text{타수} - \text{삼진} - \text{홈런} - \text{희생타})$ # 인플레이 타율
타율	$\text{안타} \div \text{타수}$ # 타율 3할은 우수한 타자의 기준
출루율	$(\text{안타} + \text{볼넷} + \text{사구}) \div (\text{타수} + \text{볼넷} + \text{사구} + \text{희생타})$ # 출루율 4할이 우수
장타율	$(1\text{루타} + 2\text{루타} \times 2 + 3\text{루타} \times 3 + \text{홈런} \times 4) \div \text{타수}$ # 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

csv 파일 입출력

- csv 파일명을 재사용할 수 있도록 변수에 할당합니다.

```
>>> file = 'KB0_Hitters_2021.csv'
```

- csv 파일을 읽고 데이터프레임을 생성합니다.

```
>>> df = pd.read_csv(filepath_or_buffer = file) # [주의] csv 파일의 인코딩 방식이 'UTF-8'이  
# 아니면 인코딩 방식을 지정해야 합니다!
```

- csv 파일을 바이너리 모드로 읽습니다.

```
>>> text = open(file = file, mode = 'rb').read() # text는 bytes 코드이므로 출력하면 사람이  
# 읽을 수 없습니다.
```

- csv 파일의 문자 인코딩 방식을 확인합니다.

```
>>> chardet.detect(text) # 바이너리 코드의 인코딩 방식과 신뢰도(confidence)를 반환합니다.
```

csv 파일 입출력(계속)

- csv 파일을 읽고 데이터프레임을 생성합니다.

```
>>> df = pd.read_csv(filepath_or_buffer = file, encoding = 'EUC-KR')
```

- df의 인덱스를 정수 1부터 시작하도록 변경합니다.

```
>>> df.index = range(1, df.shape[0] + 1) # [참고] df.shape은 행과 열 개수를 튜플로 반환합니다.
```

```
>>> df.head() # [참고] df.shape은 행과 열 개수를 튜플로 반환합니다.
```

- 데이터프레임을 csv 파일로 저장합니다.

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

[참고] csv 파일의 인코딩 방식을 지정할 수 있습니다.

압축 파일 입출력

- 데이터프레임을 포함한 모든 Python 객체를 외부 파일로 저장하려면 압축 파일을 사용하는 것이 좋습니다.
 - **joblib** 라이브러리는 Python 객체를 다양한 형태의 압축 파일로 저장하고, 압축 파일을 Python으로 호출하는 함수를 포함하고 있습니다.
- df를 확장자가 z인 압축 파일로 저장합니다.

```
>>> joblib.dump(value = df, filename = 'test.z')
```

- z 파일을 호출하고 데이터프레임에 할당합니다.

```
>>> df1 = joblib.load(filename = 'test.z')
```

[참고] 실습 파일 삭제

- 현재 작업 경로에서 'test'를 포함하는 파일명으로 리스트를 생성합니다.

```
>>> files = [file for file in os.listdir() if 'test' in file]
```

```
>>> files # files에는 이번 시간에 생성한 4개 파일명을 원소로 갖습니다.
```

- 반복문으로 해당 파일을 모두 삭제합니다.

```
>>> for file in files:
```

```
    os.remove(path = file) # [참고] os.remove() 함수는 한 번에 하나의 파일을 삭제합니다.
```

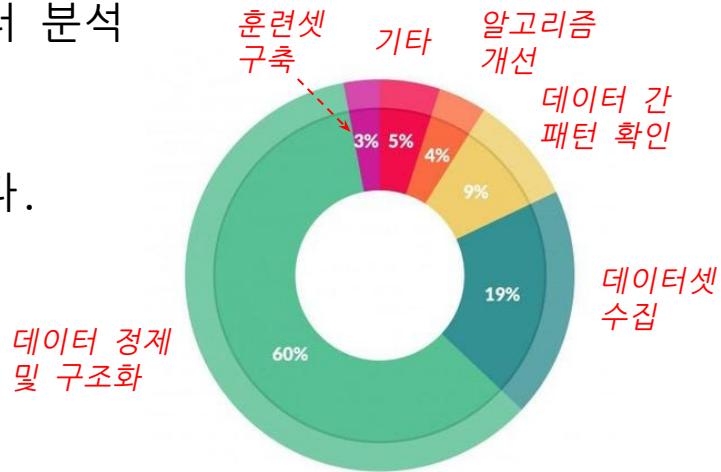
- 현재 작업 경로에 있는 폴더명과 파일명을 출력합니다.

```
>>> os.listdir() # 'test'를 포함하는 파일을 모두 삭제했습니다.
```


데이터 전처리

데이터 전처리

- 데이터의 품질이 머신러닝 알고리즘보다 분석 결과에 더 직접적인 영향을 미치기 때문에 데이터 전처리의 중요성을 강조해도 지나치지 않습니다.
- 데이터 과학자는 데이터 수집/전처리에 데이터 분석 전 과정의 80% 시간을 사용한다고 합니다.
- 데이터 전처리는 아래 과정을 위해 수행합니다.
 - 결측값과 이상치를 탐지하고 처리합니다.
 - 데이터에 잠재된 패턴과 특징을 파악합니다.
 - 다양한 파생변수를 생성합니다.



출처: Forbes

데이터프레임 전처리 항목

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

데이터프레임 전처리 항목(계속)

- 파생변수를 생성합니다.
- 데이터프레임을 오름차순 또는 내림차순 정렬합니다.
- 그룹을 설정하고 집계함수로 데이터를 요약합니다.
- 데이터프레임의 형태를 변환합니다.(Long type ↔ Wide type)
- 두 데이터프레임을 행 또는 열 방향으로 결합합니다.
- 데이터프레임의 특정 열 기준 중복 여부를 확인하고 필요시 삭제합니다.
- 두 데이터프레임에서 서로 일치하는 행을 병합합니다.

실습 데이터셋 소개

- 공공데이터포털에서 제공하는 국토교통부 아파트매매 실거래자료 중 2021년 동안 서울특별시에서 거래된 아파트 가격 데이터를 2가지 형태로 제공합니다.
 - 'APT_Price_Seoul_2021.csv'
 - 'APT_Price_Seoul_2021.xlsx'
- 국내 대형 포털에서 제공하는 서울특별시 아파트단지별 상세정보를 2가지 형태로 제공합니다.
 - 'Naver_APT_Detail_Seoul.csv'
 - 'Naver_APT_Detail_Seoul.xlsx'

관련 라이브러리 호출

- 관련 라이브러리를 호출합니다.

```
>>> import os
```

```
>>> import chardet
```

```
>>> import numpy as np
```

```
>>> import pandas as pd
```

```
>>> import joblib
```

작업 경로 확인 및 변경

- 현재 작업 경로를 확인합니다.

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

- data 폴더로 작업 경로를 변경합니다.

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

- 현재 작업 경로에 있는 폴더명과 파일명을 출력합니다.

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

xlsx 파일을 읽고 데이터프레임 생성

- 가격 xlsx 파일을 읽고 데이터프레임을 생성합니다.

코드 앞에 `%time`을 추가하면
코드 실행 시간을 반환합니다.

```
>>> price = pd.read_excel(io = 'APT_Price_Seoul_2021.xlsx') # 약 5초 소요됩니다.
```

- price의 정보를 확인합니다.

```
>>> price.info() # 거래일의 자료형은 numpy.datetime64입니다. [참고] xlsx 파일은 셀서식을 유지합니다.
```

- price의 처음 5행을 출력합니다.

```
>>> price.head()
```


csv 파일 인코딩 방식 확인

- 가격 csv 파일명을 재사용할 수 있도록 변수에 할당합니다.

```
>>> file = 'APT_Price_Seoul_2021.csv'
```

- csv 파일을 바이너리 모드로 읽습니다.

```
>>> text = open(file = file, mode = 'rb').read()
```

```
>>> text[:100] # [참고] csv 파일 용량이 크면 일부만 출력하는 것이 좋습니다.
```

- csv 파일의 문자 인코딩 방식을 확인합니다.

```
>>> chardet.detect(text[:100]) # csv 파일의 문자 인코딩 방식을 확인합니다.  
[참고] csv 파일 용량이 크면 오래 걸리므로 일부만 확인합니다.
```

csv 파일을 읽고 데이터프레임 생성

- 가격 csv 파일을 읽고 데이터프레임을 생성합니다.

```
>>> price = pd.read_csv(filepath_or_buffer = file, encoding = 'UTF-8')
```

- price의 열별 자료형을 확인합니다.

[참고] csv 파일의 인코딩 방식이 'UTF-8'이면
encoding 매개변수를 생략할 수 있습니다.

```
>>> price.dtypes # 거래일의 자료형이 object입니다. [참고] csv 파일은 날짜 데이터를 문자열로 읽습니다.
```

- 날짜형으로 읽을 열이름을 parse_dates 매개변수에 리스트로 지정합니다.

```
>>> df = pd.read_csv(filepath_or_buffer = file, parse_dates = ['거래일'])
```

- df의 열별 자료형을 확인합니다.

문자열이 날짜 기본형('yyyy-mm-dd', 'yyyy/mm/dd', 'yyyy.mm.dd',
'yyyy mm dd', 'yyyymmdd')이면 날짜형으로 읽을 수 있습니다.

```
>>> df.dtypes # df의 열별 자료형을 다시 확인합니다. 거래일의 자료형은 numpy.datetime64입니다.
```

[참고] xlsx 파일 vs csv 파일

- xlsx 파일과 csv 파일을 읽을 때의 다음과 같은 장점과 단점이 있습니다.

구분	xlsx 파일	csv 파일
장점	인코딩 방식 확인 불필요 날짜/시간 셀서식 유지	입출력 속도가 매우 빠름
단점	입출력 속도가 매우 느림	인코딩 방식 확인 필요 날짜/시간을 문자열로 생성

- csv 파일의 셀서식 관련 단점을 보완하는 두 가지 방법입니다.
 - 날짜/시간 열이름을 먼저 문자형으로 읽고 나중에 날짜형으로 변환합니다.
 - `pd.read_csv()` 함수의 `parse_dates` 매개변수에 날짜형으로 읽을 열이름을 지정합니다.

열 선택

- 열을 선택할 때 대괄호 안에 열이름을 나열합니다.

```
>>> price['지역코드'] # 열이름을 문자열 스칼라로 지정하면 시리즈로 반환합니다.
```

```
>>> price[['지역코드']] # 열이름을 리스트로 지정하면 데이터프레임으로 반환합니다.  
[참고] 2개 이상의 열을 선택하려면 리스트로 지정합니다.
```

```
>>> price[['아파트', '지역코드']] # 열이름 순서를 변경하면 위치를 바꿔서 반환합니다.
```

- 슬라이스로 연속한 열을 선택하려면 loc 인덱서를 추가합니다.(배열 인덱싱)

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

- 조건을 만족하는 열을 선택합니다.(불리언 인덱싱)

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

열 삭제

- 삭제할 열이름을 `drop()` 함수에 리스트로 지정합니다.

```
>>> price.drop(columns = ['지역코드'])
```

[주의] columns 매개변수를 생략하면 반드시 axis = 1을 추가해야 합니다. 추가하지 않으면 인덱스에서 찾습니다.

- `price`의 처음 5행을 출력합니다.

```
>>> price.head()
```

price에는 여전히 지역코드를 포함하고 있습니다.

- 열을 삭제하고 재할당하면 데이터프레임에서 해당 열을 삭제합니다.

```
>>> price = price.drop(columns = ['지역코드'])
```

```
>>> price.head()
```

price에서 지역코드를 삭제했습니다.

열이름 변경

- 일부 열이름을 변경한 결과를 출력합니다.

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

'기존 이름': '새 이름'으로 설정합니다.

- price의 열이름을 출력합니다.

[주의] columns 매개변수를 생략하면 안됩니다!

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

- 전체 열이름을 변경합니다. # [주의] 데이터프레임의 열이름과 원소 개수가 같은 리스트를 지정해야 합니다.

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

```
>>> price.head()
```

열별 자료형 변환

- 시리즈의 원소 자료형을 변환합니다.

```
>>> price['거래금액'].astype(dtype = float) # [참고] 거래금액 원소에逗가 있으므로 실수로 변환할 수 없습니다.
```

- 거래금액 원소에 있는逗를 삭제합니다. # [참고] 시리즈.str에 문자열을 처리하는 다양한 함수가 있습니다.

```
>>> price['거래금액'] = price['거래금액'].str.replace(pat = ',', repl = '')
```

- 시리즈의 원소 자료형을 변환하고 재할당합니다.

```
>>> price['거래금액'] = price['거래금액'].astype(dtype = float)
```

```
>>> price.dtypes # price의 열별 자료형을 확인합니다. 거래금액을 numpy.float64로 변환했습니다.
```

열별 자료형 변환(계속)

- 데이터프레임의 열별로 자료형 변환 방법을 딕셔너리로 지정합니다.

```
>>> price = price.astype(dtype = {'거래일': np.datetime64, '총': float})
```

```
>>> price.dtypes # price의 열별 자료형을 확인합니다.  
                거래일을 numpy.datetime64, 총을 numpy.float64로 변환했습니다.
```

- 데이터프레임의 여러 열을 같은 자료형으로 일괄 변환합니다.

```
>>> cols = ['총', '거래금액'] # 정수형으로 변환할 열이름으로 리스트를 생성합니다.
```

```
>>> price[cols] = price[cols].astype(dtype = int) # 선택한 열의 자료형을 일괄 변환합니다.
```

```
>>> price.dtypes # 총과 거래금액을 numpy.int64로 변환했습니다.
```


[참고] 문자열을 날짜형으로 변환

- 날짜 기본형이 아닌 문자열 리스트로 시리즈를 생성합니다.

```
>>> birth = pd.Series(data = ['2001년 2월 3일']); birth
```

[참고] birth의 자료형은 object입니다.

- birth를 날짜형으로 변환하려고 하면 에러가 발생합니다.

```
>>> birth.astype(dtype = np.datetime64)
```

- `to_datetime()` 함수는 문자열을 날짜형으로 변환합니다. # [주의] 반드시 `format` 매개변수에 날짜 포맷을 지정해야 합니다.

```
>>> birth = pd.to_datetime(arg = birth, format = '%Y년 %m월 %d일')
```

```
>>> birth.iloc[0] # birth의 0번 인덱스 원소를 출력합니다.
```

[참고] 날짜/시간 관련 주요 포맷

- 아래 표는 날짜/시간 포맷을 정리한 것입니다.

포맷	상세 내용	예시	포맷	상세 내용	예시
%a	요일(영어 약자)	Wed	%M	분(정수)	01
%A	요일(영어)	Wednesday	%p	AM/PM 표기	AM
%b	월(영어 약자)	Jan	%S	초(정수)	01
%B	월(영어)	January	%w	요일(정수)	3
%c	날짜와 시간	time.ctime()	%x	날짜만 출력	01/01/20
%d	일(정수)	01	%X	시간만 출력	01:01:01
%H	시(정수-24시간)	12	%Y	연도(4자리)	2022
%I	시(정수-12시간)	01	%y	연도(2자리)	20
%m	월(정수)	01	%Z	타임존 ^{time zone} 출력	KST

[참고] 날짜 시간 데이터 연산

- 현재 날짜 시간 데이터를 생성합니다.

```
>>> ctime = pd.Timestamp.today(); ctime
```

현재 날짜를 '년-월-일 시:분:초.마이크로초' 형태로 반환합니다.

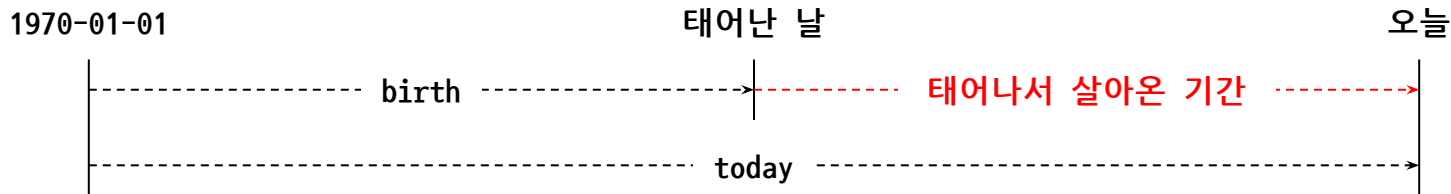
- 날짜 시간 데이터는 산술 연산이 가능합니다.

```
>>> dtGap = ctime - birth.iloc[0]
```

태어나서 현재까지 살아온 기간을 dtGap에 할당합니다.

```
>>> dtGap.days
```

dtGap에서 경과일수를 출력합니다.



[참고] 날짜 분해 함수

- 날짜 변수에서 년, 월, 일을 정수형 시리즈로 반환합니다.

```
>>> price['거래일'].dt.year # 거래일에서 년year을 정수형 시리즈로 반환합니다.
```

```
>>> price['거래일'].dt.month # 거래일에서 월month을 정수형 시리즈로 반환합니다.
```

```
>>> price['거래일'].dt.day # 거래일에서 일day을 정수형 시리즈로 반환합니다.
```

- 날짜 변수에서 요일을 문자형 시리즈로 반환합니다.

```
>>> price['거래일'].dt.day_name() # 거래일에서 영문 요일을 문자형 시리즈로 반환합니다.
```

```
>>> price['거래일'].dt.day_name(locale = 'ko_KR') # locale 매개변수에 'ko_KR'을 지정하면  
한글 요일을 반환합니다.
```

조건에 맞는 행 선택: 연속형 변수

- 거래금액이 100억 이상인 행을 선택하여 df1에 할당합니다.

```
>>> df1 = price[price['거래금액'] >= 10000000] # [참고] 불리언 인덱싱으로 행을 선택할 때 loc
# 인덱서를 생략할 수 있습니다.
```

```
>>> df1.head() # df1의 처음 5행을 출력합니다.
# [참고] 행이름이 0부터 시작하지 않습니다.
```

- 거래금액이 100억 미만이고 60층 이상인 행을 선택하여 df2에 할당합니다.

```
>>> df2 = price[(price['거래금액'] < 10000000) & (price['층'] >= 60)]
```

시리즈로 논리곱 연산을 실행할 때 논리 연산자를 사용하면 에러가
발생하며 비트 연산자를 대신 사용해야 합니다.

```
>>> df2.head()
```

[주의] 비트 연산자 앞뒤 코드를 반드시 소괄호로 감싸야 합니다.

[참고] 시리즈의 비교 연산 함수

- 시리즈에서 비교 연산을 실행하는 함수는 불리언 시리즈를 반환합니다.

```
>>> price['층'].gt(60).sum() # 층이 60 초과면 True, 아니면 False인 부울형 시리즈를 반환합니다.  
[참고] 마지막에 추가한 sum() 함수는 True의 개수를 반환합니다.
```

```
>>> price['층'].ge(60).sum() # 층이 60 이상이면 True, 아니면 False인 부울형 시리즈를 반환합니다.
```

```
>>> price['층'].lt(60).sum() # 층이 60 미만이면 True, 아니면 False인 부울형 시리즈를 반환합니다.
```

```
>>> price['층'].le(60).sum() # 층이 60 이하이면 True, 아니면 False인 부울형 시리즈를 반환합니다.
```

```
>>> price['층'].eq(60).sum() # 층이 60이면 True, 60이 아니면 False인 부울형 시리즈를 반환합니다.
```

```
>>> price['층'].ne(60).sum() # 층이 60이 아니면 True, 60이면 False인 부울형 시리즈를 반환합니다.
```

```
>>> price['거래금액'].lt(1000000) & price['층'].ge(60) # 비교 연산 함수는 시리즈이므로  
소괄호로 감싸지 않아도 됩니다.
```

조건에 맞는 행 선택: 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')]
```

- **isin()** 함수는 시리즈 원소가 리스트에 있으면 True, 없으면 False를 반환합니다.

```
>>> price[price['자치구'].isin(values = ['강남구', '서초구'])]
```

- **str.contains()** 함수는 원소에 패턴이 있으면 True, 없으면 False를 반환합니다.

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

[참고] 시리즈를 문자열로 처리하는 주요 함수

- 시리즈를 문자열로 처리할 때 str을 추가합니다. # [주의] str은 시리즈 원소를 문자열로 처리하기 때문에 문자열이 아니면 에러가 발생합니다.

```
>>> drID = pd.Series(data = ['서울 00-123456-01', '경기 01-654321-02'])
```

```
>>> drID.str.split(pat = ' |-', expand = True) # 문자열(원소)을 지정한 패턴으로 분리하고  
결과를 데이터프레임으로 반환합니다.
```

```
>>> drID.str.find(sub = '서울') # 문자열(원소)마다 지정한 패턴이 있으면 시작 인덱스를 반환합니다.  
[참고] 지정한 패턴이 없으면 -1을 반환합니다.
```

```
>>> drID.str.replace(pat = ' ', repl = '') # 문자열(원소)마다 지정한 패턴을 교체할 문자열로  
변경합니다.
```

```
>>> drID.str.slice(start = 0, stop = 2) # 문자열(원소)을 지정한 인덱스로 자릅니다.
```

```
>>> drID.str.extract(pat = r'([가-힣]+)') # 문자열(원소)마다 지정한 패턴에 해당하는 문자열을  
추출합니다. [주의] 패턴을 소괄호로 감싸야 합니다.
```


행이름으로 행 삭제 및 행이름 초기화

- 삭제할 행이름을 `drop()` 함수에 리스트로 지정합니다.

```
>>> df1 # df1을 출력합니다. [참고] 인덱스(행이름)가 0부터 시작하지 않습니다.
```

```
>>> df1.drop(index = [10512, 10513]) # [주의] 행이름에 없는 값을 입력하면 에러가 발생합니다.
```

```
>>> df1.drop(index = df1.index[0:2]) # 인덱스를 슬라이싱하면 연속하는 행이름으로 삭제합니다.
```

- 데이터프레임의 행이름을 초기화합니다.

```
>>> df1.reset_index() # 행이름을 초기화하면 기존 행이름을 index라는 열로 추가합니다.  
[참고] 기존 행이름을 index 열로 추가하지 않으려면 drop = True를 추가합니다.
```

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

결측값 처리: 단순대체

- 데이터프레임의 셀 값별 결측값 여부를 확인합니다.

```
>>> price.isna() # isna() 함수는 결측값 여부를 True 또는 False로 반환합니다.  
[참고] isnull() 함수는 isna() 함수의 alias이므로 같은 동작을 수행합니다.
```

- 데이터프레임의 열별 결측값 개수를 확인합니다.

```
>>> price.isna().sum() # 지번에 결측값이 14개 있습니다.
```

- 지번이 결측값인 행을 선택합니다.

```
>>> price[price['지번'].isna()]
```

- 지번이 결측값인 행에서 결측값을 빈 문자열로 대체한 결과를 반환합니다.

```
>>> price[price['지번'].isna()].fillna(value = '')
```

[참고] 결측값을 이전 셀 값으로 채우기

- 현재 작업 경로에 있는 폴더명과 파일명을 출력합니다.

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

- 법정동별 평균 거래금액 데이터를 읽고 데이터프레임을 생성합니다.

```
>>> meanPrice = pd.read_excel(io = 'APT_Mean_Price_Dong_2021.xlsx')
```

- meanPrice를 출력합니다.

```
>>> meanPrice # 자치구에 결측값이 많습니다.  
[참고] xlsx에서 병합한 셀은 처음 값만 제대로 읽고 나머지는 결측값으로 대체합니다.
```

- 자치구에 있는 결측값을 이전 셀 값으로 채웁니다.

```
>>> meanPrice.fillna(method = 'ffill') # [참고] 결측값을 이후 셀 값으로 채우려면 'bfill'을 대신 지정합니다.
```

[참고] 데이터프레임에서 결측값이 있는 행 삭제

- 결측값이 있는 행을 삭제하기 전에 변수의 중요도를 판단하는 것이 좋습니다.
 - 아래 데이터프레임에서 결측값이 있는 행을 삭제하면 모든 행을 삭제합니다.
 - 만약 지번이 중요한 변수라면 정상값을 갖는 행도 삭제하므로 행 개수가 감소합니다.
 - 지번에서 결측값이 있는 행만 삭제하고 적당한 값으로 결측을 대체하는 것이 좋습니다.

아파트	시도	자치구	읍면동	지번	...
래미안	서울특별시	강남구	압구정동	NaN	...
힐스테이트	서울특별시	강동구	NaN	2	...
아이파크	서울특별시	NaN	성북동	4	...
⋮	⋮	⋮	⋮	⋮	...

결측값 처리: 행 삭제

- 데이터프레임에서 결측값이 있는 모든 행을 삭제하고 행 개수를 확인합니다.

```
>>> price.dropna().shape[0] # [참고] dropna() 함수의 subset 매개변수에 일부 열이름을 지정하면 해당 열에서 결측값이 있는 행을 삭제합니다.
```

- 지번에서 결측값이 있는 행을 선택하고 행 개수를 확인합니다.

```
>>> price[price['지번'].isna()].shape[0]
```

- 지번에서 결측값이 없는 행을 선택하고 행 개수를 확인합니다.

```
>>> price[price['지번'].notna()].shape[0] # notna() 함수는 결측값이 아니면 True, 결측값이면 False를 반환합니다.
```

- 지번에서 결측값이 아닌 행을 선택하고 price에 재할당합니다.

```
>>> price = price[price['지번'].notna()] # [참고] notna() 함수 대신 isna() 함수 실행 결과를 반전하도록 ~ 연산자를 사용할 수 있습니다.
```

파생변수 생성: 연속형 변수

- 거래금액을 전용면적으로 나누고 3.3을 곱한 평당금액을 생성합니다.

```
>>> price['평당금액'] = price['거래금액'] / price['전용면적'] * 3.3
```

```
>>> price.head() # [참고] 데이터프레임에 없는 열이름으로 시리즈를 생성하면 데이터프레임의 가장 오른쪽에  
해당 열이름을 추가합니다.
```

- 거래금액의 단위를 만원에서 억원으로 변경합니다.

```
>>> price['거래금액'] = price['거래금액'] / 10000
```

- pandas 옵션에서 실수를 출력하는 소수점 자리수를 3으로 설정합니다.

```
>>> pd.options.display.precision = 3
```

```
>>> price.head() # [참고] 평당금액을 소수점 셋째 자리까지 출력합니다.
```

[참고] 데이터프레임에 열 추가 및 삽입

- price에서 일부 열을 삭제한 데이터프레임 imsi를 생성합니다.

```
>>> imsi = price.drop(columns = ['층', '평당금액']); imsi
```

- imsi에 여러 시리즈를 추가한 결과를 출력합니다.

```
>>> imsi.assign(층 = price['층'], # [참고] assign() 함수는 실행 결과를 출력하므로 데이터프레임에  
# 반영하려면 재할당해야 합니다.
```

```
    평당금액 = price['평당금액'])
```

- imsi의 원하는 위치에 시리즈를 지정한 열이름으로 삽입합니다.

```
>>> imsi.insert(loc = 0, value = price['평당금액'], column = '평당금액')
```

```
>>> imsi.head() # [참고] insert() 함수는 데이터프레임에 하나의 시리즈를 삽입할 수 있습니다.  
# 아울러 실행 결과를 데이터프레임에 업데이트하므로 재할당하지 않아도 됩니다.
```

[참고] 시리즈 데이터 지연

- imsi의 거래금액을 1번 지연한 시리즈를 반환합니다.

```
>>> imsi['거래금액'].shift(1) # 시계열(일별) 데이터를 1번 지연하면 1일 시차(lag)를 두는 것입니다.  
[참고] 지연한 개수만큼 결측값으로 채웁니다.
```

- imsi의 거래금액을 1~2번 지연한 파생변수를 추가합니다.

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

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

```
>>> imsi.head()
```


[참고] 시리즈 이동평균

- 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() # [참고] 첫 번째 원소부터 rolling을 시작하며 window에 지정한  
                                   개수에 미달하면 결측값을 반환합니다.
```

```
>>> nums.rolling(window = 3, min_periods = 1).mean() # nums로 3일 이동평균을 계산할 때  
                                                       최소 원소 개수를 1로 지정합니다.
```

파생변수 생성: 범주형 변수

- 연속형 변수를 기준에 따라 2개 이상의 구간으로 나누어 범주형 변수로 변환하는 것을 구간화^{binning}라고 합니다.
 - 구간화를 통해 이상치와 비선형 문제를 해결하고, 예측값을 쉽게 처리할 수 있습니다.
- 평당금액을 '5천 이상', '5천 미만'으로 구분한 금액구분을 생성합니다.

```
>>> price['금액구분'] = np.where(price['평당금액'].ge(5000), # 첫 번째 인수에 조건을 지정합니다.
```

```
    '5천 이상', '5천 미만') # True면 첫 번째 값, False면 두 번째 값을 반환합니다.
```

```
>>> price.head()
```

```
# [참고] np.where() 함수에 조건만 지정하면 True인 인덱스를 대신 반환합니다.
```

[참고] pandas에서 추천하는 코딩 방식

- 평당금액이 5000 이상인 행의 새 변수(금액구분)에 '5천 이상'을 할당합니다.

```
>>> imsi.loc[imsi['평당금액'].ge(5000), '금액구분'] = '5천 이상'
```

```
>>> imsi.head() # 오른쪽에 금액구분을 추가하고 평당금액이 5000 이상이면 '5천 이상', 아니면 결측값(NaN)을  
# 채웁니다. 따라서 아래 코드를 실행하면 결측값을 '5천 미만'으로 채웁니다.
```

- 평당금액이 5000 미만인 행의 새 변수(금액구분)에 '5천 미만'을 할당합니다.

```
>>> imsi.loc[imsi['평당금액'].lt(5000), '금액구분'] = '5천 미만'
```

```
>>> imsi.head() # 금액구분에 결측값이 없습니다.
```

[참고] 구간화 함수

- 연속형 변수를 세 개 이상으로 분리할 때 `np.where()` 함수를 중첩합니다.

```
>>> np.where(price['평당금액'].ge(10000),
```

```
    '1억 이상', # 첫 번째 조건을 만족하면 '1억 이상'을 반환하고, 그렇지 않으면 두 번째 조건  
               만족 여부를 확인합니다.
```

```
    np.where(price['평당금액'].ge(5000),
```

```
        '5천 이상', # 두 번째 조건 만족 여부에 따라 '5천 이상' 또는 '5천 미만'을  
                   반환합니다.
```

```
        '5천 미만')) # 코드를 구성하는 요소는 단순하지만 가독성이 안 좋습니다.
```

[참고] 구간화 함수(계속)

- 연속형 변수를 세 개 이상으로 분리할 때 `np.select()` 함수를 사용합니다.

```
>>> np.select(condlist = [price['평당금액'].ge(10000),  
                           price['평당금액'].ge(5000),  
                           price['평당금액'].lt(5000)],  
              choicelist = ['1억 이상', '5천 이상', '5천 미만'])  
# 조건에 맞는 값을 반환합니다.
```

파생변수 생성: 문자형 변수 결합

- 여러 문자형 변수를 + 연산자로 결합합니다.

```
>>> price['시도'] + ' ' + price['자치구'] + ' ' + \ # 한 줄로 끝나지 않는 코드 마지막에
price['읍면동'] + ' ' + price['지번']           백슬래시를 추가합니다.
```

- 여러 문자형 열이름으로 리스트를 생성합니다.

```
>>> cols = ['시도', '자치구', '읍면동', '지번']
```

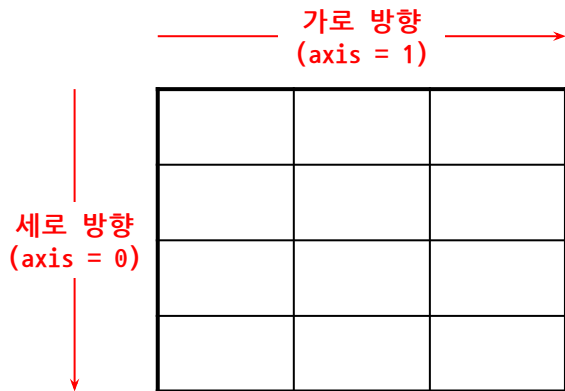
- 데이터프레임의 행(시리즈)별로 문자열을 결합하는 함수를 반복 실행합니다.

```
>>> price['주소'] = price[cols].apply(func = lambda x: ' '.join(x), axis = 1)

>>> price.head()
```

[참고] 같은 함수 반복 실행 함수

- `map()` 함수는 시리즈의 원소별로 지정한 함수를 반복 실행합니다.
- `apply()` 함수는 데이터프레임의 행 또는 열을 선택하고 시리즈별로 지정한 함수를 반복 실행합니다.
 - 행을 선택하는 것은 가로 방향으로 `axis = 1`을 추가합니다.(기본값: 0)
 - 열을 선택하는 것은 세로 방향으로 `axis = 0`을 추가하거나 생략합니다.
- `applymap()` 함수는 데이터프레임의 셀 값별로 지정한 함수를 반복 실행합니다.



[참고] 같은 함수 반복 실행 실습

- 아파트(시리즈)의 원소(문자열)별 글자수를 반환합니다.

```
>>> price['아파트'].map(arg = len)
```

- 데이터프레임의 열(시리즈) 또는 행(시리즈)별 원소 개수를 반환합니다.

```
>>> price[cols].apply(func = len, axis = 0)
```

```
>>> price[cols].apply(func = len, axis = 1)
```

- 데이터프레임의 셀 값(문자열)별 글자수를 반환합니다.

```
>>> price[cols].applymap(func = len)
```


데이터프레임 정렬

- 시리즈 또는 데이터프레임을 오름차순 또는 내림차순 정렬합니다.

```
>>> price['층'].sort_values() # 층을 오름차순 정렬합니다.  
[참고] ascending 매개변수에 전달하는 인수의 기본값은 True입니다.
```

```
>>> price['층'].sort_values(ascending = False) # 층을 내림차순 정렬합니다.
```

```
>>> price.sort_values(by = ['층']) # price를 층으로 오름차순 정렬합니다.
```

```
>>> price.sort_values(by = ['층'], ascending = False) # price를 층으로 내림차순 정렬  
합니다.
```

- 데이터프레임 정렬 기준 열이 2개 이상일 때 열별로 방향을 설정합니다.

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

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

집계함수로 데이터 요약

- 집계함수로 시리즈의 기술통계량을 확인합니다.

```
>>> price['거래금액'].count() # 거래금액에서 결측값을 제외한 빈도수를 반환합니다.
```

```
>>> price['거래금액'].mean() # 거래금액의 평균을 반환합니다.
```

```
>>> price['거래금액'].std() # 거래금액의 표준편차를 반환합니다.
```

```
>>> price['거래금액'].min() # 거래금액의 최솟값을 반환합니다.
```

```
>>> price['거래금액'].max() # 거래금액의 최댓값을 반환합니다.
```

```
>>> price['거래금액'].describe() # 거래금액의 다양한 기술통계량을 반환합니다.
```

범주별 집계함수로 데이터 요약

- 범주형 변수로 그룹을 묶고 집계함수로 요약합니다.

```
>>> price.groupby(by = ['자치구']).count()['거래금액'] # 자치구별 거래금액에서 결측값을  
# 제외한 빈도수를 반환합니다.
```

```
>>> price.groupby(by = ['자치구']).mean()['거래금액'] # 자치구별 거래금액의 평균을 반환  
# 합니다.
```

```
>>> price.groupby(by = ['자치구']).std()['거래금액'] # 자치구별 거래금액의 표준편차를  
# 반환합니다.
```

```
>>> price.groupby(by = ['자치구']).min()['거래금액'] # 자치구별 거래금액의 최솟값을 반환  
# 합니다.
```

```
>>> price.groupby(by = ['자치구']).max()['거래금액'] # 자치구별 거래금액의 최댓값을 반환  
# 합니다.
```

```
>>> price.groupby(by = ['자치구']).describe()['거래금액'] # 자치구별 거래금액의 다양한  
# 기술통계량을 반환합니다.
```

[참고] agg() 함수 사용법

- 데이터프레임의 열별 집계 결과를 시리즈로 반환합니다.

```
>>> price.agg(func = 'mean', numeric_only = True)
```

- 데이터프레임에서 특정 열의 집계 결과에 행이름을 지정할 수 있습니다.

```
>>> price.agg(건수 = ('아파트', 'count'), 평균 = ('거래금액', 'mean'))
```

- 범주형 변수로 그룹을 묶을 수 있습니다.

```
>>> price.groupby(by = ['자치구']) \ # [참고] 여러 줄로 작성하려면 마지막에 \ 기호를 추가합니다.  
                                     [주의] \ 기호 뒤에 공백 포함 아무것도 입력하면 안됩니다!  
    .agg(건수 = ('아파트', 'count'), 평균 = ('거래금액', 'mean')) \  
    .sort_values(by = ['평균'], ascending = False)
```

[참고] 범주형 변수의 빈도수/상대도수 확인

- 범주형 변수의 빈도수를 확인합니다.

```
>>> price['자치구'].value_counts() # 자치구별 빈도수를 내림차순 정렬한 결과를 반환합니다.
```

```
>>> price['자치구'].value_counts(ascending = True) # 자치구별 빈도수를 오름차순 정렬한 결과를 반환합니다.
```

```
>>> price['자치구'].value_counts().sort_index() # 자치구별 빈도수를 시리즈 인덱스로 오름차순 정렬한 결과를 반환합니다.
```

- 범주형 변수의 상대도수를 확인합니다.

```
>>> price['자치구'].value_counts(normalize = True) # 자치구별 상대도수를 내림차순 정렬한 결과를 반환합니다.
```

```
>>> price['자치구'].value_counts(normalize = True).round(4) * 100  
# 소수점 넷째 자리까지 남도록 반올림한  
결과에 100을 곱하면 백분율이 됩니다.
```

데이터프레임의 2가지 형태

자치구	금액구분	매매건수
강남구	5천 미만	806
강남구	5천 이상	1717
강동구	5천 미만	1103
강동구	5천 이상	935
강북구	5천 미만	921
강북구	5천 이상	9
⋮	⋮	⋮

[Long type]

df.pivot()
→
←
df.melt()

자치구	5천 미만	5천 이상
강남구	806	1717
강동구	1103	935
강북구	921	9
⋮	⋮	⋮

[Wide type]

데이터프레임의 2가지 형태

- 데이터프레임을 형태에 따라 Long type과 Wide type으로 구분합니다.
 - 데이터 분석 과정에서 Wide type을 주로 사용합니다.
 - 하지만 데이터를 요약하거나 준비된 데이터셋이 Long type인 경우가 있습니다.
 - 그래프를 그릴 때 Long type으로 변환해야 할 필요도 있습니다.
 - 따라서 Wide type과 Long type을 상호 변환하는 방법을 알고 있어야 합니다.
- 데이터프레임의 형태를 변환할 때 **pivot()** 또는 **melt()** 함수를 사용합니다.
 - **pivot()** 함수는 Long type을 Wide type으로 변환합니다.
 - **melt()** 함수는 Wide type을 Long type으로 변환합니다.

Long type 데이터프레임 생성

- 두 범주형 변수의 빈도수로 Long type 데이터프레임을 생성합니다.

```
>>> cols = ['자치구', '금액구분'] # 두 범주형 변수명으로 리스트를 생성합니다.
```

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

```
>>> elong.head() # [참고] elong은 자치구와 금액구분을 인덱스(행이름)로 갖는 데이터프레임입니다.  
                25개 자치구마다 2종의 금액구분이 있으므로 elong의 행 개수는 50입니다.
```

- elong의 행이름을 초기화하고 기존 행이름을 열로 추가합니다.

```
>>> elong = elong.reset_index()
```

```
>>> elong.head() # [참고] elong의 행이름을 초기화하면 50행, 3열인 데이터프레임으로 변환합니다.  
                마지막 열이름은 빈도수이므로 아래 코드를 실행하여 '매매건수'로 변경하는 것이 좋습니다.
```

```
>>> elong = elong.rename(columns = {'평당금액': '매매건수'})
```


Long type을 Wide type으로 변환

- Long type을 Wide type으로 변환합니다.

```
>>> widen = elong.pivot(index = '자치구', # index로 적용할 열이름을 지정합니다.
```

```
      columns = '금액구분', # Wide type의 열이름으로 적용할 Long type의  
                           # 열이름을 지정합니다.
```

```
      values = '매매건수') # Wide type의 값으로 채울 Long type의 열이름을  
                          # 지정합니다.
```

```
>>> widen.head() # [참고] widen은 25개 자치구별로 '5천 미만'과 '5천 이상'의 열을 갖습니다.(25행)
```

- widen의 행이름을 출력합니다.

```
>>> widen.index # [참고] 인덱스(행이름)에 name 속성이 있고, 속성값은 '자치구'입니다.
```

```
>>> widen.index.name # 인덱스(행이름)의 name을 출력합니다.  
                    # [참고] widen의 행이름을 초기화하면 인덱스 name을 열이름으로 설정합니다.
```

widen 행이름 초기화

- widen의 행이름을 초기화하고 기존 행이름을 열로 추가합니다.

```
>>> widen = widen.reset_index() # widen의 행이름을 열에 추가하도록 drop = True를 생략합니다.  
                                [참고] 행이름의 name 속성값인 '자치구'를 열이름으로 설정합니다.
```

```
>>> widen.head() # widen의 처음 5행을 출력합니다.  
                [참고] 행이름 위에 '금액구분'을 출력하는데, 이것은 열이름의 name입니다.
```

- widen의 열이름을 출력하고, name 속성값을 삭제합니다.

```
>>> widen.columns # widen의 열이름을 출력합니다.  
                [참고] 컬럼명(열이름)에 name 속성이 있고, 속성값은 '금액구분'입니다.
```

```
>>> widen.columns.name # 컬럼명(열이름)의 name을 출력합니다.
```

```
>>> widen.columns.name = '' # 컬럼명(열이름)의 name에 빈 문자열을 할당합니다.
```

```
>>> widen.head() # widen의 처음 5행을 출력합니다.  
                [참고] 행이름 위에 아무것도 출력하지 않습니다.
```

Wide type을 Long type으로 변환

- Wide type을 Long type으로 변환하고 오름차순 정렬 및 인덱스를 초기화합니다.

```
>>> widen.melt(id_vars = '자치구', # id_vars 매개변수에 맨 왼쪽에 놓을 열 이름을 지정하고,  
               value_vars = ['5천 미만', '5천 이상'], # value_vars 매개변수에는 세로로 늘일 열 이름을 지정합니다.  
               var_name = '금액종류', # value_vars에 지정한 열 이름을 원소로 갖는 새  
                                     열 이름을 지정합니다.  
               value_name = '거래건수') \ # value_vars 매개변수에 지정한 열의 값을 원소로 갖는  
                                         새 열 이름을 지정합니다.  
      .sort_values(by = ['자치구', '금액종류']) \  
      .reset_index(drop = True) \  
      .head()
```

피벗 테이블 생성

- 두 범주형 변수로 연속형 변수를 요약한 피벗 테이블을 생성합니다.

```
>>> pd.pivot_table(data = price, # 데이터프레임을 지정합니다.  
                   [주의] data 매개변수를 생략하면 에러가 발생합니다.  
  
                   values = '평당금액', # 집계함수에 적용할 연속형 변수의 열이름을 지정합니다.  
  
                   index = '자치구', # 행이름에 적용할 열이름을 지정합니다.  
  
                   columns = '금액구분', # 열이름에 적용할 열이름을 지정합니다.  
  
                   aggfunc = np.mean) # 집계함수를 np.통계함수 또는 문자열로 지정합니다.  
                                     [참고] 함수가 두 개 이상이면 리스트로 지정합니다.
```

교차 테이블 생성

- 두 범주형 변수의 빈도수/상대도수를 원소로 갖는 교차 테이블을 생성합니다.

```
>>> pd.crosstab(index = price['자치구'], # 행이름에 적용할 시리즈를 지정합니다.
                 [참고] 입력변수(원인)를 지정합니다.

                 columns = price['금액구분'], # 열이름에 적용할 시리즈를 지정합니다.
                 [참고] 목표변수(결과)를 지정합니다.

                 normalize = 'index', # normalize 매개변수를 추가하면 상대도수를 반환합니다.
                 [참고] 'index'는 행별, 'columns'는 열별 상대도수입니다.

                 margins = True, # 행과 열 합계를 추가합니다.

                 margins_name = '합계') # 행과 열 합계의 이름을 지정합니다. (기본값: 'All')
```

데이터프레임 결합

- 두 데이터프레임의 열이름이 순서까지 정확하게 같은지 확인합니다.

```
>>> df1.columns.equals(other = df2.columns) # 두 데이터프레임의 열이름이 같으므로 True입니다.  
                                             [참고] 순서만 달라도 False를 반환합니다.
```

- 열이름이 같은 두 데이터프레임을 행(세로) 방향으로 결합합니다.

```
>>> pd.concat(objs = [df1, df2]) # [참고] 두 데이터프레임의 기존 행이름을 유지합니다.
```

```
>>> pd.concat(objs = [df1, df2], ignore_index = True) # 두 데이터프레임을 세로로 결합하고  
                                                       행이름을 초기화합니다.
```

- df2의 일부 열이름을 변경합니다.

```
>>> df2 = df2.rename(columns = {'아파트': '아파트명'})
```

```
>>> df1.columns.equals(other = df2.columns) # 두 데이터프레임 열이름이 서로 다르므로 False를  
                                             반환합니다.
```

데이터프레임 결합(계속)

- 열이름이 다른 두 데이터프레임을 행(세로) 방향으로 결합합니다.

```
>>> pd.concat(objs = [df1, df2], ignore_index = True) # [참고] 열이름이 서로 다른 셀 값을  
결측값으로 채웁니다.
```

- 두 데이터프레임을 열(가로) 방향으로 결합합니다.

```
>>> pd.concat(objs = [df1, df2], axis = 1) # [참고] 행이름이 같은 행을 결합하고 행이름이 서로  
다른 셀 값은 결측값으로 채웁니다.
```

- df2의 행이름을 초기화합니다.

```
>>> df2 = df2.reset_index(drop = True)
```

- 행이름을 초기화한 두 데이터프레임을 열(가로) 방향으로 결합합니다.

```
>>> pd.concat(objs = [df1, df2], axis = 1) # 행이름이 같은 행에는 결측값이 없고 행이름이 서로  
다른 행에는 결측값이 있습니다.
```

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 열(가로) 방향으로 병합합니다.

데이터프레임을 병합하기 전에 두 가지를 확인해야 합니다.

- 두 외래키의 값이 서로 일치하는 행이 있는가?
- 오른쪽 외래키에 중복 원소가 있는가? (1:1, M:1, M:N 관계)

내부 병합 Inner Join

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

외부 병합 Full 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 병합 Left Outer Join

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

병합 데이터셋 준비

- 현재 작업 경로에 있는 폴더명과 파일명을 출력합니다.

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

- 상세정보 xlsx 파일을 읽고 데이터프레임을 생성합니다.

```
>>> detail = pd.read_excel(io = 'Naver_APT_Detail_Seoul.xlsx')
```

- detail의 정보를 확인합니다.

```
>>> detail.info()
```

- detail의 처음 5행을 출력합니다.

```
>>> detail.head()
```

외래키 확인 및 전처리

- 두 데이터프레임의 외래키에서 일치하는 원소 개수를 확인합니다.

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

- 두 데이터프레임의 외래키를 각각 출력합니다.

```
>>> price['주소'].head() # price의 주소를 출력합니다. '서울특별시'로 시작합니다.
```

```
>>> detail['지번주소'].head() # detail의 지번주소를 출력합니다. '서울시'로 시작합니다.
```

- price의 주소에서 '특별'을 삭제하고, 일치하는 원소 개수를 다시 확인합니다.

```
>>> price['주소'] = price['주소'].str.replace(pat = '특별', repl = '')
```

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

[참고] 표본 추출 및 시드 고정

- 모집단(1~45의 정수)에서 6개 표본을 비복원추출하는 코드를 10번 반복합니다.

```
>>> for i in range(10):
```

```
    np.random.seed(seed = 1) # 시드는 임의 값을 생성하기 전에 설정하는 초깃값입니다.  
                             # 시드에 같은 값을 지정하면 항상 재현 가능한 결과를 얻습니다.
```

```
    lotto = np.random.choice(a = range(1, 46), size = 6, replace = False)
```

```
    lotto.sort() # lotto의 원소를 오름차순 정렬합니다.
```

```
    print(lotto)
```

[참고] 중복 원소 확인 함수

- 1~5의 정수를 복원추출하여 시리즈를 생성합니다.

```
>>> np.random.seed(seed = 2) # 시드를 고정합니다.
```

```
>>> nums = np.random.choice(a = range(1, 6), size = 3) # 1~5의 정수에서 3개를 복원추출합니다.
```

```
>>> nums = pd.Series(data = nums); nums # nums를 시리즈로 변환합니다.
```

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

```
>>> nums.duplicated() # 순방향으로 원소 중복 여부를 True/False로 반환합니다.  
[참고] keep 매개변수에 중복일 때 선택할 순서를 지정합니다.(기본값: 'first')
```

```
>>> nums.duplicated(keep = 'last') # 역방향으로 원소 중복 여부를 True/False로 반환합니다.
```

```
>>> nums.duplicated(keep = False) # 모든 중복 원소를 True로 반환합니다.(탐색 방향과 상관 없음)
```

데이터프레임 중복 원소 확인 및 제거

- detail의 지번주소가 중복이면 True, 아니면 False인 시리즈를 생성합니다.

```
>>> dup = detail['지번주소'].duplicated(keep = False)
```

- detail에서 dup이 True인 행을 선택하고 지번주소로 오름차순 정렬합니다.

```
>>> detail[dup].sort_values(by = ['지번주소']) # 실제 업무에서는 중복 발생 원인을 확인하고  
# 데이터를 전처리해야 합니다.
```

- detail의 지번주소에서 순방향으로 중복인 행을 제거하고 detail에 재할당합니다.

```
>>> detail = detail[~ detail['지번주소'].duplicated()] # [참고] ~ 연산자는 진리값을 반전  
# 합니다.
```

- detail의 행 개수를 확인합니다.

```
>>> detail.shape[0] # 행 개수가 감소했습니다.(9668 → 9640)
```

데이터프레임 병합

- price와 detail에서 일치하는 열이름을 확인합니다.

```
>>> set(price.columns) & set(detail.columns) # 두 데이터프레임에서 일치하는 열이름이 없으면  
# 병합할 때 외래키 이름을 각각 지정해야 합니다.
```

- 두 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', # 왼쪽 데이터프레임의 외래키 이름을 지정합니다.
```

```
right_on = '지번주소') # 오른쪽 데이터프레임의 외래키 이름을 지정합니다.
```

데이터프레임 병합(계속)

- detail의 외래키 이름을 '주소'로 변경합니다.

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

- price와 detail에서 일치하는 열이름을 확인합니다.

```
>>> set(price.columns) & set(detail.columns) # 두 데이터프레임에서 '주소'만 일치합니다.
```

- 외래키 이름이 같으면 on 매개변수를 사용합니다. # [참고] 외래키 이름이 모두 같으면 생략할 수 있습니다.

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

- apt의 정보를 확인합니다.

```
>>> apt.info() # apt는 41264행 23열인 데이터프레임입니다.
```

[참고] 외래키 설정

- 두 데이터프레임의 외래키가 2개 이상이고 서로 다르면(예를 들어 df1과 df2에서 외래키 관계가 x1은 y1과 같고, x2는 y2와 같은 경우) 매칭 순서에 맞게 리스트로 지정합니다.

```
>>> pd.merge(left = df1, right = df2, how = 'inner',  
              left_on = ['x1', 'x2'], right_on = ['y1', 'y2']) # [주의] 외래키 순서를  
                                                                매칭해서 지정합니다.
```

- 두 데이터프레임의 외래키가 2개 이상이고 모두 같으면(예를 들어 df1과 df2에서 외래키가 x1과 x2인 경우) 외래키를 리스트로 지정하거나 생략합니다.

```
>>> pd.merge(left = df1, right = df2, how = 'inner', on = ['x1', 'x2'])
```


외부 파일로 저장

- apt를 xlsx 파일로 저장합니다. *# xlsx 파일을 저장할 때 가장 오래 걸립니다.*

```
>>> apt.to_excel(excel_writer = 'APT_List_Seoul_2021.xlsx', index = None)
```

- apt를 csv 파일로 저장합니다.

```
>>> apt.to_csv(path_or_buf = 'APT_List_Seoul_2021.csv', index = None)
```

- apt를 z 파일로 저장합니다.

```
>>> joblib.dump(value = apt, filename = 'APT_List_Seoul_2021.z')
```

데이터 시각화

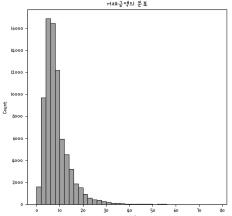
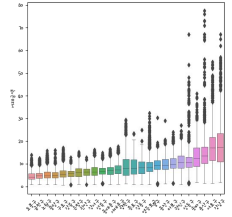
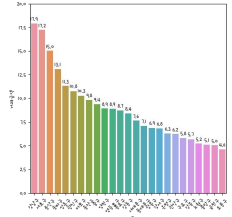
데이터 시각화

- 일변량 데이터의 분포 또는 이변량 데이터의 관계를 그래프로 확인함으로써 분석 데이터셋에 대한 이해의 폭을 넓힐 수 있습니다.
- 데이터 분석 결과를 시각적으로 표현함으로써 데이터 분석 과정에 참여하지 않은 사람들에게 분석 결과를 쉽고 빠르게 전달할 수 있습니다.
 - 2차원의 정적인 이미지로 표현하는 것이 일반적이지만 3차원 입체 또는 동적인 이미지로 전환하면 더욱 효과적입니다.
- 데이터 시각화는 그래프로 데이터에 잠재된 패턴을 발굴하고 분석 결과를 쉽고 빠르게 전달하는 것을 목적으로 수행하는 분석 방법입니다.

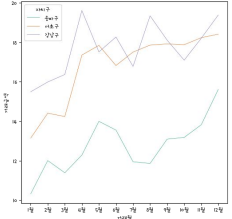
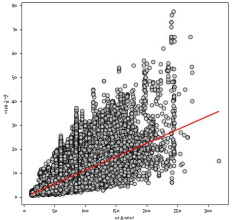


출처: Data Squirrel(youtube)

데이터 시각화 종류

구분	예시	특징
히스토그램		<ul style="list-style-type: none"> • 히스토그램은 일변량 연속형 변수의 도수분포표를 시각화한 그래프입니다. • 세로축은 빈도수이며 밀도로 변경할 수 있습니다. • 히스토그램의 막대는 서로 붙어 있으며, 세로축을 밀도로 변경하면 막대의 총면적은 확률 1을 의미합니다.
상자 수염 그림		<ul style="list-style-type: none"> • 일변량 연속형 변수의 분포에 사분위수와 이상치를 추가한 그래프입니다. • 가로축에 범주형 변수를 지정하면 집단 간 연속형 변수의 분포를 비교할 수 있습니다.
막대 그래프		<ul style="list-style-type: none"> • 일변량 막대 그래프는 범주형 변수의 빈도수를 막대로 그린 그래프입니다. • 이변량 막대 그래프는 범주형 변수에 따라 연속형 변수의 크기를 비교할 수 있습니다.

데이터 시각화 종류(계속)

구분	예시	특징
선 그래프		<ul style="list-style-type: none"> • 선 그래프는 시간에 따라 연속형 변수의 변화를 표현한 그래프입니다. • 주가 데이터와 같이 시계열 변수를 시각화할 때 사용합니다.
산점도		<ul style="list-style-type: none"> • 산점도는 이변량 연속형 변수의 선형관계를 2차원 평면에 점으로 표현한 그래프입니다. • 선형 회귀분석의 입력변수와 목표변수에 직선의 관계가 존재하는지 확인할 수 있습니다.

관련 라이브러리 호출

- 관련 라이브러리를 호출합니다.

```
>>> import os
```

```
>>> import chardet
```

```
>>> import numpy as np
```

```
>>> import pandas as pd
```

```
>>> import joblib
```

작업 경로 확인 및 변경

- 현재 작업 경로를 확인합니다.

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

- data 폴더로 작업 경로를 변경합니다.

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

- 현재 작업 경로에 있는 폴더명과 파일명을 출력합니다.

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

실습 데이터셋 준비

- z 파일을 호출하고 데이터프레임 apt에 할당합니다.

```
>>> apt = joblib.load(filename = 'APT_List_Seoul_2021.z')
```

- apt의 정보를 확인합니다.

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

- apt의 처음 5행을 출력합니다.

```
>>> apt.head()
```

- apt의 열이름을 출력합니다.

```
>>> apt.columns
```


실습 데이터셋 전처리

- apt에서 필요 없는 열을 삭제합니다.

```
>>> apt = apt.drop(columns = ['주소', '아파트ID', '아파트명'])
```

- apt에 거래월을 추가합니다.

```
>>> apt['거래월'] = apt['거래일'].dt.strftime(date_format = '%m월')
```

- 거래월 원소의 맨 앞에 있는 '0'을 삭제합니다. # 거래일 원소(날짜 데이터)에서 '01월', '02월' 형태의 문자열을 생성합니다.

```
>>> apt['거래월'] = apt['거래월'].str.replace(pat = '^0', repl = '')
```

- apt의 처음 5행을 출력합니다. # [참고] 정규표현식의 '^' 기호는 문자열 시작 위치를 지정합니다.

```
>>> apt.head()
```

시각화 설정: 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.
```

- 테스트용 그래프를 그립니다.

```
>>> sns.histplot(data = apt, x = '거래금액')
```

```
>>> plt.title(label = '아파트 거래금액 분포'); # [참고] 코드 마지막에 추가한 세미콜론(;)은  
plt.show() 함수와 같은 기능을 실행합니다.
```

위 코드를 실행하면 한글을 네모로 출력하므로 한글폰트를 설정해야 합니다.

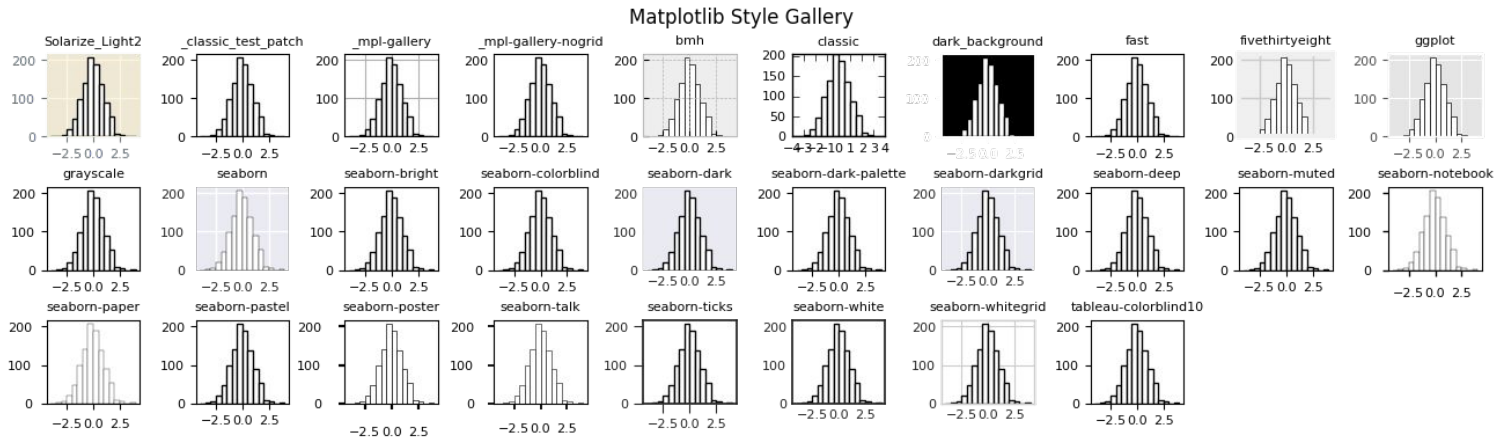
[참고] 한글폰트 외 그래프 크기 및 해상도 등 다양한 그래픽 파라미터를 설정할 수 있습니다.

시각화 설정: 스타일시트

- matplotlib 스타일시트를 설정합니다.

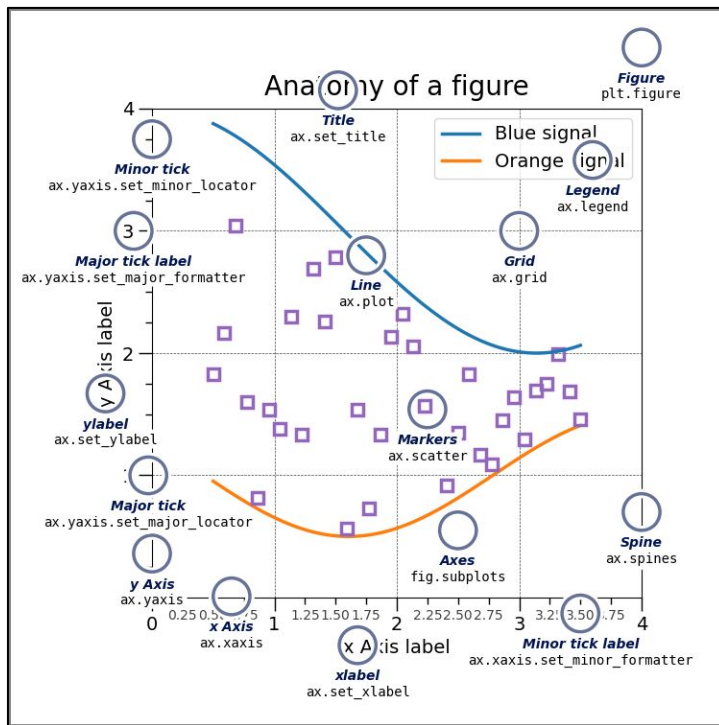
>>> plt.style.available # matplotlib 라이브러리에서 사용할 수 있는 스타일시트 목록을 확인합니다.
[참고] 스타일 갤러리 웹페이지를 참고하세요.

>>> plt.style.use(style = 'seaborn-white') # 그래프에 적용할 스타일시트를 지정합니다.



참고: http://tonysyu.github.io/raw_content/matplotlib-style-gallery/gallery.html

[참고] Anatomy of a figure



출처: <https://matplotlib.org/stable/gallery/showcase/anatomy.html>

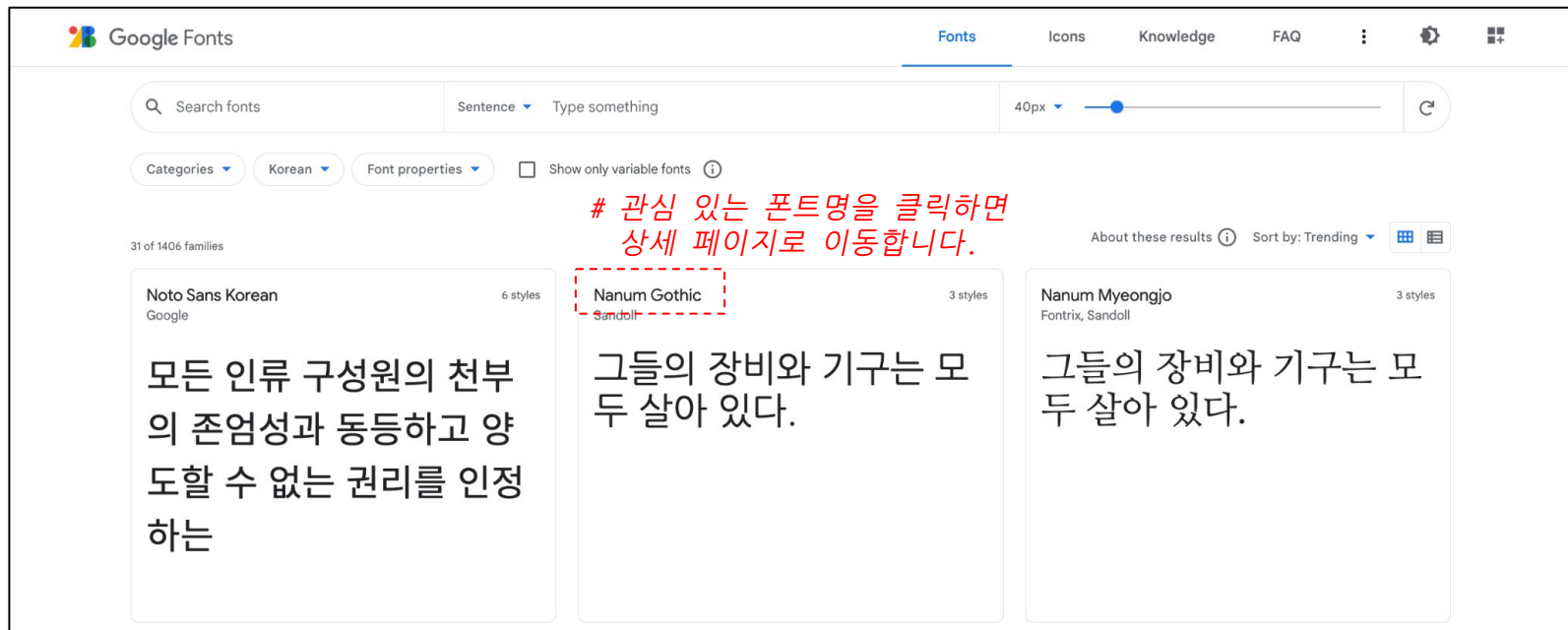
[참고] matplotlib.pyplot.rc Property Alias

- 그래픽 파라미터에서 일부 속성^{property}은 가명^{alias}으로 대체할 수 있습니다.

속성	가명	상세 내용
linewidth	lw	선의 두께를 설정합니다.
linestyle	ls	선의 종류를 설정합니다.
color	c	막대 또는 점의 채우기 색을 설정합니다.(산점도)
facecolor	fc	점의 채우기 색을 설정합니다.(상자 수염 그림 이상치)
edgecolor	ec	점의 테두리 색을 설정합니다.(산점도)
markeredgewidth	mew	점의 테두리 색을 설정합니다.(상자 수염 그림 이상치)
antialiased	aa	픽셀 이미지의 각을 부드럽게 보완합니다.

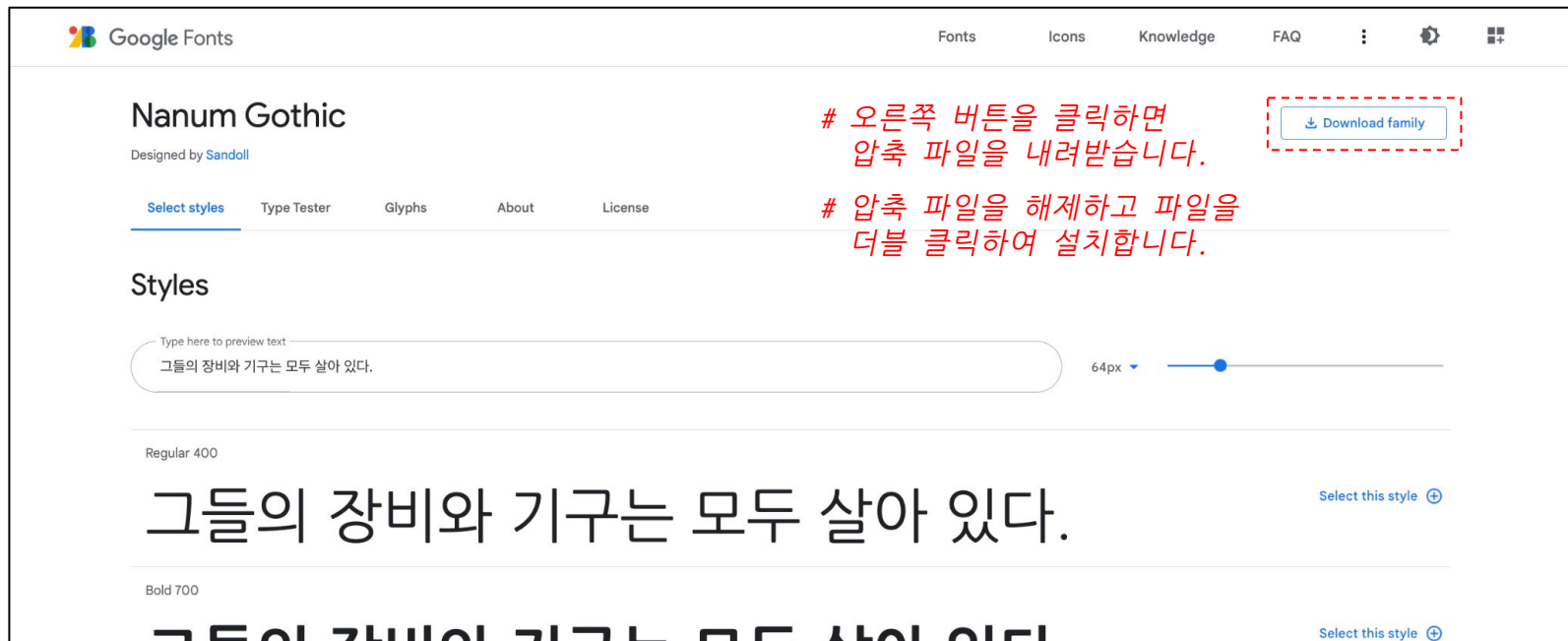
[참고] 한글폰트 설치

- 구글 폰트(<https://fonts.google.com/?subset=korean>)에서 한글폰트를 탐색합니다.



[참고] 한글폰트 설치(계속)

- 상세 페이지에서 오른쪽 위에 있는 **Download family** 버튼을 클릭합니다.



시각화 설정: 한글폰트명 탐색

- 현재 사용 중인 컴퓨터에 설치한 전체 폰트 파일명을 리스트로 반환합니다.

```
>>> fontList = fm.findSystemFonts(fonttext = 'ttf')
```

- 특정 문자열을 갖는 파일명으로 리스트를 생성합니다.

```
>>> fontPath = [font for font in fontList if 'Gamja' in font]
```

- 반복문으로 폰트명을 출력합니다.

```
>>> for font in fontPath:
```

```
    print(fm.FontProperties(fname = font).get_name())
```

반복문을 실행한 결과에서 마음에 드는 폰트명을 선택하고 **plt.rc()** 함수에 지정합니다.
[참고] rc는 *runtime configuration*를 의미하며, *pyplot*을 실행하는 환경을 의미합니다.

시각화 설정: 그래픽 파라미터 설정

- 그래프 크기와 해상도를 설정합니다.

```
>>> plt.rc(group = 'figure', figsize = (8, 4), dpi = 150)
```

- 한글폰트와 글자 크기를 설정합니다.

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

- 축에 유니코드 마이너스를 출력하지 않도록 설정합니다.

```
>>> plt.rc(group = 'axes', unicode_minus = False) # [참고] 왼쪽 코드를 설정하지 않으면  
                                                    음수 앞에 '␣'를 출력합니다.
```

- 범례에 채우기 색과 테두리 색을 추가합니다. # *fc*는 *facecolor*(채우기), *ec*는 *edgecolor*(테두리)
관련 매개변수이고 '0'은 검정, '1'은 흰색입니다.

```
>>> plt.rc(group = 'legend', frameon = True, fc = '1', ec = '0')
```

[참고] 한글을 네모로 출력하는 문제 해결 방법

- 한글폰트를 설정했음에도 한글을 네모로 출력하는 에러가 발생할 수 있습니다.
 - 에러 메시지: Font family ['폰트명'] not found. Falling back to DejaVu Sans
 - matplotlib 임시 폴더에 저장된 json 파일에 해당 한글폰트가 없기 때문입니다.
- json 파일을 삭제하고, Jupyter Notebook을 재실행하면 해결할 수 있습니다.

```
>>> import matplotlib, glob # 관련 라이브러리를 호출합니다.
                                [참고] glob.glob() 함수는 조건에 맞는 파일명을 리스트로 반환합니다.

>>> path = matplotlib.get_cachedir() # matplotlib 라이브러리 임시 폴더 경로를 path에 할당합니다.

>>> fileName = glob.glob(f'{path}/fontlist-*.json')[0] # 폰트 정보를 갖는 json 파일명을
                                                         fileName에 할당합니다.

>>> os.remove(path = fileName) # matplotlib 라이브러리 임시 폴더에 있는 json 파일을 삭제합니다.
                                Jupyter Notebook을 재실행하면 한글폰트를 설정할 수 있습니다.
```

[참고] 그래픽 파라미터 설정 관련 모듈 생성

- 시각화 라이브러리 호출, 스타일시트, 한글폰트 및 그래픽 파라미터를 설정하는 코드를 모듈(py 파일)로 저장하면 필요할 때마다 호출할 수 있으므로 편리합니다.
- Anaconda 메인에서 Text File을 열고, 모듈(py 파일)로 저장할 시각화 설정 관련 코드를 붙여넣습니다.
- 상단 메뉴에서 File → Save Text As를 클릭하고 텍스트 파일을 저장합니다.
 - 파일명을 GraphicSetting.py로 저장합니다. *# [주의] py 파일을 Jupyter Notebook 파일과 같은 폴더에 저장해야 호출할 수 있습니다.*
- 시각화 설정 모듈을 호출합니다.

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

[참고] Python 파일 탐색 경로 확인

- 모듈(py 파일)을 호출하려면 아래 두 가지 조건 중 하나를 만족해야 합니다.
 - 현재 작업 중인 Jupyter Notebook 파일 경로에 py 파일을 저장했다.
 - Python 파일 탐색 경로 중 한 곳에 py 파일을 저장했다.
- 작업할 Jupyter Notebook 파일 경로가 바뀔 때마다 py 파일을 매번 옮겨야 하므로 첫 번째 조건은 불편합니다. 따라서 두 번째 조건을 따르는 것이 좋습니다.
- Python 파일 탐색 경로를 확인합니다.

```
>>> import sys # 관련 라이브러리를 호출합니다.
```

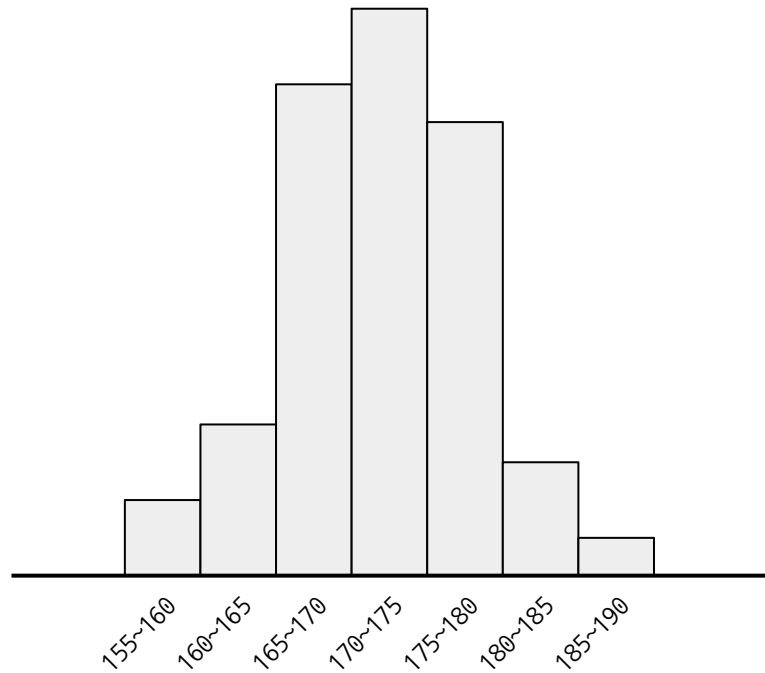
```
>>> sys.path # Python 파일 탐색 경로를 모두 출력합니다. 많은 경로 중 한 곳(마지막 경로 추천)에 py 파일을  
저장하면 해당 모듈을 항상 호출할 수 있습니다.
```

히스토그램

- 히스토그램은 일변량 연속형 변수의 도수분포표를 시각화한 것입니다.

계급	도수	상대도수
155 ~ 160	2	0.04
160 ~ 165	4	0.08
165 ~ 170	13	0.26
170 ~ 175	15	0.30
175 ~ 180	12	0.24
180 ~ 185	3	0.06
185 ~ 190	1	0.02
합계	50	1.00

도수분포표의 계급은 이상 ~ 미만입니다.



히스토그램 그리기

- 거래금액으로 히스토그램을 그립니다.

```
>>> sns.histplot(data = apt, x = '거래금액'); # data 매개변수에 데이터프레임을 지정합니다.  
# x 매개변수에 연속형 변수명을 지정합니다.
```

- 히스토그램에 그래픽 요소를 추가합니다.

```
>>> sns.histplot(data = apt, x = '거래금액', bins = 50, # bins 매개변수에 막대 개수를  
# 50으로 지정합니다.
```

```
color = '1', edgecolor = '0'); # color 매개변수에 채우기 색, edgecolor  
# 매개변수에 테두리 색을 지정합니다.
```

```
>>> sns.histplot(data = apt, x = '거래금액', binwidth = 2, # binwidth 매개변수에 막대  
# 너비를 2로 지정합니다.
```

```
color = '1', edgecolor = '0');
```

[참고] bins와 binwidth 매개변수로 히스토그램 막대 크기를 설정하면 최솟값부터 시작하므로 경계를 명확하게 구분할 수 없다는 단점이 있습니다. bins와 binrange 매개변수에 히스토그램 계급을 지정하면 단점을 보완할 수 있습니다.

히스토그램에 계급 추가

- 거래금액 최솟값과 최댓값을 확인합니다.

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

[참고] **describe()** 함수 실행 결과(시리즈)에서 일부 인덱스만 선택합니다.

- 히스토그램에 막대 개수와 범위(막대 경계)로 계급을 지정합니다.

```
>>> sns.histplot(data = apt, x = '거래금액', bins = 60, binrange = (0, 120),
                 color = '1', edgecolor = '0');
```

[주의] 히스토그램 계급은 최솟값보다 작거나 같은 값으로 시작하고 최댓값보다 크거나 같은 값으로 끝나야 합니다.

[참고] 색상 목록

- matplotlib에서 제공하는 CSS Color 목록을 확인하고 원하는 색을 고릅니다.

```
>>> import matplotlib.colors as mcl # 관련 모듈을 호출합니다.
```

```
>>> mcl.CSS4_COLORS # 148가지 색이름과 Hex Code를 딕셔너리로 출력합니다.  
[출처] https://matplotlib.org/stable/gallery/color/named\_colors.html
```

CSS Colors

black	bisque	forestgreen	slategrey	firebrick	khaki	darkslategray	darkorchid
dimgray	darkorange	limegreen	lightsteelblue	maroon	palegoldenrod	darkslategrey	darkviolet
dimgray	burlywood	darkgreen	cornflowerblue	darkred	darkkhaki	teal	mediumorchid
gray	antiquewhite	green	royalblue	red	ivory	darkcyan	thistle
gray	tan	lime	ghostwhite	mistyrose	beige	aqua	plum
darkgray	navajowhite	seagreen	lavender	salmon	lightyellow	cyan	violet
darkgray	blanchedalmond	mediumseagreen	midnightblue	tomato	lightgoldenrodyellow	darkturquoise	purple
silver	papayawhip	springgreen	navy	darksalmon	olive	cadetblue	darkmagenta
lightgray	moccasin	mintcream	darkblue	coral	yellow	powderblue	fuchsia
lightgray	orange	mediumspringgreen	mediumblue	orangered	olivedrab	lightblue	magenta
gainsboro	wheat	mediumaquamarine	blue	lightsalmon	yellowgreen	deepskyblue	orchid
whitesmoke	oldlace	aquamarine	slateblue	sienna	darkolivegreen	skyblue	mediumvioletred
white	floralwhite	turquoise	darkslateblue	seashell	greenyellow	lightskyblue	deeppink
snow	darkgoldenrod	lightseagreen	mediumslateblue	chocolate	chartreuse	steelblue	hotpink
rosybrown	goldenrod	mediumturquoise	mediumpurple	saddlebrown	lawngreen	aliceblue	lavenderblush
lightcoral	cornsilk	azure	rebeccapurple	sandybrown	honeydew	dodgerblue	palevioletred
indianred	gold	lightcyan	blueviolet	peachpuff	darkseagreen	lightslategray	crimson
brown	lemonchiffon	paleturquoise	indigo	peru	palegreen	lightslategray	pink
				linen	lightgreen	slategrey	lightpink

히스토그램 막대 채우기 색 변경

- 금액구분(범주형 변수)에 따라 채우기 색을 다르게 설정합니다.

```
>>> sns.histplot(data = apt, x = '거래금액', bins = 60, binrange = (0, 120),
                  hue = '금액구분', edgecolor = '0');
```

hue 매개변수에 지정한 범주형 변수의 범주별로 채우기 색을 다르게 설정합니다.

- 채우기 색 배합을 범주형 변수에 맞는 팔레트로 변경합니다.

```
>>> sns.histplot(data = apt, x = '거래금액', bins = 60, binrange = (0, 120),
                  hue = '금액구분', edgecolor = '0', palette = 'Set1');
```

[참고] 팔레트 탐색: Color Brewer

- Color Brewer는 미국의 지리학 교수인 Cynthia Brewer가 지도 제작을 위해 개발한 세 종류의 팔레트셋('sequential', 'diverging', 'qualitative')입니다.
- **seaborn** 라이브러리는 Color Brewer Palette를 탐색하는 함수를 제공합니다.

```
>>> sns.choose_colorbrewer_palette(data_type = 'qualitative');
```



[참고] 팔레트 설정

- 기본 팔레트 색을 출력합니다.

```
>>> sns.color_palette() # [참고] 기본 팔레트는 'tab10'이며, plt.cm의 한 종류입니다.
```

- Color Brewer에서 탐색한 팔레트 색을 출력합니다.

```
>>> sns.color_palette(palette = 'Set1', n_colors = 9) # [참고] n_colors 매개변수에 색의 개수를 변경할 수 있습니다.
```

- 기본 팔레트를 변경합니다.

```
>>> sns.set_palette(palette = 'Set1', n_colors = 9)
```

- 새로 설정한 기본 팔레트 색을 확인합니다.

```
>>> sns.color_palette()
```

[참고] 사용자 팔레트 생성

- 색이름을 원소로 갖는 리스트(사용자 팔레트)를 생성합니다.

```
>>> myPal = ['crimson', 'royalblue']
```

- 기존 그래프에 사용자 팔레트를 적용합니다.

```
>>> sns.histplot(data = apt, x = '거래금액', bins = 60, binrange = (0, 120),
                 hue = '금액구분', edgecolor = '0', palette = myPal,
                 hue_order = ['5천 미만', '5천 이상']));
```

hue_order 매개변수에 지정한 리스트 원소 순서대로 팔레트 색을 적용합니다.

히스토그램에 제목 및 축이름 추가

- 히스토그램에 제목, x축이름 및 y축이름을 추가합니다.

```
>>> sns.histplot(data = apt, x = '거래금액', bins = 60, binrange = (0, 120),
                  hue = '금액구분', edgecolor = '0', palette = myPal,
                  hue_order = ['5천 미만', '5천 이상'])

>>> plt.title(label = '거래금액의 분포') # 그래프 제목을 추가합니다.

>>> plt.xlabel(xlabel = '매매가격') # x축이름을 추가합니다.

>>> plt.ylabel(ylabel = '거래건수'); # y축이름을 추가합니다.
```

히스토그램에 커널 밀도 추정 곡선 추가

- 히스토그램의 y축을 빈도수^{count} 대신 밀도^{density}로 변경합니다.

```
>>> sns.histplot(data = apt, x = '거래금액', bins = 60, binrange = (0, 120),
                 color = '1', edgecolor = '0', # [참고] 커널 밀도 추정 곡선을 강조하기 위해
                                                다시 color 매개변수를 추가했습니다.
                 stat = 'density') # stat 매개변수의 기본 인수는 'count'입니다.
```

- 히스토그램에 커널 밀도 추정 곡선을 추가합니다.

```
>>> sns.kdeplot(data = apt, x = '거래금액', color = 'red',
                linewidth = 0.5, linestyle = '-');
```

[참고] 커널 밀도 추정은 비모수적인 방법으로 밀도를 추정하는 것으로, 개별 관측값을 중심으로 하는 커널 함수의 평균을 계산합니다. 대표적인 커널 함수로는 가우시안 함수(정규분포 확률밀도함수)가 있습니다.

[참고] 선의 종류

- 선의 종류에는 다음과 같은 4가지를 주로 사용합니다.

구분	이름	기호	모양
실선	solid	'_'	
파선	dashed	'--'	
1점 쇄선	dashdot	'-.'	
점선	dotted	':'	

관심 있는 자치구 선택

- apt에서 관심 있는 자치구를 선택하고 sub에 할당합니다.

```
>>> sub = apt[apt['자치구'].str.contains(pat = '강[남동북]')]
```

- 자치구별 거래금액 평균(분포의 중심)을 확인합니다.

```
>>> sub.groupby(by = ['자치구']).mean()['거래금액']
```

- 거래금액 최솟값과 최댓값을 확인합니다.

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


히스토그램을 겹쳐서 그리기

- 자치구별 히스토그램을 겹쳐서 그립니다.

```
>>> sns.histplot(data = sub, x = '거래금액', bins = 60, binrange = (0, 120),
hue = '자치구'); # [참고] palette 매개변수를 생략하면 기본 팔레트를 적용합니다.
                 현재 기본 팔레트는 'Set1'입니다.
```

- 커널 밀도 추정 곡선을 겹쳐서 그리는 것이 더 낫습니다.

```
>>> sns.kdeplot(data = sub, x = '거래금액', hue = '자치구',
shade = True); # shade 매개변수에 True를 지정하면 커널 밀도 추정 곡선 아래를 기본
                팔레트 색으로 채웁니다.
```

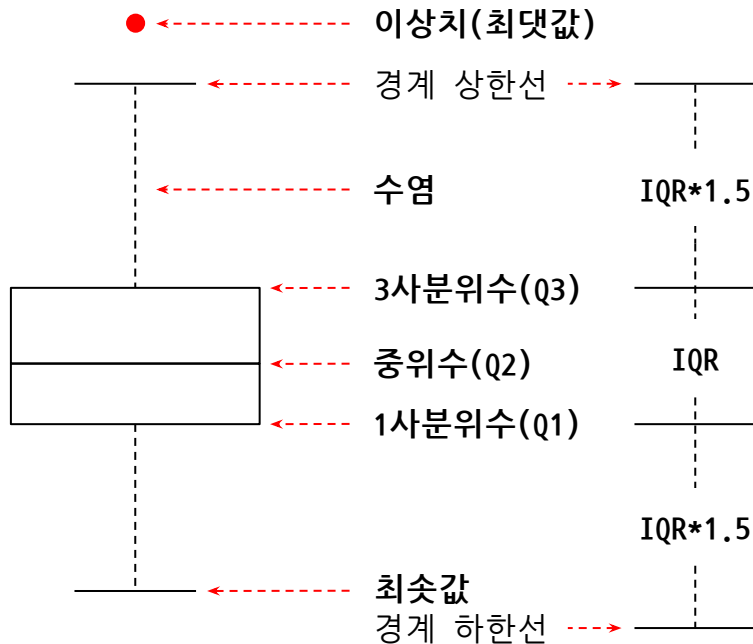
히스토그램을 나눠서 그리기

- 자치구별 히스토그램을 열(가로) 방향으로 나눠서 그립니다.

```
>>> sns.displot(data = sub, x = '거래금액', bins = 60, binrange = (0, 120),
                hue = '자치구',
                col = '자치구', # 자치구별 히스토그램을 열(column) 방향으로 출력합니다.
                               [참고] row 매개변수를 사용하면 행(row) 방향으로 출력합니다.
                legend = False, # 범례를 추가하지 않습니다.(기본값: True)
                height = 3, # 히스토그램 높이를 지정합니다.(기본값: 5)
                aspect = 0.8); # 히스토그램 폭을 지정합니다.(기본값: 1)
```

상자 수염 그림

- 상자 수염 그림은 연속형 변수의 분포에 사분위수와 이상치를 시각화한 것입니다.
 - 상자 수염 그림은 세로로 그립니다.
- 상자 수염 그림은 사분범위(Interquartile range)의 1.5배를 미달/초과하는 원소를 이상치로 판단합니다.
 - 사분범위는 3사분위수와 1사분위수의 간격이므로 전체 데이터의 가운데 50%를 포함하는 범위입니다.



일변량 상자 수염 그림 그리기

- 거래금액으로 상자 수염 그림을 그립니다.

```
>>> sns.boxplot(data = apt,
                y = '거래금액', # y 매개변수에 연속형 변수를 지정합니다.
                                [참고] x 매개변수를 사용하면 상자 수염 그림을 세워서 그립니다.
                color = '1', # 상자 채우기 색을 지정합니다.
                                [참고] 생략하면 기본 팔레트의 첫 번째 색으로 채웁니다.
                linewidth = 0.5);
```

일변량 상자 수염 그림 그리기

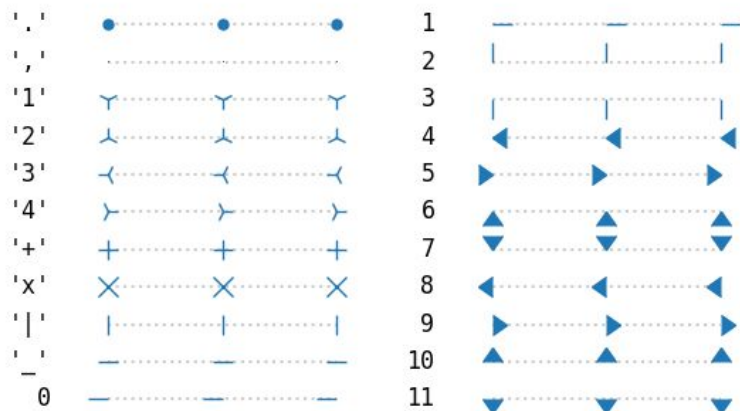
- 이상치 관련 속성으로 딕셔너리를 생성하고 상자 수염 그림에 추가합니다.

```
>>> outProps = {'marker': 'o', # 이상치 모양을 지정합니다.(기본값: 'd')
                'markersize': 3, # 이상치 크기를 지정합니다.(기본값: 5)
                'markerfacecolor': 'pink', # 이상치 채우기 색을 지정합니다.(기본값: 'black')
                'markeredgecolor': 'red', # 이상치 테두리 색을 지정합니다.(기본값: 'black')
                'markeredgewidth': 0.2} # 이상치 테두리 두께를 설정합니다.(기본값: 1)

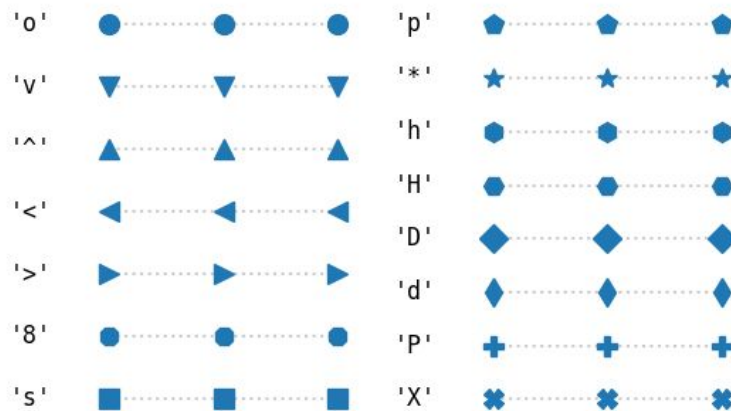
>>> sns.boxplot(data = apt, y = '거래금액', color = '1', linewidth = 0.5,
                flierprops = outProps);
```

[참고] marker의 종류

un-filled markers



filled markers



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

이변량 상자 수염 그림 그리기

- apt의 자치구별 거래금액 중위수를 오름차순 정렬한 grp를 생성합니다.

```
>>> grp = apt.groupby(by = ['자치구']).median()['거래금액']
```

```
>>> grp = grp.sort_values(); grp.head()
```

- x축에 자치구, y축에 거래금액을 지정하고 이변량 상자 수염 그림을 그립니다.

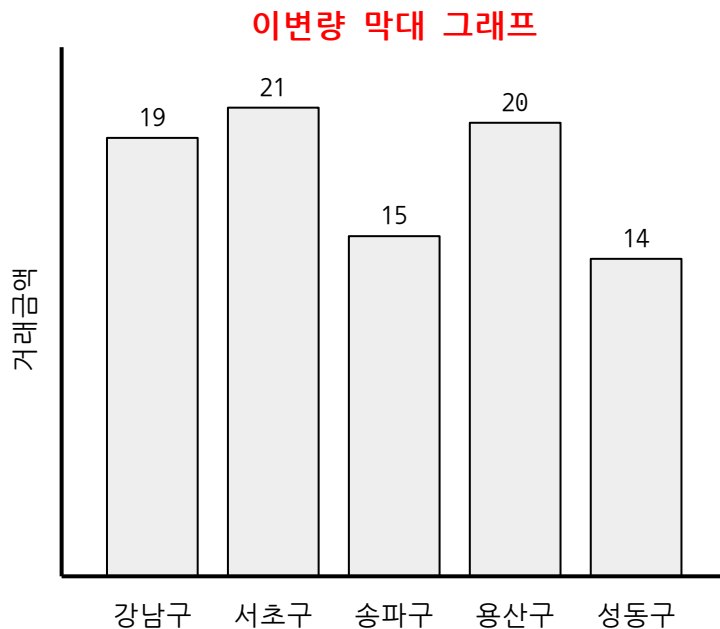
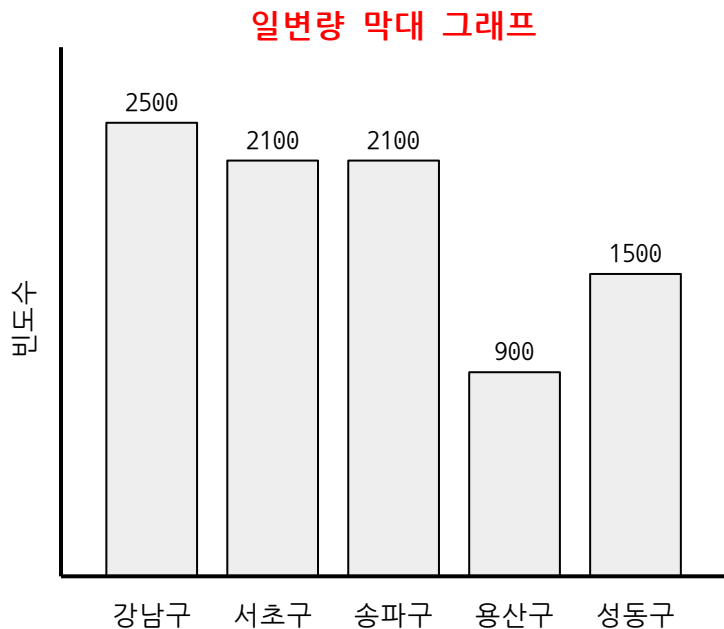
```
>>> sns.boxplot(data = apt, x = '자치구', y = '거래금액', linewidth = 0.5,
```

```
        flierprops = outProps, order = grp.index) # x축 순서를 grp 인덱스로  
                                                    지정합니다.
```

```
>>> plt.xticks(rotation = 45); # x축 눈금명을 45도 회전시킵니다.
```

막대 그래프

- 막대 그래프는 범주형 변수를 요약하여 시각화한 것입니다.



일변량 막대 그래프 그리기

- apt의 자치구별 거래금액 빈도수를 내림차순 정렬한 grp를 생성합니다.

```
>>> grp = apt.groupby(by = ['자치구']).count()['거래금액']
```

```
>>> grp = grp.sort_values(ascending = False); grp.head()
```

- 자치구별 빈도수로 일변량 막대 그래프를 그립니다.

```
>>> sns.countplot(data = apt, x = '자치구', order = grp.index)
```

```
>>> plt.ylim(0, 4000) # 막대 위에 텍스트를 출력할 공간을 확보하기 위해 y축 범위를 제한합니다.  
[주의] 데이터마다 빈도수 범위가 다르므로 미리 확인하고 설정합니다.
```

```
>>> plt.xticks(rotation = 45);
```

막대 위에 텍스트 추가

- 막대 위에 자치구별 빈도수를 텍스트로 추가합니다.

```
>>> for i, v in enumerate(grp): # enumerate() 함수는 객체의 인덱스와 원소 쌍을 튜플로 반환합니다.
```

```
    plt.text(x = i, y = v, s = v, # x, y 매개변수에 x, y 좌표를 지정합니다.  
            # s 매개변수에 문자열로 출력할 값을 지정합니다.
```

```
        ha = 'center', va = 'bottom', # ha 매개변수에 수평정렬 방식을 지정합니다.  
        # va 매개변수에 수직정렬 방식을 지정합니다.
```

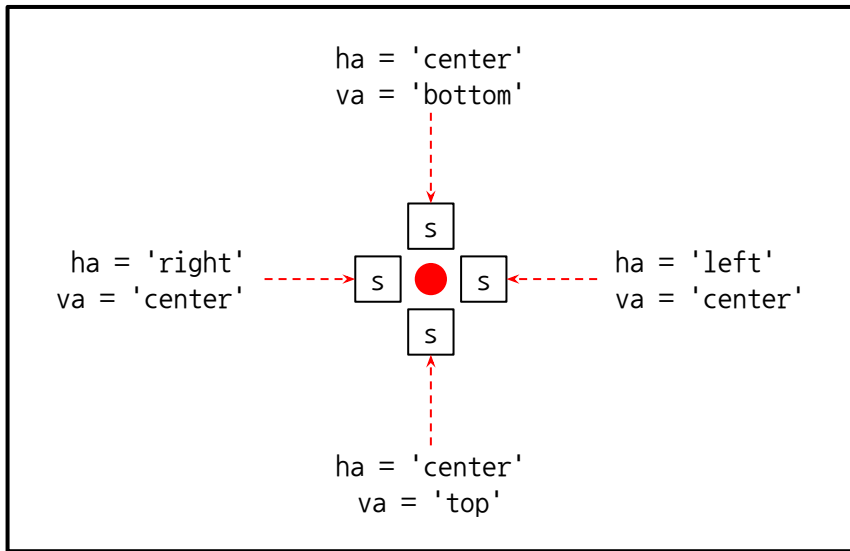
```
        color = 'black', fontsize = 9, # color 매개변수에 글자색을 지정합니다.  
        # fontsize 매개변수에 크기를 지정합니다.
```

```
        fontweight = 'bold') # fontweight 매개변수에 글자 굵기를 지정합니다.  
        # 'light', 'regular', 'medium', 'bold', 'extra bold' 등
```

[주의] 위 코드를 앞 페이지에서 작성한 막대 그래프 코드 셀에 추가해야 합니다.

[참고] plt.text() 함수

- `plt.text()` 함수는 지정한 위치(`x`, `y` 좌표와 `ha`, `va` 값)에 문자열을 출력합니다.
 - `x`, `y` 매개변수에 좌표를 지정합니다.
 - `x`, `y` 좌표를 빨간 점으로 가정합니다.
 - `s` 매개변수에 지정한 값을 문자열로 출력합니다.
 - `ha`, `va` 매개변수에 지정한 값에 따라 출력할 글자의 위치가 달라집니다.
 - 문자열 기준으로 빨간 점의 위치를 가로(`ha`)와 세로(`va`)로 조정합니다.



[참고] 파이 차트 그리기

- sub의 자치구별 빈도수를 내림차순 정렬한 grp를 생성합니다.

```
>>> grp = sub['자치구'].value_counts(); grp.head()
```

- grp로 파이 차트를 그립니다.

```
>>> plt.pie(x = grp.values, # 파이 차트의 썸(wedge) 크기를 자치구별 빈도수로 지정합니다.  
            [참고] 파이 차트는 썸의 크기를 수치의 백분율로 계산합니다.
```

```
    explode = [0, 0, 0.2], # 썸별 시작 위치(실수)를 원소로 갖는 리스트로 지정합니다.  
               [참고] 특정 썸을 강조하려면 0보다 큰 값을 지정합니다.
```

```
    labels = grp.index, # 썸별 라벨을 지정합니다.  
               [참고] 라벨 출력 위치의 기본값은 1.2 입니다.
```

```
    autopct = '%.1f%%'); # 썸 안에 백분율을 출력할 포맷을 지정합니다.  
                          [참고] 백분율 출력 위치의 기본값은 0.6 입니다.
```

[참고] 파이 차트 꾸미기

- 원하는 팔레트로 썰기의 채우기 색을 지정합니다.

```
>>> wedgeColors = sns.color_palette(palette = 'Spectral')
```

- 파이 차트의 다양한 그래픽 요소를 변경합니다.

```
>>> plt.pie(x = grp.values, explode = [0, 0, 0.2], labels = grp.index,
            autopct = '%.1f%%', colors = wedgeColors, # [참고] 팔레트명을 직접 지정하면
                                                    # 예러가 발생합니다.
            startangle = 90, counterclock = False, # 첫 번째 썰기 시작 위치와 출력 방향을
                                                    # 변경합니다.
            textprops = dict(color = '0', size = 12), # 글자 색과 크기를 지정합니다.
            wedgeprops = dict(ec = '0.5', lw = 0.5)); # 썰기의 테두리 색과 선의 두께를
                                                    # 지정합니다.
```

이변량 막대 그래프 그리기

- apt의 자치구별 거래금액 평균을 내림차순 정렬한 grp를 생성합니다.

```
>>> grp = apt.groupby(by = ['자치구']).mean()['거래금액']
```

```
>>> grp = grp.sort_values(ascending = False).round(1); grp.head()
```

- 자치구별 거래금액 평균으로 이변량 막대 그래프를 그립니다.

```
>>> sns.barplot(data = apt, x = '자치구', y = '거래금액', order = grp.index,
```

```
        estimator = np.mean, ci = None)
```

```
>>> plt.ylim(0, 21)
```

estimator 매개변수에 집계함수를 지정합니다.(기본값: 'mean')
ci 매개변수에는 95% 신뢰구간 출력 여부를 지정합니다.

```
>>> plt.xticks(rotation = 45);
```

[참고] 95% 신뢰구간이란 표본 평균이 모평균의 ± 2 표준편차 안에
95% 확률로 포함된다는 것을 의미합니다.

막대 위에 텍스트 추가

- 막대 위에 거래금액 평균을 텍스트로 추가합니다.

```
>>> for i, v in enumerate(grp):  
    plt.text(x = i, y = v, s = v,  
            ha = 'center', va = 'bottom',  
            color = 'black', fontsize = 9,  
            fontweight = 'bold')
```

[참고] 묶음 막대 그래프 그리기

- sub의 자치구와 금액구분별 거래금액 평균으로 grp를 생성합니다.

```
>>> grp = sub.groupby(by = ['자치구', '금액구분']).mean()['거래금액']
```

```
>>> grp = grp.round(1); grp.head()
```

[참고] grp는 자치구와 금액구분인 다중인덱스 *MultiIndex*를 갖는데,
다중인덱스는 레벨로 구분할 수 있습니다.

- 이변량 막대 그래프의 x축에 범주형 변수를 추가한 묶음 막대 그래프를 그립니다.

```
>>> sns.barplot(data = sub, x = '자치구', y = '거래금액', hue = '금액구분',
                order = grp.index.levels[0], hue_order = grp.index.levels[1],
                estimator = np.mean, ci = None)

>>> plt.ylim(0, 27)
```


[참고] 묶음 막대 위에 텍스트 추가

- 묶음 막대 위에 거래금액 평균을 텍스트로 추가합니다.

```
>>> for i, v in enumerate(grp):
```

```
    if i % 2 == 0:
```

```
        i = i/2 - 0.2 # 묶음 막대가 2개일 때, x축 좌표(인덱스)에 ±0.2씩 간격을 줍니다.  
                      # 따라서 grp의 인덱스가 짝수면 인덱스를 2로 나누고 0.2를 뺍니다.
```

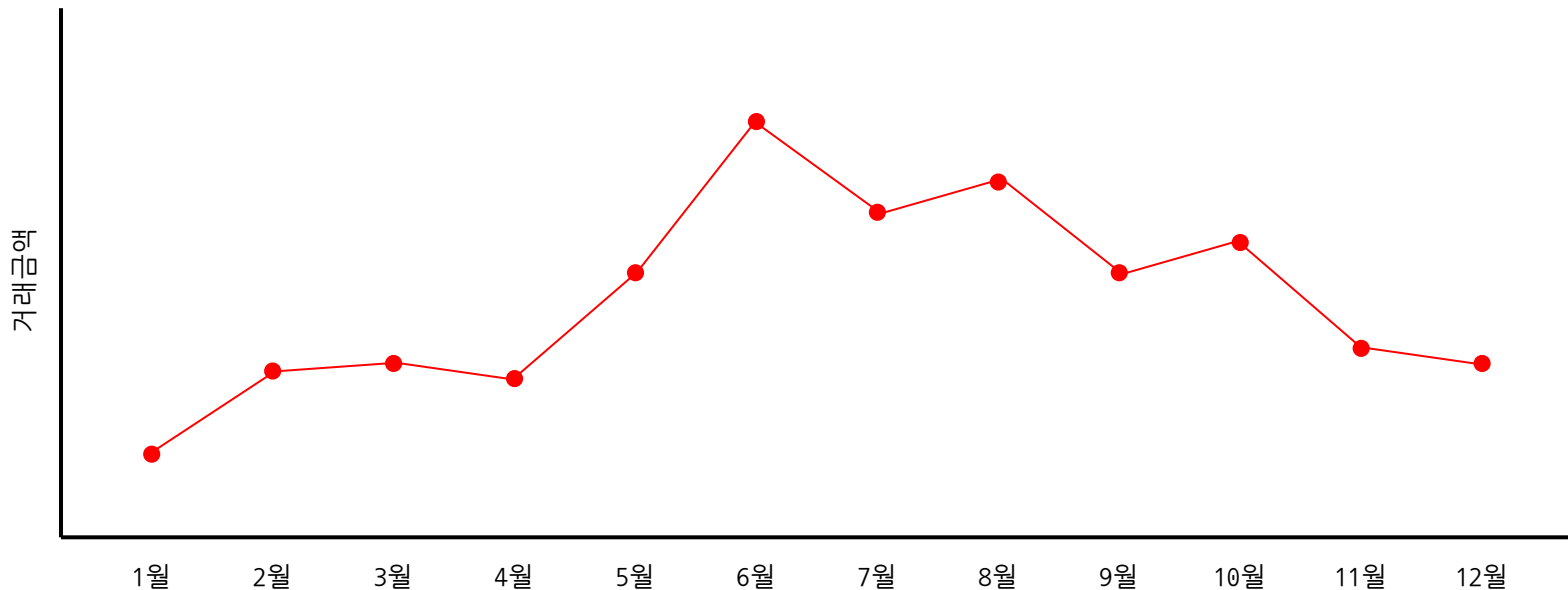
```
    else:
```

```
        i = (i-1)/2 + 0.2 # 만약 grp의 인덱스가 홀수면 인덱스에서 1을 뺀 값을 2로 나누고 0.2를  
                          # 더합니다.
```

```
    plt.text(x = i, y = v, s = v, ha = 'center', va = 'bottom')
```

선 그래프

- 선 그래프는 시간의 흐름에 따라 연속형 변수가 변하는 양상을 그린 그래프입니다.



선 그래프 그리기

- 거래월별 거래금액 평균으로 선 그래프를 그립니다.

```
>>> sns.lineplot(data = apt, x = '거래월', y = '거래금액', color = 'red',  
                  estimator = np.mean, ci = None);
```

- 선 그래프에 점을 추가합니다.

```
>>> sns.lineplot(data = apt, x = '거래월', y = '거래금액', color = 'red',  
                  estimator = np.mean, ci = None, marker = 'o');
```

선 그래프를 겹쳐서 그리기

- 자치구별 선 그래프를 겹쳐서 그립니다.

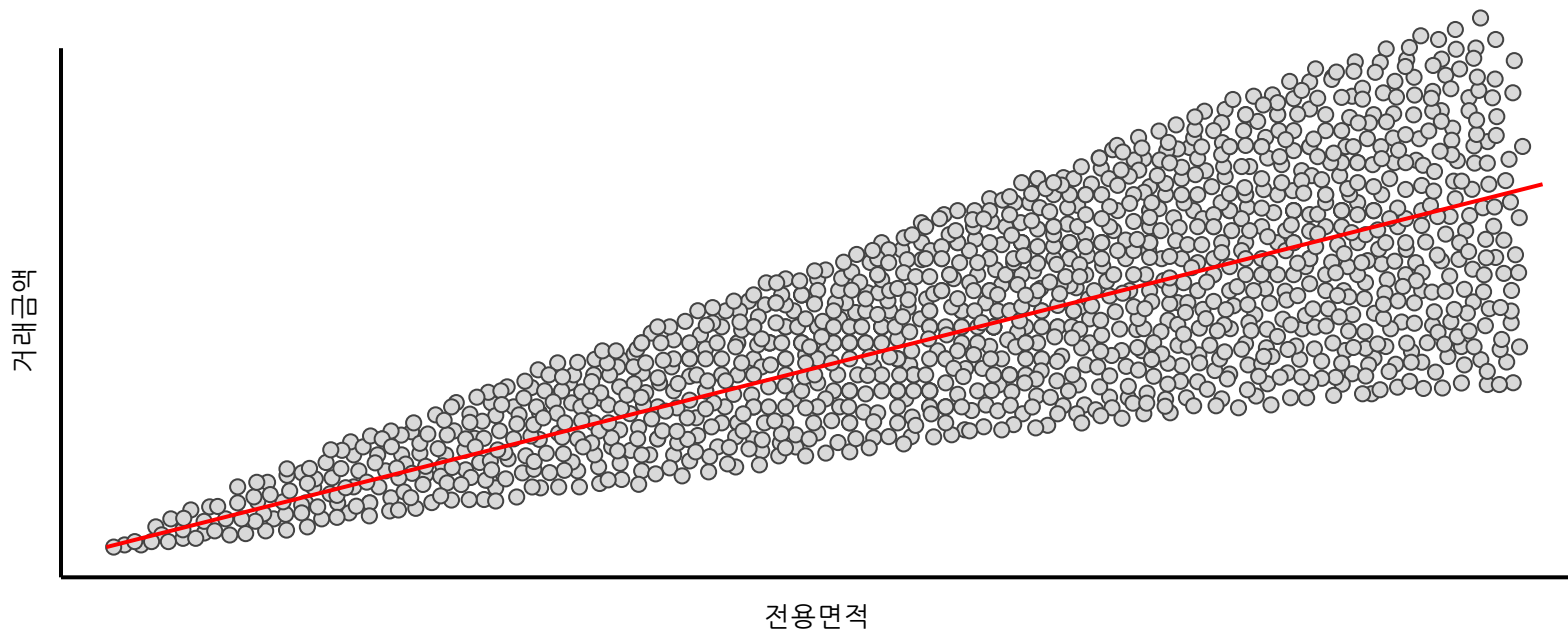
```
>>> sns.lineplot(data = sub, x = '거래월', y = '거래금액', hue = '자치구',
                  estimator = np.mean, ci = None, marker = 'o');
```

- 자치구별로 점 모양을 다르게 설정합니다.

```
>>> sns.lineplot(data = sub, x = '거래월', y = '거래금액', hue = '자치구',
                  estimator = np.mean, ci = None, markers = True,
                  style = '자치구'); # markers 매개변수에 True, style 매개변수에 범주형 변수를
                                     지정하면 범주형 변수에 따라 점 모양을 다르게 설정합니다.
```

산점도

- 산점도는 이변량 연속형 변수의 선형관계를 점으로 표현한 그래프입니다.



산점도 그리기

- 전용면적과 거래금액으로 산점도를 그립니다.

```
>>> sns.scatterplot(data = apt, x = '전용면적', y = '거래금액',
                    color = '0.3', # color 매개변수에 점의 채우기 색을 지정합니다.
                               [참고] 생략하면 기본 팔레트의 첫 번째 색으로 적용합니다.
                    ec = '0.8', # edgecolor 매개변수에 점의 테두리 색을 지정합니다.(기본값: '1')
                    s = 15, # s 매개변수에 점의 크기를 지정합니다.(기본값: 50)
                    alpha = 0.2); # alpha 매개변수에 채우기 색의 투명도를 0(투명) ~ 1(불투명)의
                               실수로 지정합니다.(기본값: 1)
```

점의 채우기 색 변경

- apt를 세대수로 오름차순 정렬합니다.

```
>>> apt = apt.sort_values(by = ['세대수'])
```

*# apt를 세대수로 오름차순 정렬하고 아래 산점도를
그리면 세대수가 큰 점이 더욱 뚜렷하게 보입니다.*

- 세대수(연속형 변수)에 따라 채우기 색을 다르게 설정합니다.

```
>>> sns.scatterplot(data = apt, x = '전용면적', y = '거래금액',  
                    hue = '세대수', palette = 'RdYlGn',  
                    ec = '0.8', s = 15, alpha = 0.2);
```

강남구 데이터로 산점도 그리기

- apt에서 강남구만 선택하고 gng에 할당합니다.

```
>>> gng = apt[apt['자치구'].eq('강남구')]
```

- gng로 산점도를 그립니다.

```
>>> sns.scatterplot(data = gng, x = '전용면적', y = '거래금액',  
                    color = '0.3', ec = '0.8', s = 15, alpha = 0.2);
```


산점도에 회귀직선, 수직선 및 수평선 추가

- 점과 회귀직선 관련 그래픽 요소를 딕셔너리로 생성합니다.

```
>>> pointProps = dict(color = '0.3', ec = '0.8', s = 15, alpha = 0.2)
```

```
>>> regProps = dict(color = 'red', lw = 1.5)
```

- 산점도에 회귀직선, 수직선(x 평균) 및 수평선(y 평균)을 추가합니다.

```
>>> sns.regplot(data = gng, x = '전용면적', y = '거래금액', ci = None,
```

```
                scatter_kws = pointProps, line_kws = regProps)
```

```
>>> plt.axvline(x = gng['전용면적'].mean(), lw = 0.5, ls = '--')
```

```
>>> plt.axhline(y = gng['거래금액'].mean(), lw = 0.5, ls = '--');
```

산점도 행렬 그리기

- 여러 산점도를 행렬 형태로 표현하면 여러 입력변수와 목표변수 간 관계를 빠르게 확인할 수 있습니다.
 - 산점도 행렬의 주대각에는 변수의 히스토그램, 삼각행렬에는 산점도를 그립니다.
 - 열 개수가 많으면 시간이 오래 걸리고 가독성이 떨어집니다.
- 산점도 행렬에 추가할 변수명으로 리스트를 생성합니다.

```
>>> cols = ['거래금액', '전용면적', '층', '세대수'] # [참고] 목표변수명과 입력변수명을  
# 차례대로 지정합니다.
```

- 선택한 변수로 산점도 행렬을 그립니다.

```
>>> sns.pairplot(data = sub[cols], plot_kws = pointProps);
```

산점도 행렬을 간결하게 그리기

- x축에 입력변수, y축에 목표변수를 지정하면 산점도 행렬을 간결하게 그립니다.

```
>>> sns.pairplot(data = sub,  
  
                 x_vars = ['전용면적', '층', '세대수'],  
  
                 y_vars = ['거래금액'], # [참고] 여러 변수명을 리스트로 지정하면 해당 개수만큼  
                                     산점도 행렬의 행이 증가합니다.  
  
                 kind = 'reg', # kind 매개변수에 'reg'를 지정하면 회귀직선을 추가합니다.  
                             [참고] 인수의 기본값은 'scatter'입니다.  
  
                 plot_kws = dict(scatter_kws = pointProps,  
  
                                line_kws = regProps));
```

End of Document