본 페이지는 편집 시 유의사항을 담고 있습니다.

페이지 수 오류를 방지하기 위해 실제 편집 작업 시에는 지워 주시기 바랍니다.

이 파일은 B5로 제작을 원하는 저자분들께서 사용하시면 됩니다.

편집 과정중 꽉차게 들어가는 이미지, 배경색상이 있을 경우 하얀색 종이가 보이지 않도록 이미지, 색상 크기 반영하셔야 하는 점 알려드립니다.

이파일은 실제 제작시 필요한 사방여백 3mm가 포함된 크기입니다.(188x263mm)

상하좌우 3mm는 실제 제작시 재단되어 반영되지 않으니 참고해주세요!

**어린 왕자**(제목을 적어주세요.)

**어린 왕자(제목을 적어주세요)**

**발 행**｜2023년 00월 00일

**저 자**｜생텍쥐 페리(저자명, 필명을 적어주세요)

**펴낸이**｜한건희

**펴낸곳**｜주식회사 부크크

**출판사등록**｜2014.07.15(제2014-16호)

**주 소**｜서울특별시 금천구 가산디지털1로 119 SK트윈타워 A동 305호

**전 화**｜1670-8316

**이메일**｜info@bookk.co.kr

**ISBN**｜979-11-410-0000-0

**www.bookk.co.kr**

**ⓒ 셍텍쥐 페리 2023**

본 책은 저작자의 지적 재산으로서 무단 전재와 복제를 금합니다.

어

린

왕

자

Evan, Sarah 지음

**CONTENT**

내용

[Chapter 1. 집필 동기 11](#_Toc132048333)

[1. 이 책을 쓰게 된 동기 11](#_Toc132048334)

[A. 강사 입장에서 11](#_Toc132048335)

[B. 수강생 입장에서 11](#_Toc132048336)

[2. 무엇을 배울 수 있는가 11](#_Toc132048337)

[Chapter 2. 개발환경설정 12](#_Toc132048338)

[1. Github 회원가입 12](#_Toc132048339)

[2. Streamlit 회원가입 12](#_Toc132048340)

[3. Git 설치 12](#_Toc132048341)

[4. Python 설치 12](#_Toc132048342)

[5. Visual Studio Code 설치 12](#_Toc132048343)

[6. 가상환경 Virtualenv 설치 12](#_Toc132048344)

[7. requirements.txt 파일 작성 12](#_Toc132048345)

[8. Streamlit, Hello World 12](#_Toc132048346)

[Chapter 3. Python 기본문법 13](#_Toc132048347)

[1. Python Basic Tutorial 13](#_Toc132048348)

[A. Python 프로그래밍 시작 13](#_Toc132048349)

[B. 주석처리 13](#_Toc132048350)

[C. Variables 14](#_Toc132048351)

[D. 사칙연산 17](#_Toc132048352)

[E. 비교 연산자Comparison Operators 18](#_Toc132048353)

[F. 논리 연산자Logical Operators 19](#_Toc132048354)

[G. String 21](#_Toc132048355)

[H. 리스트List 29](#_Toc132048356)

[I. 튜플Tuple 35](#_Toc132048357)

[J. 딕셔너리Dictionary 38](#_Toc132048358)

[K. 조건문 41](#_Toc132048359)

[L. 반복문Iteration 43](#_Toc132048360)

[M. 사용자 정의 함수User-Defined Function 49](#_Toc132048361)

[2. pandas 55](#_Toc132048362)

[A. 데이터 불러오기 57](#_Toc132048363)

[B. 컬럼 선택 57](#_Toc132048364)

[C. 행 선택 58](#_Toc132048365)

[D. .loc와 iloc 59](#_Toc132048366)

[E. describe() 61](#_Toc132048367)

[F. rename() 62](#_Toc132048368)

[G. value\_counts() 64](#_Toc132048369)

[H. isin() 65](#_Toc132048370)

[I. 시계열Time Series 데이터 핸들링 66](#_Toc132048371)

[J. GroupBy 연산 71](#_Toc132048372)

[3. Matplotlib 74](#_Toc132048373)

[A. 선 그래프(Line Plot) 74](#_Toc132048374)

[B. 막대 그래프(Bar Plot) 77](#_Toc132048375)

[C. 박스플롯(Boxplot) 79](#_Toc132048376)

[4. Seaborn 83](#_Toc132048377)

[A. 선 그래프(Line Chart) 83](#_Toc132048378)

[B. 막대 그래프(Bar Plot) 86](#_Toc132048379)

[5. Plotly 90](#_Toc132048380)

[1. Scikit-Learn 90](#_Toc132048381)

[Chapter 4. Streamlit 91](#_Toc132048382)

[1. Slider bar 91](#_Toc132048383)

[2. Select bar 91](#_Toc132048384)

[3. Tab 91](#_Toc132048385)

[4. Checkbox 91](#_Toc132048386)

[Chapter 5. 공공데이터 수집 92](#_Toc132048387)

[1. 서울열린데이터광장 92](#_Toc132048388)

[2. 공공데이터포털 92](#_Toc132048389)

[Chapter 6. 부동산 실거래가 대시보드 93](#_Toc132048390)

[1. 데이터 수집 93](#_Toc132048391)

[2. Home 화면 꾸미기 93](#_Toc132048392)

[3. 탐색적 자료분석 화면 꾸미기 93](#_Toc132048393)

[4. 머신러닝 모형 만들기 93](#_Toc132048394)

[5. Prediction 화면 꾸미기 93](#_Toc132048395)

[6. OpenAI 챗봇 구현하기 93](#_Toc132048396)

[Chapter 7. 배포 94](#_Toc132048397)

[1. secrets.toml 94](#_Toc132048398)

[2. 배포 94](#_Toc132048399)

[3. 마무리 94](#_Toc132048400)

[인용 자료 95](#_Toc132048401)

작가의 말 242

레옹 베르트에게 이 책을 바칩니다.

이 책을 어른에게 바친 데 대해 어린이들에게 용서를 바랍니다.

나에게는 그럴 만한 중요한 이유가 있습니다.

그것은 무엇보다도 그 사람은 이 세상에서 나와 가장 친한

친구이기 때문라는 점입니다. 그리고 그는 무엇이든지

알아들을 수 있으며 어린이들을 위한 책까지도 다 이해한다는 점입니다.

세 번째 이유는 그가 프랑스에 살고 있는데

그 곳에서 추위와 굶주림에 떨고 있다는 사실입니다.

그는 위로받아야 할 처지에 있는 것입니다.

그래도 이 모든 이유들이 부족하다면 예전 어린 시절의

그에게 이 책을 바치겠습니다. 어른들도 모두 한때는 어린이였으니까요.

(물론 그것을 기억하는 어른은 별로 없지만.)

그래서 바치는 글을 이렇게 고쳐 씁니다.

'어린 시절의 레옹 베르트에게 이 책을 바칩니다

# Chapter 1. 집필 동기

## 이 책을 쓰게 된 동기

### 강사 입장에서

### 수강생 입장에서

## 무엇을 배울 수 있는가

# Chapter 2. 개발환경설정

## Github 회원가입

### Streamlit 회원가입

### Git 설치

### Python 설치

### Visual Studio Code 설치

### 가상환경 Virtualenv 설치

### requirements.txt 파일 작성

### Streamlit, Hello World

# Chapter 3. Python 기본문법

## Python Basic Tutorial

### Python 프로그래밍 시작

프로그래밍의 첫번째 시작은 “Hello World!”를 출력하는 것에서부터 시작한다. Python에서는 다음과 같이 print()함수를 사용한다.

|  |
| --- |
| print("Hello, Streamlit!")  [결과]  Hello, Streamlit! |

### 주석처리

주석처리는 크게 1줄 주석처리와 다중 주석처리가 존재한다. 다음 코드를 통해 확인해 본다. 아래 코드를 실행하면 주석 처리된 곳은 생략하고, print( ) 함수만 실행되는 것을 확인할 수 있다.

|  |
| --- |
| # 한 줄 주석 처리  """  abc  다중 주석 처리  1234556  """  print("Hello, Streamlit!")  [결과]  Hello, Streamlit! |

### Variables

프로그래밍에서 변수Variables를 저장하는 방법은 크게 4가지(Camel case, Pascal case, Kebab case, Snake case) 방식으로 구분한다. 원 단어가 임의의 학생을 저장하는 변수 my student가 있다고 가정한다. 각자 원하는 방식을 사용하도록 한다. 다음 예시를 통해서 관련 코드를 익히도록 한다.

|  |
| --- |
| myStudent = “sarah” # Camel Case  MyStudent = “sarah” # Pascal Case  my-student = “sarah” # Kebab Case  my\_student = “sarah” # Snake Case |

Python에서 Data Structures를 말할 때, Primitive 자료형과 Non-Primitive 자료형으로 구분한다 (Jaiswal, 2017).[[1]](#footnote-1)

Table 1. Python Data Structures

|  |  |
| --- | --- |
| Primitive | Non-Primitive |
| Integer  Float  String  Boolean | Array  List  Tuple  Dictionary  Set  File |

여기에서 List는 크게 아래와 같이 Linear와 Non-Linear와 구분할 수 있다.

Table 2. List Data Types[[2]](#footnote-2)

|  |  |
| --- | --- |
| Linear | Non-Linear |
| Stacks  Queues | Graphs  Trees |

#### Integer

Integer는 정수형을 말하며, 4, 5, -1과 같은 숫자를 의미한다. 각자의 나이를 입력하고 출력하는 코드를 작성한다.

|  |
| --- |
| num\_int = 30  print(num\_int)  print(type(num\_int))  [결과]  30  <class 'int'> |

#### Float

Float는 실수형을 말하며, 일반적으로 1.11또는 3.11과 같은 소수를 표현한다. 이번에는 본인의 키를 입력하고 출력하는 코드를 작성한다.

|  |
| --- |
| my\_height = 181.8  print(my\_height)  print(type(my\_height))  [결과]  181.8  <class 'float'> |

#### String

String은 문자열을 말하며, 알파벳, 단어 또는 기타 문자의 모음이다. Python에서는 한쌍의 작은 따옴표(‘’) 또는 큰 따옴표(“”) 안에 일련의 문자를 포함하여 문자열을 생성할 수 있다. 우선 다음 코드를 확인해본다. 임의의 글자를 입력하고 출력하는 코드를 작성한다. 문자열을 다루는 방법에 대해서는 추후에 한번 더 다룰 예정이다.

|  |
| --- |
| my\_name = "evan"  print(my\_name)  print(type(my\_name))  [결과]  evan  <class 'str'> |

#### Boolean

일반적으로 True(참) 또는 False(거짓)을 표현하는 데이터 타입이며, True는 1로, False는 0으로 변환되기도 한다. 해당 데이터 타입은 특히 조건식 및 비교식에 매우 유용하게 활용할 수 있다. 임의의 두개의 숫자를 비교하는 코드를 작성하고, 결괏값을 확인해본다.

|  |
| --- |
| x = 5  y = 3  z = x == y  print(z)  print(type(z))  [결과]  False  <class 'bool'> |

### 사칙연산

일반적으로 형변환이라고 부른다. 임의의 두개를 변수를 만들고 사칙연산을 만들어 보도록 한다. 이 때 주의해서 봐야 할 것은 나눗셈이다. 두 개의 정수를 나눌 때, 정수가 실수형으로 변한다는 것에 주의해야 한다.

|  |
| --- |
| x = 6  y = 3  print(x + y)  print(x - y)  print(x \* y)  print(x / y)  [결과]  9  3  18  2.0 |

이번에는 사칙연산을 통해 나머지, 몫, 제곱을 구하는 연산자를 확인한다.

|  |
| --- |
| x = 6  y = 3  print(x % y)  print(x // y)  print(x \*\* y)  [결과]  0  2  216 |

### 비교 연산자Comparison Operators

비교 연산자를 통해서 알고자 하는 것은 비교의 결괏값이 True인지, False인지 구분하기 위함이다. 우선 비교 연산자는 크게 6개가 존재 한다. 아래 표를 통해서 확인하도록 한다.

Table 3. Comparison Operators

|  |  |
| --- | --- |
| Operator | Meaning |
| A == B | A와 B가 같다 |
| A != B | A와 B가 같지 않다 |
| A > B | A가 B보다 크다 |
| A < B | A가 B보다 작다 |
| A >= B | A가 B보다 크거나 같다 |
| A <= B | A가 B보다 작거나 같다 |

Python 코드의 예시를 통해 살펴보도록 한다.

|  |
| --- |
| x = 10  y = 11  print(x == y)  print(x != y)  print(x > y)  print(x < y)  print(x >= y)  print(x <= y)  [결과]  False  True  False  True  False  True |

### 논리 연산자Logical Operators

논리 연산자는 주어진 조건식이 True인지 False인지 확인하는데 사용하며, 의사 결정 시에 유용하게 사용된다.

|  |
| --- |
| a = 3  b = 5  print((a > 2) and (b >= 6)  [결과]  False |

위 코드는 Python에서 논리 연산자를 사용하는 예제이다. 논리 연산자는 크게 and, or, not 3개가 존재한다. 논리 연산자의 기본적인 개념을 표로 정리하였다.

Table 4. Logical Operators[[3]](#footnote-3)

|  |  |  |
| --- | --- | --- |
| Operator | Example | Meaning |
| and | X and Y | X와 Y가 모두 True이면 True |
| or | X or Y | X와 Y중 한 개라도 True이면 True |
| not | not X | X가 False이면 True |

다음은 위 논리 연산자의 모든 결과를 보여주는 표이다.

Table 5. Truth Table

|  |  |  |  |
| --- | --- | --- | --- |
| X | Y | X and B | A or B |
| True | True | True | True |
| True | False | False | True |
| False | True | False | True |
| False | False | False | False |

위 표가 실제로 Python에서도 구현이 되는지 코드로 확인한다. 먼저 and 연산자를 구현한 코드이며, 결과는 양쪽이 모두 True일 때, True를 반환한다.

|  |
| --- |
| print(True and True)  print(True and False)  print(False and True)  print(False and False)  [결과]  True  False  False  False |

이번에는 or 연산자를 구현한 코드이며, 결과는 양쪽 중 하나만 True여도 True로 결과가 반환이 된다.

|  |
| --- |
| print(True or True)  print(True or False)  print(False or True)  print(False or False)  [결과]  True  True  True  False |

not 연산자를 사용하면 반대로 결괏값이 반환되는 것도 같이 확인한다.

|  |
| --- |
| print(not True)  print(not False)  [결과]  False  True |

### String

앞에서 String에 대해서 간단하게 다뤘지만, 일반적으로 문자열은 보다 자세히 다뤄야 할 정도로 매우 중요하다.

#### String Operators

우선 문자열에는 덧셈 연산자와 곱셈 연산자 사용이 가능하다. 덧셈 연산자는 두개의 문자를 이어서 내보내고, 곱셈 연산자는 숫자만큼 문자열이 반복된다. 코드를 통해 확인해본다. 먼저 덧셈 연산자는 아래와 같이 표현할 수 있다.

|  |
| --- |
| a = "boy"  b = "loves"  c = "girl"  letter = a + " " + b + " " + c + " forever"  print(letter)  [결과]  boy loves girl forever |

이번에는 곱셈 연산자를 확인하는 코드를 작성한다.

|  |
| --- |
| x = "Love "  lyrics = x \* 5  print(lyrics)  [결과]  Love Love Love Love Love |

#### 인덱싱Indexing

인덱싱은 각각의 문자열 안에서 범위를 지정하는 것을 의미한다. 다음 이미지[[4]](#footnote-4)를 확인하면 문자열은 앞쪽에서는 0부터 시작하고, 뒤쪽에서는 -1부터 시작하는 것을 확인할 수 있다. 이 때, 기억해야 하는 것은 이번에 배울 인덱싱은 문자열, List, Tuple, array와 같은 자료형에도 동일하게 사용할 수 있다.



Figure 1. String Indexing

위 이미지를 토대로 간단하게 Streamlit Dashboard 문자열을 가지고 인덱싱을 진행하도록 한다 (독자도 임의대로 코드를 작성하여 실행하도록 해본다). 임의의 텍스트를 X 변수에 담고, X[인덱스번호]과 같은 형태로 코드를 실행하면 해당 인덱스번호의 글자가 조회된다.

|  |
| --- |
| X = "Streamlit Dashboard"  print(X[0]) # 0번째 글자 가져오기  print(X[5]) # 5번째 글자 가져오기  print(X[-1]) # -1번째 글자 가져오기  print(X[-3]) # -3번째 글자 가져오기  [결과]  S  m  d  a |

그런데, 만약 주어진 텍스트의 인덱스 번호를 초과하면 다음과 같은 에러가 발생이 된다.

|  |
| --- |
| print(X[100]) |

텍스트이(가) 표시된 사진

자동 생성된 설명

위 에러 메시지에서 주의 깊게 살펴봐야 하는 것은 ①『----> 1 print(X[100])』와 ②『**IndexError**: string index out of range』이다. ① 메시지는 어떤 코드에서 에러가 나타났는지를 표시해주는 것이고, ② 메시지는 에러코드의 내용이다. 간단히 요약하면, 1번째 print() 코드에서 **IndexError** 에러가 발생했다고 알려주는 것이다. 즉, X가 담고 있는 텍스트는 인덱스 번호가 100까지는 없기 때문에 조회할 것이 없다는 뜻이기도 하다. X 텍스트가 가지고 있는 인덱스 번호를 재 확인하여 입력하면 에러는 발생이 되지 않을 것이다.

#### 슬라이싱Slicing

인덱싱은 각 문자열의 문자를 한 개씩 추출한다면 슬라이싱은 범위를 지정하여 추출하는 것을 말한다. 즉, 기존 데이터에서 일부의 범위를 지정하여 하위 집합을 추출하는 것을 말한다. 이 슬라이싱의 문법은 일반적으로 [start:end:step]의 일반적인 패턴을 따르게 된다.

* start : 슬라이싱의 첫번째 인덱스이며, 해당 인덱스 번호부터 조회를 시작한다. 만약에, 생략을 하면 인덱스는 0번째부터 시작을 한다.
* end : 슬라이싱의 마지막 인덱스를 말한다. 그런데, 해당 인덱스 번호를 포함되지 않고 그 전까지만 조회가 된다. 만약에, 생략을 하면 해당 문자열의 전체 길이가 기본값으로 설정된다.
* step : 슬라이싱의 각 인덱스 사이에서 건너뛸 인덱스의 수를 말하며, 생략할 경우 기본값은 1이다.

주어진 예제를 확인하도록 한다. 이 때, 해당 인덱스 번호를 바꿔서 진행해서 원하는 대로 결과가 나오는지 확인한다.

|  |
| --- |
| greeting = "Hello Streamlit"  print(greeting[:])  print(greeting[6:])  print(greeting[:6])  print(greeting[3:8])  print(greeting[0:9:2])  [결과]  Hello Streamlit  Streamlit  Hello  lo St  HloSr |

인덱싱과 슬라이싱의 원리는 향후에 나오게 되는 List, Tuple 등 다른 자료형에도 그대로 사용이 되기 때문에, 반드시 기억하도록 한다.

#### 문자열 수정

주어진 문자열을 수정하는 코드를 작성한다. 일반적으로는 다음과 같은 코드 형태로 작성할 수 있다. 주어진 문자열 중 H를 A로 바꾸는 코드를 작성한다. 그런데 다음과 같은 에러가 난다.

|  |
| --- |
| greeting = 'Hello, Streamlit'  greeting[0] = 'A'  print(greeting) |

텍스트이(가) 표시된 사진

자동 생성된 설명

우선, 에러가 나는 이유는 문자열은 불변immutable 객체로 인식되기 때문이다. 불변 객체는 객체 생성 이후 내부의 상태가 변하지 않는 것을 말한다. 그렇다면, 새로운 객체를 만들려면 새로운 변수명을 만드는 방법으로 우회할 수 있다.

|  |
| --- |
| greeting = "Hello Streamlit"  new\_greeting = 'A' + greeting[1:]  print(new\_greeting)  [결과]  Aello, Streamlit |

#### String Methods

String 클래스 내부에는 다양한 메서드Methods들이 존재한다.[[5]](#footnote-5) 여기에서 자주 사용하는 여러 메서드들을 확인하도록 한다.

1. str.upper() : 영어 문자들을 모두 대문자로 변경한다.

|  |
| --- |
| sampleText = "hello, streamlit"  upper\_text = sampleText.upper()  print(upper\_text)  [결과]  HELLO, STREAMLIT |

2. str.upper() : 영어 문자들을 모두 소문자로 변경한다.

|  |
| --- |
| sampleText = "HELLO, STREAMLIT"  lower\_text = sampleText.lower()  print(lower\_text)  [결과]  hello, streamlit |

3. str.capitalize() : 영어 문자의 첫번째 글자만 대문자로 변경한다.

|  |
| --- |
| sampleText = "hello, streamlit"  cap\_text = sampleText.capitalize()  print(cap\_text)  [결과]  Hello, streamlit |

4. str.title() : 문자열 내의 각 단어들의 첫번째 글자만 대문자로 변경한다.

|  |
| --- |
| sampleText = "hello, streamlit"  title\_text = sampleText.title()  print(title\_text)  [결과]  Hello, streamlit |

5. str.strip() : 주어진 문자열의 앞뒤 공백이 있다면 모두 제거한다.

|  |
| --- |
| sampleText = " hello, streamlit "  stripped\_text = sampleText.strip()  print(stripped\_text)  [결과]  hello, streamlit |

6. str.replace(old, new) : 기존(old) 문자열을 새로운(new) 문자열로 바꾼다. 주어진 텍스트 hello에서 welcome으로 변경하는 코드를 작성한다.

|  |
| --- |
| sampleText = "hello, streamlit"  new\_text = sampleText.replace("hello", "welcome")  print(new\_text)  [결과]  welcome, streamlit |

7. str.split(sep=None, maxsplit=-1) : 문자열을 sep 파라미터에 입력한 분리 기준점으로 분리한 후 List[[6]](#footnote-6)로 반환한다.

|  |
| --- |
| sampleText = "hello, streamlit"  words = sampleText.split(sep=",")  print(words)  [결과]  ['hello', ' streamlit'] |

8. str.join(iterable) : List나 Tuple내의 값을 하나의 문자열로 변환한다.

|  |
| --- |
| words = ["hello", "streamlit"]  text = ", ".join(words)  print(text)  [결과]  hello, streamlit |

### 리스트List

Python에 존재하는 자료형으로 시퀀스Sequence 데이터를 다룬다. 시퀀스를 갖는 것은 데이터에 순서가 있다는 뜻이며, 순서가 존재하기 때문에 인덱스와 슬라이싱을 사용할 수 있다. 아래 코드는 주어진 객체가 시퀀스인지 확인하는 코드이다. Sequnece 클래스에 대한 설명은 본 책에서는 생략한다.[[7]](#footnote-7)

|  |
| --- |
| from collections.abc import Sequence  my\_num = 100  my\_list = [1, 2, 3]  my\_string = "hello"  is\_num = isinstance(my\_num, Sequence)  is\_list = isinstance(my\_list, Sequence)  is\_string = isinstance(my\_string, Sequence)  print(is\_num)  print(is\_list)  print(is\_string)  [결과]  False  True  True |

위 결과를 보면 알 수 있듯이 리스트와 문자열은 시퀀스 데이터로 판정을 받았지만, 수치형은 시퀀스 데이터가 아님을 알 수 있다. 그 외 다른 데이터 자료형을 정의하고, 테스트를 해보는 것은 독자에게 맡기도록 한다.

#### 리스트 생성

먼저 리스트를 생성하는 다양한 방법에 대해 살펴보도록 한다. 개발을 하면 종종임의의 빈 리스트Empty List를 생성할 때가 있기 때문에, 관련 코드는 한번씩 숙지하면 좋다. 마지막 변수 e에 저장된 리스트는 리스트 안에 또다른 리스트가 존재하는 것으로 중첩리스트Nested List라고 부르기도 한다.[[8]](#footnote-8)

|  |
| --- |
| a = []  b = list()  c = [2]  d = ['streamlit']  e = [1, 2, ['hello streamlit']]  print(a)  print(b)  print(c)  print(d)  print(e)  [결과]  []  []  [2]  ['streamlit']  [1, 2, ['hello streamlit']] |

#### 인덱싱 & 슬라이싱

이미 문자열에서 인덱싱과 슬라이싱에 관한 개념적인 설명은 문자열 인덱싱 및 슬라이싱에서 진행했기 때문에 여기에서는 간단하게 다루도록 한다.

|  |
| --- |
| a = [1, 2, 3]  print(a[0]) # 첫번째 요소  print(a[1]) # 두번째 요소  print(a[2]) # 세번째 요소  print(a[-1]) # 마지막 요소  [결과]  1  2  3  3 |

중첩리스트를 다루는 방법은 []를 연속해서 사용하는 것이다. 아래와 같이 리스트를 생성하도록 한다. 이 때, be with you만 조회하는 코드를 작성하면 다음과 같이 할 수 있다.

|  |
| --- |
| my\_list = [1, 2, ["I", "want", "to", "be", "with", "you"], 4, 5]  # be with you 조회  my\_list[2][3:]  [결과]  ['be', 'with', 'you'] |

my\_list의 인덱스에 2번째 인덱스를 선택하면 ["I", "want", "to", "be", "with", "you"]만 남게 되고, 그 후에 3번째 인덱스부터 마지막까지 지정하면 원하는 결과를 조회할 수 있다.

#### append()

기존 정의된 리스트에 새로운 값 1개를 추가할 때 사용한다.

|  |
| --- |
| letter = ["evan", "likes"]  letter.append("sarah's smile")  letter  [결과]  ['evan', 'likes', "sarah's smile"] |

#### extend()

append() 메서드는 리스트의 한 개의 값만 추가가 가능하다. 만약 여러 개의 값을 추가하고 싶다면 extend() 메서드를 사용한다.

|  |
| --- |
| letter = ["Do", "You"]  letter.extend(["Like", "Evan", "?"])  letter  [결과]  ['Do', 'You', 'Like', 'Evan', '?'] |

#### insert()

insert() 메서드는 특정 인덱스의 목록에 새로운 값을 추가한다.

|  |
| --- |
| letter = ["Evan", "has", "on", "sarah"]  letter.insert(2, "a crush")  letter  [결과]  ['Evan', 'has', 'a crush', 'on', 'sarah'] |

#### 리스트 값 변경

리스트는 변경가능한Mutable 객체이다. 연산자 “=”를 활용하면 새 값을 할당하여 리스트의 항목을 변경할 수 있다.

|  |
| --- |
| letter = ["Evan", "is", "with", "someone"]  letter[3] = "Sarah"  letter  [결과]  ['Evan', 'has', 'a crush', 'on', 'sarah'] |

#### 리스트 값 삭제

리스트 내의 값을 삭제하는 방법은 remove()를 이용할 수 있다. 이 때, remove() 메서드 안에는 리스트 내의 임의의 값을 입력하여 제거 할 수 있다. 임의의 숫자를 생성한 후, 두 개의 값을 연속해서 제거한다.

|  |
| --- |
| # 임의의 숫자 리스트를 생성한다.  dates = [302, 401, 418, 430, 529]    # 401를 제거한다.  dates.remove(401)  print(dates)  # 529를 제거한다.  dates.remove(529)  print(dates)  [결과]  [302, 418, 430, 529]  [302, 418, 430] |

그 외에도 pop()함수를 사용하거나 del 메서드를 사용하여 리스트 내의 값을 제거할 수 있다.

#### 리스트 값 정렬

리스트 값을 정렬하는 메서드로 sort()가 존재한다. 문자 및 숫자에 각각 적용한 후, 결과를 확인해본다. 먼저 문자는 가나다 또는 알파벳순으로 정렬된다.

|  |
| --- |
| letters = ["sarah", "loves", "evan"]  letters.sort()  letters  [결과]  ['evan', 'loves', 'sarah'] |

숫자는 오름차순으로 정렬된다.

|  |
| --- |
| dates = [329, 322, 317, 302, 217, 201]  dates.sort()  dates  [결과]  [201, 217, 302, 317, 322, 329] |

### 튜플Tuple

Python의 튜플은 리스트와 유사하다. 두 객체의 차이점은 튜플내의 값들이 한번 할당되면 변경할 수 없는 반면, 리스트는 변경 가능하다. 튜플을 생성하는 방법은 (값1, 값2) 형태로 저장한다. 아래 예시를 통해 확인하도록 한다.

|  |
| --- |
| my\_tuple = ("Hi", 1, 3.12)  print(my\_tuple)  my\_nested\_tuple = ("Hi", [1, 2, 3], (3.12, 1.23, 4.21))  print(my\_nested\_tuple)  [결과]  ('Hi', 1, 3.12)  ('Hi', [1, 2, 3], (3.12, 1.23, 4.21)) |

그런데, 소괄호( )가 없이 튜플을 생성할 수도 있다.

|  |
| --- |
| my\_tuple = 1, 2, 3  print(my\_tuple)  [결과]  (1, 2, 3) |

#### 인덱싱 및 슬라이싱

튜플에서도 인덱싱 및 슬라이싱은 가능하다. 인덱싱과 슬라이싱의 기본원리는 문자열, 리스트와 동일하다. 중첩 튜플(Nested Tuple)을 생성하고, 인덱싱과 슬라이싱을 진행한다.

|  |
| --- |
| my\_tuple = (1, 2, ("I", "want", "to", "be", "with", "you"), 4, 5)  # be with you 조회  my\_tuple[2][3:]  [결과]  ('be', 'with', 'you') |

#### 튜플 Methods

리스트에서 제공하는 append(), remove() 메서드 사용은 불가능하다. 그러나, count(), index() 메서드는 사용이 가능하다. 메서드 count()는 튜플 내의 특정 단어가 몇번 출현하는지 빈도를 파악하는 함수이고, index()는 튜플 내의 찾고자 하는 단어가 첫번째 등장하는 인덱스 값을 반환하는 함수이다.

|  |
| --- |
| my\_tuple = ("Love", "Love", "Love", "Hate", "Love", "Love")  print(my\_tuple.count('Love'))  print(my\_tuple.index('Hate'))  [결과]  5  3 |

#### 리스트와 튜플의 차이점[[9]](#footnote-9)

일반적으로 튜플은 리스트와 유사하며, 비슷한 상황에서 사용이 가능하다. 그런데, 아래와 같은 상황에서 약간의 차이점이 존재한다.

* 일반적으로 이기종Heterogeneous(다른) 데이터 유형에는 튜플을 사용하고 동종Homogeneous(유사)데이터 유형에는 리스트를 사용한다.
* 리스트가 수정가능한 객체인 반면 튜플은 불변이기 때문에 튜플을 통해서 반복문을 구성하는 것이 약간 속도가 빠르다고 알려져 있다.
* 불변인 객체 튜플은 딕셔너리 객체의 key값으로 구성이 될 수 있지만, 리스트는 딕셔너리 객체의 key값으로 구성이 될 수 없다.
* 만약, 변경할 필요가 없는 데이터를 가지고 있다면, 이를 만약에 튜플로 구현한다면 데이터가 쓰기-금지write-protected 상태로 유지될 수 있다.

### 딕셔너리Dictionary

딕셔너리는 key-value 값으로 이루어진 데이터로, 숫자 범위로 인덱싱되는 시퀀스와 달리 딕셔너리는 key로 인덱싱되며, key는 모든 불변 유형이며, 문자열과 숫자가 key값으로 사용될 수 있다. 튜플 또한 key값으로 사용할 수 있다. 그런데, 이 때에도 마찬가지로 튜플 내에서도 문자열과, 숫자가 있는 경우에만 가능하면, 다른 객체가 들어있는 경우 key값으로 사용될 수 없다. 리스트와 달리 key값은 슬라이싱, append(), extend()과 같은 리스트 관련 메서드를 사용할 수 없다는 점에 유의한다. 기본적으로 딕셔너리는 {} 형태로 저장된다.

|  |
| --- |
| my\_dic = {'name':'sarah', 'age': 20, 'job': 'Data Analyst',  'shopping':[1,2,3]}  print(my\_dic)  print(my\_dic['name'])  print(my\_dic['age'])  print(my\_dic['shopping'])  [결과]  {'name': 'sarah', 'age': 20, 'job': 'Data Analyst', 'shopping': [1, 2, 3]}  sarah  20  [1, 2, 3] |

기존 정의된 딕셔너리 my\_dic에 새로운 key-value값을 추가하는 방법은 아래와 같이 할 수 있다.

|  |
| --- |
| my\_dic['country'] = 'S. Korea'  print(my\_dic)  [결과]  {'name': 'sarah', 'age': 20, 'job': 'Data Analyst', 'shopping': [1, 2, 3], 'country': 'S. Korea'} |

기존에 정의된 값을 바꾸려면, 기존 key값에 새로운 value값을 입력한다. 만약, shopping key에 정보를 가방으로 바꾸도록 한다.

|  |
| --- |
| my\_dic['shopping'] = '가방'  print(my\_dic)  [결과]  {'name': 'sarah', 'age': 20, 'job': 'Data Analyst', 'shopping': '가방', 'country': 'S. Korea'} |

만약에 특정 value를 삭제하고 싶다면, del을 활용한다. 가방을 사지 않기로 가정하고 key값 shopping을 제거한다.

|  |
| --- |
| del my\_dic['shopping']  print(my\_dic)  [결과]  {'name': 'sarah', 'age': 20, 'job': 'Data Analyst', 'country': 'S. Korea'} |

#### 딕셔너리 메서드

딕셔너리 메서드는 다양하게 존재하지만, 자주 사용하는 것 3가지만 간단하게 확인한다. 먼저 정의된 딕셔너리에서 key값만 조회하는 메서드는 keys() 메서드가 있다.

|  |
| --- |
| my\_dic.keys()  [결과]  dict\_keys(['name', 'age', 'job', 'country']) |

이번에는 value값만 조회하는 메서드 values()를 확인해본다.

|  |
| --- |
| my\_dic.values()  [결과]  dict\_values(['sarah', 20, 'Data Analyst', 'S. Korea']) |

이번에는 key-value값을 동시에 조회하는 메서드 items()를 확인해본다. 이때 주의해야 하는 것은 key-value값은 튜플 형태로 묶여서 나온다.

|  |
| --- |
| my\_dic.items()  [결과]  dict\_items([('name', 'sarah'), ('age', 20), ('job', 'Data Analyst'), ('country', 'S. Korea')]) |

### 조건문

조건문이란, 특정한 조건이 True이면 해당 조건에 부합하는 코드 블록을 실행하는 것을 말한다. 그런데 때로는 여러 조건을 확인해야 하는 경우, 여러 구문을 사용할 수 있다. 먼저 단일 조건인 경우에는 아래와 같이 코드 작성이 가능하다.

|  |
| --- |
| a = 38  b = 31  if a >= b:  print("a가 b보다 크거나 같다")  else:  print("b가 a보다 작다")  [결과]  a가 b보다 크거나 같다 |

만약 다중 조건일 때는 if-elif-else 구문을 사용한다. 다음 문법을 통해 확인해본다.

|  |
| --- |
| a = 31  b = 31  if a > b:  print("a가 b보다 크다")  elif a == b:  print("a가 b와 같다")  else:  print("a가 b보다 작다")  [결과]  a가 b와 같다 |

지금까지 비교를 한 것은 숫자끼리의 비교이다. 그러나, 조건문은 단순히 숫자끼리의 비교 뿐만이 아니라, 문자열 매칭을 조건문으로 사용할 수 있다. 우선 아래 코드를 확인하면, 주어진 문자열에 let이 있는지 여부를 체크한 후, 해당 코드 블록이 실행되는 것을 확인할 수 있다.

|  |
| --- |
| my\_string = "Don't let them in, don't let them see"  if "let" in my\_string:  print("문자열 let이 존재한다.")  else:  print("문자열 let이 존재하지 않는다")  [결과]  문자열 let이 존재한다. |

Python 문법 중 한가지 특징은 조건문을 한 줄로 표현할 수도 있다. 기존에 작성했던 a, b 숫자 비교하는 것을 한 줄로 처리하도록 해본다.

|  |
| --- |
| a = 38  b = 31  result = "a가 b보다 크거나 같다" if a >= b else "b가 a보다 작다"  print(result)  [결과]  a가 b보다 크거나 같다 |

### 반복문Iteration

반복문이란, 특정 코드 블록을 여러 번 반복하는 과정을 말한다. 일반적으로 반복문은 for-loop 구문과, while 구문으로 나누어서 살펴볼 수 있다.

#### for Loop

Python에서 for-loop는 특정 횟수 동안의 코드 블록을 실행하는 데 사용된다. 일반적으로 리스트, 튜플, 문자열 등의 시퀀스를 반복하는데 사용된다.



for-loop 반복문의 기본 코드는 아래와 같이 작성할 수 있다.

|  |
| --- |
| for idx in sequence\_data:  # 코드 |

위 코드에서 idx는 시퀀스 데이터가 가지고 있는 각 값에 순차적으로 접근할 수 있다. for-loop는 시퀀스데이터의 전체 길이만큼 반복된 후 종료된다.

간단하게 아래 코드를 실행해본다.

|  |
| --- |
| programming = ['파이썬', 'HTML', 'CSS', '자바']  for idx in programming:  print(idx)  [결과]  파이썬  HTML  CSS  자바 |

이번에는 range() 메서드를 활용하여 반복문을 실행해본다. Python에서 제공하는 range() 메서드는 인덱스 번호가 0번째부터 시작을 한다. 만약 range(2)를 사용한다면, 인덱스 번호는 0, 1 두개가 출력이 된다. 다음 코드로 확인한다.

|  |
| --- |
| for i in range(2):  print(i)  [결과]  0  1 |

Python에서는 for loop에도 else 구문이 추가될 수 있다. 위 코드에 else 구문을 추가해본다.

|  |
| --- |
| for x in range(2):  print(x)  else:  print("반복문 완전 종료!")  [결과]  0  1  반복문 완전 종료! |

이번에는 enumerate() 메서드를 활용해본다. enumerate() 메서드를 사용할 경우, 시퀀스를 반복하면서 각 항목의 인덱스를 추적할 수 있다.

|  |
| --- |
| my\_tuple = ("Hi", 1, 3.12)  for i, item in enumerate(my\_tuple):  print(i, item)  [결과]  0 Hi  1 1  2 3.12 |

이번에는 zip() 메서드를 활용해본다. 해당 메서드의 가장 큰 장점은 서로 다른 두개의 리스트에서 연산을 수행할 수도 있다.

|  |
| --- |
| total\_sales = [5000, 4500, 300]  prices = [2.5, 3.3, 1.9]  for sales, price in zip(total\_sales, prices):  revenue = sales \* price  print(f'전체 매출액 : {revenue}')  [결과]  전체 매출액 : 12500.0  전체 매출액 : 14850.0  전체 매출액 : 570.0 |

#### List Comprehension

반복문을 한줄로 처리하는 방식이다. List Comprehension은 아래 그림[[10]](#footnote-10)과 같이 크게 3가지 구문으로 구성되어 있다.

차트이(가) 표시된 사진

자동 생성된 설명

주어진 리스트에서 문자열 매칭 코드를 작성해본다. 주어진 이름에서 알파벳 a가 들어간 이름만 나오도록 한다.

|  |
| --- |
| names = ["Evan", "Sarah", "John", "Issac"]  newlist = []  for name in names:  if "a" in name:  newlist.append(name)  print(newlist)  [결과]  ['Evan', 'Sarah', 'Issac'] |

위 복잡한 코드를 한줄로 정의할 수 있다.

|  |
| --- |
| names = ["Evan", "Sarah", "John", "Issac"]  newlist = [name for name in names if "a" in name]  print(newlist)  [결과]  ['Evan', 'Sarah', 'Issac'] |

#### While 문

While 조건문은 특정 조건이 참인 전제에서 코드 블록을 반복적으로 실행할 수 있는 구조이다. While 루프의 구문은 다음과 같이 작성할 수 있다.

|  |
| --- |
| while condition:  # 코드 블록 |

While Loop의 조건은 반복문이 실행되기전에 진행된다. 여기에서 조건이 참이면 코드 블록으로 넘어가고, 코드 블록이 실행된 후, 조건을 재확인한 후, 코드 블록이 실행되는 구조다. 만약, 조건이 참으로 유지되는 한 반복문은 계속 유지가 된다. 다음은 while Loop를 사용하여 1부터 5까지의 숫자를 출력하도록 한다.

|  |
| --- |
| i = 1  while i <= 3:  print(i)  i += 1  [결과]  1  2  3 |

이번에는 간단하게 로그인을 구현하는 예제를 만들어본다. 즉, 문자열을 활용하여 while loop를 만들어본다.

|  |
| --- |
| username = input("사용자 이름을 입력해주세요..!")  password = ''  while password != 'Sarah':  print('정확한 비밀번호가 필요합니다.')  password = input("비밀번호를 입력해주세요..!")  print(f'{username}님 환영합니다')  [결과]  사용자 이름을 입력해주세요..! Evan  정확한 비밀번호가 필요합니다.  비밀번호를 입력해주세요..! a  정확한 비밀번호가 필요합니다.  비밀번호를 입력해주세요..! Sarah  Evan님 환영합니다 |

### 사용자 정의 함수User-Defined Function

사용자 정의 함수는 자주 사용할 것 같은 기능을 분석가가 또는 개발자가 입맛에 재구성을 하는 것을 말한다. 일반적으로 사용자 정의 함수를 만들 때, 기본 함수들을 이용하거나, 동일 코드가 여러 번 반복할 것 같은데, 작성하는 것이 좋다. Python에서는 def 키워드를 제공하고 있으며, 아래와 같은 기본 문법으로 정의할 수 있다. 아래 코드에서 docstring은 소스 코드에 포함된 문서(document)라고 볼 수 있다. 즉, 함수에 대한 설명을 남기는 것이라고 보면 된다.

|  |
| --- |
| def 함수이름(매개변수):  “””docstring”””  # 코드  # 코드  return 반환값 |

이제 간단하게 함수를 구현하고 실행해본다.

|  |
| --- |
| def hello():  """이 함수는 'Hello World!'를 출력한다!"""  print("Hello World!") |

이 함수를 실행하면 Hello World!가 나올 것이다.

|  |
| --- |
| hello()  [결과]  Hello World! |

여기에서 help(hello)를 호출하면 아래와 같이 docstring[[11]](#footnote-11)에 정의된 구문을 확인할 수 있다.

|  |
| --- |
| help(hello)  [결과]  Help on function hello in module \_\_main\_\_:  hello()  이 함수는 'Hello World!'를 출력한다! |

이번에는 사용자 정의 함수에 매개변수를 추가하도록 한다.

|  |
| --- |
| def hello(name):  print("안녕!", name) |

위 정의된 함수에 임의의 이름을 입력한다.

|  |
| --- |
| hello("Sarah")  [결과]  안녕! Sarah |

숫자를 입력해도 일단 출력이 된다.

|  |
| --- |
| hello(123)  [결과]  안녕! 123 |

그런데, 숫자를 name으로 처리하는 적절하지 않기 때문에, string만 받도록 처리하는 함수를 구현한다. 특히 프로젝트를 진행할 때에도 아래와 같이 함수를 작성하는 것을 권장한다.

|  |
| --- |
| def hello(name: str) -> None:  if not isinstance(name, str):  raise TypeError("Name must be a string.")  print(f"안녕, {name}!") |

함수 안에 있는 isinstance 함수는 입력된 name이 str인지 아닌지 체크하는 함수이고, 만약, name이 str이 아니라면, TypeError 오류 메시지가 출력되도록 하는 함수이다. 만약, name이 정상적으로 str이라면, 그 다음 코드로 넘어가도록 설계하는 것이다. 실제로 테스트를 해보도록 한다. 기존 문자가 입력된 것은 특별한 에러 없이 잘 출력이 되지만, 두번째 코드는 아래와 같이 에러가 나타나는 것을 확인 할 수 있다.

|  |
| --- |
| hello("Sarah")  hello(111)  [결과]  안녕, Sarah! |

텍스트이(가) 표시된 사진

자동 생성된 설명

이번에는 가변 매개변수인 args에 대해 살펴보도록 한다. 일반적으로 사용자 정의함수에는 다양한 매개변수를 정의할 수 있다.

|  |
| --- |
| def add(a, b, c):  return a + b + c  print(add(1, 2, 3))  [결과]  6 |

그런데, 아래와 같이 한 개의 매개변수를 지정하지 않으면 에러가 발생한다.

|  |
| --- |
| print(add(1, 2)) |

텍스트이(가) 표시된 사진

자동 생성된 설명

이렇게 가변적인 상황에 대응할 수 있도록 도와주는 것이 args 이다. 함수를 재정의하고, 어떻게 데이터가 입력이 되는지 확인한다.

|  |
| --- |
| def add(\*args):  print(args, type(args))  return args  add(1, 2, 3)  [결과]  (1, 2, 3) <class 'tuple'>  (1, 2, 3) |

들어온 모든 값을 더하기 위해서는 반복문을 사용해야 한다. 함수 내부를 업그레이드 한다.

|  |
| --- |
| def add(\*args):  addition = 0  for number in args:  addition += number  return addition  print(add(1, 2))  print(add(1, 2, 3))  print(add(1, 2, 3, 4))  [결과]  3  6  10 |

그런데, 여전히 함수의 코드가 복잡하다. 좀 더 간결하게 하기 위해서 Python에서 제공하는 기본 함수 sum()을 활용한다.

|  |
| --- |
| def add(\*args):  return sum(args)  print(add(1, 2))  print(add(1, 2, 3))  print(add(1, 2, 3, 4))  [결과]  3  6  10 |

이번에는 \*\*kwargs에 대해 알아본다. 일반적으로 함수에서 정의되지 않은 매개변수를 받을 때 사용되는데, 딕셔너리 형식으로 전달하게 된다. 아래 예시 코드를 통해 확인한다.

|  |
| --- |
| def temp(\*\*kwargs):  for key, value in kwargs.items():  print(key, value)  temp(name="Sarah", age=30, city="Seoul")  [결과]  name Sarah  age 30  city Seoul |

이번에는 개별적인 문자열만 받아서, 하나의 문장으로 완성하는 코드를 작성하는 함수를 만들어본다.

|  |
| --- |
| def whatIsLove(\*\*kwargs):  result = ""  for arg in kwargs.values():  result += arg + ", "  return result  print(whatIsLove(a="Love is patient",  b="love is kind",  c="It does not envy"))  [결과]  Love is patient, love is kind, It does not envy, |

만약, 다른 텍스트를 더 추가하더라도, 계속적으로 문장을 이어갈 수 있다.

## pandas

pandas[[12]](#footnote-12)는 데이터 조작 및 분석 도구를 제공하는 Python 라이브러리이다. 일반적으로 CSV 또는 Excel 파일과 같은 구조화된 데이터를 전처리(Processing)하고 다루는(Handling)데 사용된다. Pandas를 사용하면 사용자가 간단하고 효율적인 방식으로 처리하는 불러오고, 정제하고, 필터링하고, 변환하고 시각화를 할 수 있다. 또한 데이터 단순 집계, 그룹화를 활용한 집계, 피벗 테이블에 대한 매우 유용한 기능을 제공한다. 데이터 분석가 뿐만 아니라, 데이터 싸이언티스트, 데이터 엔지니어 등을 포함한 데이터를 다루는 모든 사람에게는 필수적인 도구이다.

일반적으로 pandas는 크게 두개의 객체를 가지고 있다. 먼저 Series 객체는 정수, 실수, 문자열 등 다양한 데이터 유형을 저장할 수 있는 1차원 배열과 유사한 객체이며, Series의 각 요소에 레이블을 지정하는 인덱스가 존재한다. Series를 생성하는 방법은 아래와 같다.

|  |
| --- |
| import pandas as pd  d = {'a': 1, 'b': 2, 'c': 3}  ser = pd.Series(data=d, index=['a', 'b', 'c'])  ser  [결과]  a 1  b 2  c 3  dtype: int64 |

데이터프레임DataFrame 객체는 여러 개의 Series 또는 서로 다른 유형의 배열을 저장할 수 있는 2차원 배열과 유사한 객체로 정의할 수 있다. 각 데이터프레임의 각 요소에 레이블을 지정하는 행, 열 인덱스가 모두 존재한다. 다음은 데이터프레임의 일반적인 예시이다.

|  |
| --- |
| import pandas as pd  d = {'col1': [1, 2], 'col2': [3, 4]}  df = pd.DataFrame(data=d)  print(df)  [결과]  col1 col2  0 1 3  1 2 4 |

데이터 전처리를 하면, 반환값이 Series로 나올 때도 있고, DataFrame으로 나올때도 있기 때문에, 중간에 한번씩 확인하는 것이 좋다.

### 데이터 불러오기

데이터를 불러오는 방법은 pd.read\_csv(각 파일의 경로) 함수를 활용하면 사용할 수 있다. 본 프로젝트와 유사한 부동산 데이터를 불러온다.[[13]](#footnote-13)

|  |
| --- |
| import pandas as pd  df\_boston = pd.read\_csv("./data/boston.csv")  df\_boston |

테이블이(가) 표시된 사진

자동 생성된 설명

### 컬럼 선택

특정한 컬럼을 선택할 때는 List에 특정 컬럼명을 작성하여 아래와 같이 선택할 수 있다. 독자들도 임의의 컬럼을 작성하여 추출하도록 한다.

|  |
| --- |
| cols = ['CRIM', 'ZN', 'INDUS']  result = df\_boston[cols].head()  print(result)  [결과]  CRIM ZN INDUS  0 0.00632 18.0 2.31  1 0.02731 0.0 7.07  2 0.02729 0.0 7.07  3 0.03237 0.0 2.18  4 0.06905 0.0 2.18 |

### 행 선택

행을 선택할 때는 조건식을 통해서 처리하는 경우가 일반적이다. 예를 들어 ZN에서 18.0에 해당하는 행만 선택하는 코드는 아래와 같이 할 수 있다. 조건식을 작성할 때는 비교연산자를 통해서 작성이 가능하다.

|  |
| --- |
| result = df\_boston[df\_boston['ZN'] == 18.0]  cols = ['CRIM', 'ZN', 'INDUS']  print(result[cols])  [결과]  CRIM ZN INDUS  0 0.00632 18.0 2.31 |

### .loc와 iloc

pandas에서 보편적으로 행과 열을 선택하는 방법은 .loc와 .iloc를 사용하는 것이다. 두 메서드의 가장 큰 차이는 loc는 행과 열을 선택할 때, labels/names로 접근하는 방식이고, iloc는 행과 열을 선택할 때, index/position으로 접근하는 방식이다. 만약, loc를 사용해서 행과 열을 선택할 때 매칭되는 label/name이 없다면 key error가 발생한다. 반면, iloc를 사용해서 행과 열을 선택할 때 매칭되는 index가 없다면 index error가 발생한다.

.loc를 활용하여 존재하지 않는 “CRI”를 입력하면 KeyError가 발생한다.

|  |
| --- |
| df\_boston.loc['CRI'] |

텍스트이(가) 표시된 사진

자동 생성된 설명

.iloc를 활용하여 존재하지 않는 컬럼 인덱스 20을 다음과 같이 입력하면 IndexError가 발생한다.

|  |
| --- |
| df\_boston.iloc[:, 20] |

텍스트이(가) 표시된 사진

자동 생성된 설명

#### loc

간단한 예제를 통해 loc가 어떻게 적용되는지 확인한다. 아래 코드는 CRIM, ZN, target 컬럼만 추출하면서, CRIM은 1보다 작은 행을 조회하도록 한다.

|  |
| --- |
| cols = ['CRIM', 'ZN', 'target']  result = df\_boston.loc[df\_boston['CRIM'] < 1, cols]  print(result.head(3))  [결과]  CRIM ZN target  0 0.00632 18.0 24.0  1 0.02731 0.0 21.6  2 0.02729 0.0 34.7 |

이번에는 다중 조건을 입력하도록 한다. 위 조건에 추가적으로 target이 24보다 이상인 데이터만 조회하도록 한다. 아래 코드에서 &은 and 연산자를 의미하고, |은 or 연산자를 의미한다. 독자분들은 & 연산자 대신에 | 연산자를 대입하면 다른 결과가 나오는 것을 확인할 수 있다.

|  |
| --- |
| cols = ['CRIM', 'ZN', 'target']  result = df\_boston.loc[(df\_boston['CRIM'] < 1) &\  (df\_boston['target'] >= 24.0), cols]  print(result.head(3))  [결과]  CRIM ZN target  0 0.00632 18.0 24.0  2 0.02729 0.0 34.7  3 0.03237 0.0 33.4 |

#### iloc

이번에는 iloc를 활용하도록 한다. 일반적으로 iloc를 활용할 때 행은 조건식이 아닌 인덱싱 및 슬라이싱을 활용하여 추출하는 것에 유의한다. 마찬가지로 컬럼도 인덱싱 및 슬라이싱을 활용하여 추출하는 것이다. 예를 들면, 0~4번째 인덱스 행과, 처음 3개의 컬럼만 조회한다고 하면 아래와 같이 작성할 수 있다.

|  |
| --- |
| result = df\_boston.iloc[0:5, 0:3]  print(result)  [결과]  CRIM ZN INDUS  0 0.00632 18.0 2.31  1 0.02731 0.0 7.07  2 0.02729 0.0 7.07  3 0.03237 0.0 2.18  4 0.06905 0.0 2.18 |

만약, 임의의 한 개의 행만 조회한다면, 데이터프레임에서 Series 형태로 바뀐다는 것에 주의한다.

|  |
| --- |
| result = df\_boston.iloc[20, 0:3]  print(result)  print(type(result))  [결과]  CRIM 1.25179  ZN 0.00000  INDUS 8.14000  Name: 20, dtype: float64  <class 'pandas.core.series.Series'> |

### describe()

주어진 데이터의 기초 통계량을 구할 때 쓰는 함수이며, 평균, 표준편차, 각 컬럼의 사분위를 그릴 수 있다.

|  |
| --- |
| cols = ['CRIM', 'ZN', 'target']  print(df\_boston[cols].describe())  [결과]  CRIM ZN target  count 506.000000 506.000000 506.000000  mean 3.613524 11.363636 22.532806  std 8.601545 23.322453 9.197104  min 0.006320 0.000000 5.000000  25% 0.082045 0.000000 17.025000  50% 0.256510 0.000000 21.200000  75% 3.677083 12.500000 25.000000  max 88.976200 100.000000 50.000000 |

### rename()

컬럼명을 변경할 때 사용한다. 이 때, 딕셔너리 형태로 추가한다. 우선 기존 데이터셋에서 일부 데이터만 추출한다.

|  |
| --- |
| sample\_df = df\_boston.iloc[:, 0:4]  sample\_df.head() |



컬럼명을 바꾸는 방법은 아래와 같이 할 수 있다.

|  |
| --- |
| data.reanme(columns={‘Old컬럼명’ : ‘New컬럼명’}) |

추출된 sample\_df데이터에서 ZN 컬럼명을 landZone으로 변경하는 코드를 작성하도록 한다.

|  |
| --- |
| sample\_df = sample\_df.rename(columns={'ZN': 'landZone'})  sample\_df.head() |



만약, 이번에는 두개의 컬럼을 변경한다면 딕셔너리 안에 연속해서 key-value 형태로 값을 추가하면 된다. 기존 컬럼 CRIM과 INDUS을 각각 crime과 businessRatio로 변경하도록 한다.

|  |
| --- |
| sample\_df = sample\_df.rename(columns={'CRIM': 'crime',  'INDUS':'businessRatio'})  sample\_df.head() |



### value\_counts()

각 컬럼의 여러 값(value)에 대한 모든 숫자를 세어보는 메서드로 매우 기초적인 집계함수를 제공한다. 데이터타입이, object, int, 등 상관없이 사용이 가능하다. 데이터 df\_boston에서 RAD에 해당 함수를 적용해본다. 이 때 결괏값은 DataFrame에서 Series 형태로 반환되는 것에 주의한다.

|  |
| --- |
| df\_boston['RAD'].value\_counts() |



|  |
| --- |
| type(df\_boston['RAD'].value\_counts())  [결과]  pandas.core.series.Series |

이 메서드 매개변수에는 normalize라는 매개변수가 존재한다. 이 매개변수를 활성화하면 빈도수가 아닌 비율로 표시된다.

|  |
| --- |
| df\_boston['RAD'].value\_counts(normalize=True) |



### isin()

특정 컬럼의 다양한 값 중에서 일부의 값만 가져오는 코드를 작성할 때 유용하게 사용할 수 있다. 메서드 안에는 일반적으로 리스트로 정의한 값이 적용된다. 아래 코드를 통해 확인해본다. 가상의 데이터를 만든 후, 특정 ‘구’만 출력이 되도록 코드를 작성한다.

|  |
| --- |
| import pandas as pd  df = pd.DataFrame({'시군구': ['관악구', '서대문구', '강남구', '서초구'],  '부동산가격': [10000, 20000, 30000, 40000]})  cities = ['관악구', '서대문구']  filtered\_df = df[df['시군구'].isin(cities)]  filtered\_df |



이번에는 숫자에 적용을 하도록 해본다. 리스트 numbers에 정의된 값만 출력이 되는 것을 확인할 수 있다.

|  |
| --- |
| numbers = [1.0, 7.0]  filtered\_df = df\_boston[df\_boston['RAD'].isin(numbers)]  filtered\_df[['CRIM', 'RAD']].head(6) |



### 시계열Time Series 데이터 핸들링

pandas 라이브러리에서 시계열 데이터는 인덱스가 날짜 혹은 시간인 데이터를 말한다. 만약 가상으로 시계열 자료를 생성하려면 인덱스를 숫자가 아닌 DatatimeIndex 자료형[[14]](#footnote-14)으로 만들어야 한다. 시계열 데이터를 만들 때 일반적으로 기본적으로 기억해야 하는 주요 함수는 pd.to\_datetime, pd.date\_range, pd.Timestamp 등이 있지만 여기에서는 pd.to\_datetime만 집중해서 본다.

#### pd.to\_datetime()

일반적으로 기존 문자열, 날짜/시간, 리스트, Tuple 등 입력된 날짜를 pandas datetime 객체로 변환한다. 간단한 예를 들면 아래와 같이 작업할 수 있다. 먼저 한 개의 문제를 변경할 때는 아래와 같이 진행한다.

|  |
| --- |
| date\_string = '2023-03-02'  datetime\_obj = pd.to\_datetime(date\_string)  print(datetime\_obj)  print(type(datetime\_obj))  [결과]  2023-03-02 00:00:00  <class 'pandas.\_libs.tslibs.timestamps.Timestamp'> |

만약 리스트로 정의를 한다면 아래와 같이 처리가 된다.

|  |
| --- |
| date\_list = ['2023-03-02']  datetime\_obj\_list = pd.to\_datetime(date\_list)  print(datetime\_obj\_list)  print(type(datetime\_obj\_list))  [결과]  DatetimeIndex(['2023-03-02'], dtype='datetime64[ns]', freq=None)  <class 'pandas.core.indexes.datetimes.DatetimeIndex'> |

이번에는 가상의 데이터를 불러오고 object를 datetime으로 변환하는 방법을 확인한다. (데이터셋 참조 : House Property Sales Time Series[[15]](#footnote-15))

|  |
| --- |
| sales = pd.read\_csv('data/raw\_sales.csv')  sales.head(3) |

테이블이(가) 표시된 사진

자동 생성된 설명

데이터의 정보를 확인하면 datesold는 datetime이 아니라 object인 것을 확인할 수 있다.

|  |
| --- |
| sales.info()  [결과]  <class 'pandas.core.frame.DataFrame'>  RangeIndex: 29580 entries, 0 to 29579  Data columns (total 5 columns):  # Column Non-Null Count Dtype  --- ------ -------------- -----  0 datesold 29580 non-null object  1 postcode 29580 non-null int64  2 price 29580 non-null int64  3 propertyType 29580 non-null object  4 bedrooms 29580 non-null int64  dtypes: int64(3), object(2)  memory usage: 1.1+ MB |

이제 위 데이터를 pd.to\_datetime()을 활용하여 object를 datetime으로 변환한다.

|  |
| --- |
| sales['datesold'] = pd.to\_datetime(sales['datesold'])  sales.info()  [결과]  <class 'pandas.core.frame.DataFrame'>  RangeIndex: 29580 entries, 0 to 29579  Data columns (total 5 columns):  # Column Non-Null Count Dtype  --- ------ -------------- -----  0 datesold 29580 non-null datetime64[ns]  1 postcode 29580 non-null int64  2 price 29580 non-null int64  3 propertyType 29580 non-null object  4 bedrooms 29580 non-null int64  dtypes: int64(3), object(2)  memory usage: 1.1+ MB |

* 시계열의 전체 기간 길이 구하기

이번에는 위 데이터의 전체 기간의 길이를 구하도록 한다.

|  |
| --- |
| sales["datesold"].max() - sales["datesold"].min()  [결과]  Timedelta('4553 days 00:00:00') |

* 연도, 월, 일 구하기

이번에는 주어진 날짜 데이터에서 연도, 월, 일을 구하도록 한다.

|  |
| --- |
| sales["year"] = sales["datesold"].dt.year  sales["month"] = sales["datesold"].dt.month  sales["day"] = sales["datesold"].dt.day  sales.head() |

테이블이(가) 표시된 사진

자동 생성된 설명

#### shift()

shift 연산을 사용하면 인덱스는 그대로 두고 데이터만 이동이 가능하다. 이 때 시계열은 앞으로 또는 뒤로 지정된 기간만큼 이동한다. 다음 샘플 코드를 통해 확인해본다.

|  |
| --- |
| temp\_df = sales[['datesold', 'price']].copy()  temp\_df['shifted\_v1'] = temp\_df['price'].shift(1, fill\_value=0).astype(int)  temp\_df['shifted\_v2'] = temp\_df['price'].shift(2, fill\_value=0).astype(int)  temp\_df.head() |



이번에는 shift(-1), shift(-2)를 대입하도록 한다. 이 때 결과는 반대로 데이터의 마지막 부분을 확인해야 한다.

|  |
| --- |
| temp\_df['shifted\_v3'] = temp\_df['price'].shift(-1, fill\_value=0).astype(int)  temp\_df['shifted\_v4'] = temp\_df['price'].shift(-2, fill\_value=0).astype(int)  temp\_df.tail() |

테이블이(가) 표시된 사진

자동 생성된 설명

#### strftime()

주어진 datesold 컬럼에 한글 형태로 바꿔서 저장을 하도록 하는데, 이때 strftime() 메서드를 이용하여 문자열을 만들 수 있다.

|  |
| --- |
| temp\_df['한글날짜'] = temp\_df['datesold'].dt.strftime("%Y년 %m월 %d일")  temp\_df.head() |

테이블이(가) 표시된 사진

자동 생성된 설명

### GroupBy 연산

GroupBy 연산은 하나의 데이터프레임 또는 Series에 있는 특정 컬럼안에 있는 여러개의 값(value)을 열로 그룹화한 다음 집계 함수를 수행할 수 있는 방법을 말한다. 그룹별 통계량을 확인하고자 할 때 매우 유용하게 활용할 수 있다. 아래 이미지[[16]](#footnote-16)를 통해서 확인하면 크게 Split-Apply-Combine 3단계로 구성됨을 알 수 있다.



Figure 2.

Split단계에서는 groupby() 컬럼에서 그룹화할 수 있는 영역을 찾고, 분리를 한다. Apply단계에서는 독립된 그룹별 함수를 적용한다. 위 그림에서는 sum()함수가 사용이 되었지만, 그 외에도 다른 집계 함수 사용이 가능하다. Combine단계에서는 각각의 독립된 그룹별로 함수가 적용된 결과를 종합하여 다시 하나의 테이블을 합치게 된다. 아래 코드를 통해서 살펴보도록 한다.

|  |
| --- |
| agg\_dict = {"price" : "mean", "bedrooms" : "mean"}  grouped = sales.groupby("propertyType").agg(agg\_dict)  grouped |



위 코드는 sales 데이터에서 "propertyType" 열의 값에 따라 그룹화하고, 각 그룹에서 "price"와 "bedrooms" 열의 평균 값을 구하는 코드이다. 여기에서 확인해야 하는 코드는 agg\_dict 변수인데, ‘price’ 열에 대해서는 평균값을 계산하고, ‘bedroom’ 열에 대해서도 평균값을 처리하도록 구조화하였고, 이를 grouped에 변수로 저장하여 출력하는 것이다. 출력된 형태는 DataFrame이며 인덱스는 properType으로 지정된 것을 확인할 수 있다. 만약, 인덱스로 정의된 컬럼을 일반 컬럼으로 변환하고자 한다면 reset\_index()를 실행한다.

|  |
| --- |
| grouped.reset\_index() |



이번에는 컬럼별 다른 집계함수를 적용하도록 한다. 예를 들면, price에는 전체 합계와 중간값, bedrooms에는 평균값만 도출하도록 하는 것이다. 기존 정의된 딕셔너리에서 수정할 것은 value의 값을 리스트로 재정의 하는 것이다.

|  |
| --- |
| agg\_dict = {"price" : ["sum", "median"], "bedrooms" : "mean"}  grouped = sales.groupby("propertyType").agg(agg\_dict)  grouped |



만약 위 컬럼을 평탄화를 하려면 결괏값으로 도출된 데이터프레임 columns에 대한 이해가 필요하다. 우선 아래 코드를 살펴보면 결괏값이 튜플형태로 출력되는 것을 확인할 수 있다.

|  |
| --- |
| grouped.columns  [결과]  MultiIndex([( 'price', 'sum'),  ( 'price', 'median'),  ('bedrooms', 'mean')],  ) |

위 개념을 확인해서 MultiIndex로 된 코드를 하나의 열로 치환하는 코드는 아래와 같다. 결과물을 확인하면 일반적인 데이터프레임 형태로 변경되었다. 추가 설명을 하면 먼저 빈 리스트를 만든 다음 반복문을 사용하는데, 이 때 핵심이 되는 코드는 각 열에 대해 isinstance(col, tuple)인지 확인하는 코드이다. 만약 튜플형태로 판정되면, new\_col에 f-string문자열의 원리를 이용하여 튜플의 값을 연결하여 저장하고, 빈리스트에 하나씩 추가하여 기존 컬럼에 대입하면 최종적으로 업데이트가 되도록 하는 코드이다.

|  |
| --- |
| new\_columns = []  for col in grouped.columns:  if isinstance(col, tuple):  new\_col = f'{col[0]}\_{col[1]}'  else:  new\_col = col  new\_columns.append(new\_col)  grouped.columns = new\_columns  grouped |



## Matplotlib

Streamlit 라이브러리에서는 matplotlib.pyplot figure가 나타나도록 지원하고 있다.[[17]](#footnote-17) 따라서, 기본적인 시각화를 익히도록 한다. 시각화의 차트 종류는 매우 다양하지만, 여기에서는 본 프로젝트에서 사용한 시각화 차트 위주로 간단하게 준비하였다.

### 선 그래프(Line Plot)

부동산 데이터셋 중 시계열 데이터가 적용된 데이터셋을 불러온 후 간단하게 선 그래프를 그려본다. 데이터셋을 가져올 때, pandas와 다른 부분은 read\_csv() 메서드 내 파라미터를 적용해서 object가 아닌 datetime64로 저장되고 있다는 차이점이 있다.

|  |
| --- |
| import pandas as pd  import matplotlib.pyplot as plt  sales = pd.read\_csv('data/raw\_sales.csv', parse\_dates=['datesold'])  sales.head(3) |



데이터셋의 기본 정보는 아래와 같다. 앞서 pandas와 다르게 여기에서는 datetime64로 적용된 것을 확인할 수 있다.

|  |
| --- |
| print(sales.info())  [결과]  <class 'pandas.core.frame.DataFrame'>  RangeIndex: 29580 entries, 0 to 29579  Data columns (total 5 columns):  # Column Non-Null Count Dtype  --- ------ -------------- -----  0 datesold 29580 non-null datetime64[ns]  1 postcode 29580 non-null int64  2 price 29580 non-null int64  3 propertyType 29580 non-null object  4 bedrooms 29580 non-null int64  dtypes: datetime64[ns](1), int64(3), object(1)  memory usage: 1.1+ MB  None |

이제 pandas 데이터프레임의 그룹 연산을 활용하여 연도별 평균 주택가격을 구하도록 한다.

|  |
| --- |
| import numpy as np  sales['year'] = sales['datesold'].dt.year  result = np.round(sales.groupby('year')['price'].agg(np.mean), 1)  pd.DataFrame(result.head()).T |



이번에는 선 그래프 시각화를 구현한다.

|  |
| --- |
| fig, ax = plt.subplots(figsize=(10, 6))  ax.plot(result.index, result.values)  ax.set\_title('Avg. House Price per Year With Matplotlib', size = 16)  ax.set\_xlabel('Year', size = 14)  ax.set\_ylabel('Price', size = 14, labelpad=12)  plt.savefig('output/matplotlib01.png', dpi=200)  plt.show() |
| 차트이(가) 표시된 사진  자동 생성된 설명 |

위 코드에 대해서 구체적으로 설명하면 다음과 같다.[[18]](#footnote-18)

* fig, ax = plt.subplots(figsize=(10, 6))가 의미하는 것은 Matplotlib에서 그래프를 작성할 때는 기본적으로 Figure 객체가 Axes객체가 필요하다.
* ax.plot(result.index, result.values): ax 객체의 plot() 메서드를 사용하여 그래프를 그린다. 이 때, result.index는 x축 값, result.values는 y축 값으로 사용된다.
* ax.set\_title('Avg. House Price per Year With Matplotlib', size=16): ax 객체의 set\_title() 메서드를 사용하여 그래프의 제목을 지정하고. size는 제목의 크기를 지정한다.
* ax.set\_xlabel('Year', size=14): ax 객체의 set\_xlabel() 메서드를 사용하여 x축의 레이블을 지정한다.
* ax.set\_ylabel('Price', size=14, labelpad=12): ax 객체의 set\_ylabel() 메서드를 사용하여 y축의 레이블을 지정합니다. labelpad는 레이블과 축 사이의 간격을 지정한다.
* plt.savefig('output/matplotlib01.png', dpi=200): 그래프를 이미지 파일로 저장한다. dpi는 저장된 이미지의 해상도를 지정한다
* plt.show(): 그래프를 보여준다.

### 막대 그래프(Bar Plot)

막대 그래프 범주형 데이터를 직사각형 막대로 나타내는 그래프 유형으로, 각 막대의 길이나 높이는 해당 범주의 양 또는 빈도를 나타낸다. 막대 그래프는 서로 다른 범주의 값 비교 또는 범주의 값이 시간에 따라 어떻게 변화하는지 추적하는 데 사용됩니다. 데이터 시각화에서 일반적으로 사용되며 데이터를 간단하고 쉽게 이해할 수 있는 방식으로 표시한다. 현재 주어진 데이터에서 막대 그래프를 그리기 위해 간단하게 데이터를 가공하도록 한다. 여기에서 2007년과 2008년 데이터만 추출하고, 각 월별 평균가격을 집계하도록 한다.

|  |
| --- |
| import numpy as np  import seaborn as sns  sales['year'] = sales['datesold'].dt.year  sales['month'] = sales['datesold'].dt.month  # 2007, 2008 추출  df = sales.copy()  df = df[df['year'].isin([2007, 2008])]  result = np.round(df.groupby(['year', 'month'])['price'].agg(np.mean), 1)  result.reset\_index().iloc[::3, :] |



위 결괏값을 토대로 연도별 그래프를 작성한다. 우선, result에 reset\_index()를 적용하여 새로운 데이터로 저장한다.

|  |
| --- |
| result = result.reset\_index()  result['year'].unique()  [결과]  array([2007, 2008], dtype=int64) |

이 정리된 데이터를 matplotlib를 적용하여 막대그래프를 작성하도록 한다.

|  |
| --- |
| fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(10, 6))  for i, y in enumerate(result['year'].unique()):  year\_data = result[result['year'] == y]  ax[i].bar(year\_data['month'], year\_data['price'])  ax[i].set\_title('Year {}'.format(y), fontsize=14)  ax[i].set\_xlabel('Month', fontsize=12)  ax[i].set\_ylabel('Price', fontsize=12)  plt.tight\_layout()  plt.savefig('output/matplotlib02.png', dpi=200)  plt.show() |
| 차트이(가) 표시된 사진  자동 생성된 설명 |

코드 설명을 하면 다음과 같다.

* plt.subplots() 함수를 사용하여 그림 객체와 2x1 grid의 subplot 축을 생성한다.
* for i, y in enumerate(result['year'].unique()): 코드는 result DataFrame의 year 열의 고유한 값에 대해 반복문을 실행한다. 이를 통해서 각 연도마다 하나의 subplot이 만들어진다.
* result[result['year'] == y] 구문을 사용하여 현재 연도에 해당하는 행만 선택하여 year\_data 변수에 저장한다.
* ax[i].bar() 메서드를 사용하여 현재 연도의 각 월에 대한 price 값을 막대 그래프로 나타낸다.

### 박스플롯(Boxplot)

박스플롯은 데이터의 분포와 이상치outliers를 시각화하는 그래프이다. 주로 수치형 변수의 분포를 살펴볼 때 사용된다. 박스플롯은 다섯 가지 요약 통계량(최소값, 1분위 값(25%), 중간값(50%), 3분위 값(75%), 최대값)을 이용하여 그려진다. 이미지를 통해 확인하면 아래와 같다.[[19]](#footnote-19)

차트이(가) 표시된 사진

자동 생성된 설명

이러한 요약 통계량은 상자(box) 안에 표시되며, 상자의 하단부터 차례로 1분위 값, 중간값, 3분위값까지 그린다. 상자 위와 아래에 있는 선은 각각 최소값과 최대값을 나타낸다. 이상치는 상자와 선의 범위를 벗어나 있는 값들로, 일반적으로 1.5\*IQRInterquartile Range 이상 벗어나면 이상치로 판단한다. 박스플롯은 데이터의 분포와 이상치를 시각화하여 데이터의 중심 경향성과 분산정도를 파악하는 데 도움을 준다. 또한 다른 변수와의 관계를 파악하는데도 유용하게 사용된다. 예를 들어, 그룹별 박스플롯을 그리면 그룹 간 분포의 차이나 중심 경향성의 차이를 한눈에 파악할 수 있다. 박스플롯 그래프를 그리기 위해 price 가격에서 1,000,000 이하의 데이터만 추출하도록 한다.

|  |
| --- |
| sales = pd.read\_csv('data/raw\_sales.csv', parse\_dates=['datesold'])  sales['year'] = sales['datesold'].dt.year  sales['month'] = sales['datesold'].dt.month  sales = sales.loc[sales['price'] < 1000000].reset\_index(drop=True)  sales.head(3) |

테이블이(가) 표시된 사진

자동 생성된 설명

propertyType에 있는 값은 두개(house, unit)로 확인할 수 있다.

|  |
| --- |
| sales['propertyType'].unique()  [결과]  array(['house', 'unit'], dtype=object) |

이제 박스플롯 그래프를 그려보면 아래와 같이 나타나게 된다.

|  |
| --- |
| from matplotlib.ticker import ScalarFormatter  formatter = ScalarFormatter()  formatter.set\_scientific(False)  house\_df = sales[sales['propertyType'] == 'house']['price']  unit\_df = sales[sales['propertyType'] == 'unit']['price']  fig, ax = plt.subplots(figsize=(10, 6))  ax.boxplot([house\_df, unit\_df])  ax.set\_xticklabels(['House', 'Unit'])  ax.set\_xlabel('Property Type', fontsize=12)  ax.set\_ylabel('Price', fontsize=12)  ax.set\_title('Prices of Houses and Units', fontsize=14)  ax.yaxis.set\_major\_formatter(formatter)  plt.savefig('output/matplotlib03.png', dpi=200)  plt.show() |
|  |

House와 Unit의 가격을 박스 플롯으로 시각화하는 코드이다. 코드에 대한 설명은 아래와 같이 작업할 수 있다.

* ‘from matplotlib.ticker import ScalarFormatter’[[20]](#footnote-20)는 축의 눈금 레이블을 포맷하는데 사용된다. 특히 과학적 표기법scientific notation으로 표시되는 숫자를 일반적인 숫자로 변환하는 데 사용한다. 이를 통해 그래프에서 축의 숫자가 더 읽기 쉽고 이해하기 쉬운 형식으로 표시할 수 있다. 여기서 설정한 변수는 추후 ax.yaxis.set\_major\_formatter()에 대입한다.
* sales 데이터프레임에서 ‘propertyType’이 ‘house’와 ‘unit’별로 price데이터를 각각 가져온다.
* ax.boxplot([house\_df, unit\_df]) house\_df와 unit\_df 데이터프레임을 사용하여 박스 플롯을 그린다. 특히, ax.boxplot()안에 리스트로 각각의 데이터를 넣어야 한다는 것에 주의한다.
* ax.set\_xticklabels(['House', 'Unit']) x-축 레이블을 "House"와 "Unit"으로 설정한다.

## Seaborn

Seaborn 라이브러리를 기본적으로 Matplotlib의 객체를 포함하고 있기 때문에, Streamlit 라이브러리에서 maplotlib 객체로 사용이 가능하다. 따라서, 이번에는 Seaborn 라이브러리로 그래프를 그려보도록 한다.

### 선 그래프(Line Chart)

먼저, 2008년과 2018년의 데이터만 추출하고, 연도, 월별의 평균 가격을 조회하는 코드를 작성하도록 한다. 그리고 이를 result로 저장한다.

|  |
| --- |
| import numpy as np  import seaborn as sns  import pandas as pd  sales = pd.read\_csv('data/raw\_sales.csv', parse\_dates=['datesold'])  sales['year'] = sales['datesold'].dt.year  sales['month'] = sales['datesold'].dt.month  # 2007, 2008 추출  df = sales.copy()  df = df[df['year'].isin([2008, 2018])]  result = np.round(df.groupby(['year', 'month'])['price'].agg(np.mean), 1).reset\_index()  result.head(12) |



이번에는 2018년 데이터만 조회한다.

|  |
| --- |
| result.tail(12) |



전반적인 코드는 matplotlib와 크게 다른 것이 없다. 다만, seaborn의 편리성은 lineplot() 메서드안에 있는 hue라는 객체를 사용할 때 알 수 있다. 일반적으로 두개 연도별로 그래프를 작성하려면 ax.plot()를 두개 그려야하고, 각 연도별로 데이터도 별도로 추출해야 하지만, seaborn에서는 그러한 과정을 생략하는 것을 알 수 있다.

|  |
| --- |
| import matplotlib.pyplot as plt  from matplotlib.ticker import ScalarFormatter  formatter = ScalarFormatter()  formatter.set\_scientific(False)  fig, ax = plt.subplots(figsize=(10, 6))  sns.lineplot(data=result, x=result.month, y=result.price, hue=result.year, ax=ax)  ax.set\_title('Avg. House Price per Year With Seaborn', size = 16)  ax.set\_xlabel('Year', size = 14)  ax.set\_ylabel('Price', size = 14, labelpad=12)  ax.yaxis.set\_major\_formatter(formatter)  plt.savefig('output/seaborn01.png', dpi=200)  plt.show() |
| 차트이(가) 표시된 사진  자동 생성된 설명 |

### 막대 그래프(Bar Plot)

seaborn에는 다양한 종류의 barplot을 그릴 수 있다. 기본적으로 총 4개의 그래프가 존재하는데, 아래 예시를 통해 우선 확인해본다. seaborn 라이브러리에 존재하는 tips 데이터셋을 가져와서 간단하게 데이터를 확인한다.[[21]](#footnote-21)

|  |
| --- |
| import seaborn as sns  # 데이터 가져오기  tips = sns.load\_dataset("tips")  tips.head() |



이번에는 seaborn내에 존재하는 다양한 막대 그래프를 그려보도록 한다.

|  |
| --- |
| import matplotlib.pyplot as plt  fig, ax = plt.subplots(nrows = 2, ncols = 2, figsize=(10, 6))  sns.barplot(x="day", y="total\_bill", data=tips, ax = ax[0, 0])  ax[0, 0].set\_title('barplot()')  sns.countplot(x="day", data=tips, ax = ax[0, 1])  ax[0, 1].set\_title('countplot()')  sns.stripplot(x="day", y="total\_bill", data=tips, ax = ax[1, 0])  ax[1, 0].set\_title('stripplot()')  sns.pointplot(x="day", y="total\_bill", data=tips, ax = ax[1, 1])  ax[1, 1].set\_title('pointplot()')  plt.tight\_layout()  plt.savefig('output/seaborn02.png', dpi=200)  plt.show() |
| 차트이(가) 표시된 사진  자동 생성된 설명 |

위 그림에서 보는 것처럼 각 함수마다 결과는 다르게 나타나기 때문에, 각 사용자가 목적에 맞게 사용하면 된다. 위 4개의 그래프에 대한 개별적인 설명은 아래와 같이 정리할 수 있다.

* sns.barplot()[[22]](#footnote-22): 이 함수는 범주형 변수의 각 범주(category)에 대한 수치형 변수의 평균을 수직 막대그래프(vertical bar plot)로 그리는 것이다. 이 때, 각 막대의 높이는 각 범주에서의 평균을 보여주는 것이며, 신뢰구간까지 보여준다. 즉, 이 함수를 통해서 각 범주 간 수치형 변수의 평균을 비교하는데 유용하다.
* sns.countplot()[[23]](#footnote-23): 이 함수는 범주형 변수의 각 범주에 대한 관측치의 개수를 수직 막대그래프로 보여준다. 각 막대의 높이는 각 범주에서의 관측치의 개수에 해당한다.
* sns.stripplot()[[24]](#footnote-24): 이 함수는 수치형 변수를 범주형 변수의 축에 따라 데이터 점으로 나타낸다. 이 함수는 범주형 변수의 각 범주에서의 수치형 변수의 분포를 보여주는 데 유용하다.
* sns.pointplot(): 이 함수는 수치형 변수를 범주형 변수의 축에 따라 수치 변수의 중심 경향성을 나타내고, 오차 막대를 사용하여 추정치 주변의 불확실성을 표시하는 시각화 기법이다. 하나 이상의 범주형 변수의 여러 수준을 비교하고, 각 변수 간의 교호작용을 표시하는데 유용하다. 공식 문서에서는 pointplot()이 barplot()보다 범주형 변수의 여러 수준을 비교할 때 더 유용하다고 말한다. 막대의 높이는 평균값을 의미한다.

pointplot() 함수 내부의 다양한 파라미터를 적용하여 그래프를 꾸며볼 수 있다. 각 파라미터의 설명은 다음과 같다. 먼저 x축은 ‘day', y축은 'tip'으로 설정한다. hue 파라미터를 이용하여 'sex' 칼럼을 기준으로 여성과 남성으로 그룹화한다. dodge 파라미터는 여러 개의 그룹이 있을 때 겹치지 않도록 간격을 조정하는 값이며, 소수점으로 변경할 수 있다. capsize 파라미터는 에러 바의 끝에 너비의 길이를 정하는 값이다. errorbar 파라미터는 ci, pi, se, sd 중 하나를 선택하거나 또는 튜플 형태로 작성할 수 있다. 또는 errorbar를 숨길수도 있다(None).

|  |
| --- |
| fig, ax = plt.subplots(figsize=(10, 6))  sns.pointplot(data=tips, x='day', y='tip', hue='sex',  dodge=0.25, capsize=0.25, errorbar='sd', ax = ax)  plt.savefig('output/seaborn03.png', dpi=200)  plt.show() |
| 차트이(가) 표시된 사진  자동 생성된 설명 |

본 데이터셋에 적용해서 house와 unit의 errobar를 적용해서 살펴보도록 한다. 이번에는 2007~2010년까지의 데이터만 추출하도록 한다.

|  |
| --- |
| import numpy as np  import seaborn as sns  import pandas as pd  sales = pd.read\_csv('data/raw\_sales.csv', parse\_dates=['datesold'])  sales['year'] = sales['datesold'].dt.year  sales['month'] = sales['datesold'].dt.month  # 2007 ~ 2010추출  df = sales.copy()  df = df[df['year'].isin([2007, 2008, 2009, 2010])]  df.head() |

테이블이(가) 표시된 사진

자동 생성된 설명

|  |
| --- |
| df['year'].unique()  [결과]  array([2007, 2008, 2009, 2010], dtype=int64) |

위 가공된 데이터에서, x축은 year, y축은 price, 그리고 hue는 propertyType을 설정하면, year, 그리고 propertyType으로 비교분석이 가능함을 확인할 수 있다. 그래프를 통해 확인하도록 한다.

|  |
| --- |
| fig, ax = plt.subplots(figsize=(10, 6))  sns.pointplot(data=df, x='year', y='price', hue='propertyType',  dodge=0.25, capsize=0.25, errorbar='sd', ax = ax)  plt.savefig('output/seaborn04.png', dpi=200)  plt.show() |
|  |

위 그래프를 보면, house의 평균가격은 항상 unit보다 높았다. 그런데, 연도별 추이를 보면, 2007년에서 2008년 사이의 평균가격을 살펴보면 unit의 하락폭이 house의 하락폭보다 더 크다는 것을 시각적으로 알 수 있다. 또한, 전체적인 그래프의 변동성을 보면 house보다는 unit의 변동성이 더 크며, 이를 통해 unit의 가격이 house의 가격보다 경기에 더 민감하다고 추정할 수 있다.

### 박스플롯(Boxplot)

seaborn에서 박스플롯은 일종의 분포를 확인하고자 하는 것이다. seaborn에서 박스플롯을 작성할 때는 x축만 사용할 수도 있고, x축과 y축만 사용할 수도 있다. 우선 price가격의 전체적인 boxplot을 작성하면 다음과 같다. 여기에서 핵심이 되는 코드는 sns.boxplot(x=df["price"], ax = ax)이다. 기존 문법과 다르게, x축에 series 형태로 입력한 것에 유의한다.

|  |
| --- |
| import seaborn as sns  import pandas as pd  import matplotlib.pyplot as plt  from matplotlib.ticker import ScalarFormatter  formatter = ScalarFormatter()  formatter.set\_scientific(False)  sales = pd.read\_csv('data/raw\_sales.csv', parse\_dates=['datesold'])  sales['year'] = sales['datesold'].dt.year  sales['month'] = sales['datesold'].dt.month  # 2007 ~ 2010추출  df = sales.copy()  df = df[df['year'].isin([2007, 2008, 2009, 2010])]  fig, ax = plt.subplots(figsize=(10, 6))  sns.boxplot(x=df["price"], ax = ax)  ax.set\_xticks(ax.get\_xticks())  print(ax.get\_xticks())  ax.set\_xticklabels(ax.get\_xticks(), rotation=30)  ax.xaxis.set\_major\_formatter(formatter)  plt.savefig('output/seaborn05.png', dpi=200)  plt.show()  [결과]  [-1000000. 0. 1000000. 2000000. 3000000. 4000000. 5000000.  6000000. 7000000. 8000000.] |
|  |

* ax.get\_xticks()를 출력하면, x축의 라벨을 리스트로 가져올 수 있다.
* ax.set\_xticklabels(ax.get\_xticks(), rotation=30) : x축의 라벨을 설정한 후, 각 라벨의 값을 rotation에 맞춰서 방향을 변형할 수 있다.

이번에는 x축은 year, y축은 price, hue는 propertyType을 지정하면 새로운 연도별 박스플롯을 그릴 수 있다.

|  |
| --- |
| # 2007 ~ 2010추출  df = sales.copy()  df = df[df['year'].isin([2007, 2008, 2009, 2010])]  fig, ax = plt.subplots(figsize=(10, 6))  sns.boxplot(data = df, x = 'year', y = 'price',  hue = 'propertyType', ax = ax)  ax.yaxis.set\_major\_formatter(formatter)  plt.savefig('output/seaborn06.png', dpi=200)  plt.show() |
|  |

위 그래프에서 보면, 2010년에 700000이상의 데이터가 존재한다. 일반적으로 이런 데이터가 존재하면 전체적인 분포의 상태를 파악하기 어렵기 때문에 해당 데이터를 제거하고 다시 그래프를 작성할 수 있다. 앞선 그래프보다 더 각 연도별 박스플롯의 그래프가 보다 더 자세히 나타나는 것을 확인할 수 있다.

|  |
| --- |
| # 2007 ~ 2010추출  df2 = df[df['price'] <= 7000000]  fig, ax = plt.subplots(figsize=(10, 6))  sns.boxplot(data = df2, x = 'year', y = 'price',  hue = 'propertyType', ax = ax)  ax.yaxis.set\_major\_formatter(formatter)  plt.savefig('output/seaborn07.png', dpi=200)  plt.show() |
|  |

## Plotly

Plotly 라이브러리는… .

|  |
| --- |
| import pandas as pd  import plotly  import numpy as np  import plotly.express as px  sales = pd.read\_csv('data/raw\_sales.csv', parse\_dates=['datesold'])  sales['year'] = sales['datesold'].dt.year  result = sales.groupby('year')['price'].agg(np.mean)  fig = px.line(result, x=result.index, y=result.values,  title='Avg. House Price per Year With Plotly')  fig.show() |
| 차트이(가) 표시된 사진  자동 생성된 설명 |

## Scikit-Learn

# Chapter 4. Streamlit

## Slider bar

## Select bar

## Tab

## Checkbox

# Chapter 5. 공공데이터 수집

## 서울열린데이터광장

## 공공데이터포털

# Chapter 6. 부동산 실거래가 대시보드

## 데이터 수집

## Home 화면 꾸미기

## 탐색적 자료분석 화면 꾸미기

## 머신러닝 모형 만들기

## Prediction 화면 꾸미기

## OpenAI 챗봇 구현하기

# Chapter 7. 배포

## secrets.toml

## 배포

## 마무리

# 인용 자료

Jaiswal, S. (2017, Dec). *Python Data Structures Tutorial*. Retrieved from DataCamp: https://www.datacamp.com/tutorial/data-structures-python

1. Jaiswal, S. (2017, Dec). *Python Data Structures Tutorial*. Retrieved from DataCamp: https://www.datacamp.com/tutorial/data-structures-python [↑](#footnote-ref-1)
2. Python 코딩테스트에서는 Stacks, Queues, Graphs, Trees와 같은 개념을 알아야 풀 수 있는 문제들이 있어서, 해당 문법들을 공부할 필요가 있다. 본 책에서는 다루지 않는다. [↑](#footnote-ref-2)
3. 아래 표에서 X와 Y는 모두 조건식을 의미하며, 조건식의 결과가 True 또는 False인 상태를 의미한다. [↑](#footnote-ref-3)
4. https://www.learnbyexample.org/wp-content/uploads/python/String-Indexing.png [↑](#footnote-ref-4)
5. https://docs.python.org/3/library/stdtypes.html#string-methods [↑](#footnote-ref-5)
6. List와 Tuple, Dictionary와 같은 기초 문법은 다음 페이지부터 순차적으로 다룬다. [↑](#footnote-ref-6)
7. 더 읽어보기 : https://docs.python.org/3/library/collections.abc.html [↑](#footnote-ref-7)
8. 일반적으로 리스트에는 파이썬에서 존재하는 다양한 값을 저장할 수 있다. 숫자, 문자열 뿐만 아니라, 아직 배우지 않은 데이터프레임 객체, 함수, 클래스 등도 넣을 수 있어서, 활용범위가 매우 무궁무진하다. [↑](#footnote-ref-8)
9. https://www.programiz.com/python-programming/tuple [↑](#footnote-ref-9)
10. https://scaler.com/topics/images/list-comprehenstion.webp [↑](#footnote-ref-10)
11. Docstring에 관한 자세한 설명은 <https://peps.python.org/pep-0257/> 참고한다. [↑](#footnote-ref-11)
12. https://pandas.pydata.org/ [↑](#footnote-ref-12)
13. Boston 데이터셋의 정보는 <http://lib.stat.cmu.edu/datasets/boston> 에서 확인한다. 본 장에서는 독자가 해당 컬럼들의 정보를 이해했다는 가정에서 기술하였다. [↑](#footnote-ref-13)
14. https://pandas.pydata.org/docs/reference/api/pandas.DatetimeIndex.html [↑](#footnote-ref-14)
15. https://www.kaggle.com/datasets/htagholdings/property-sales?resource=download [↑](#footnote-ref-15)
16. https://www.w3resource.com/python-exercises/pandas/groupby/index.php [↑](#footnote-ref-16)
17. https://docs.streamlit.io/library/api-reference/charts/st.pyplot [↑](#footnote-ref-17)
18. https://delftswa.gitbooks.io/desosa-2017/content/matplotlib/chapter.html [↑](#footnote-ref-18)
19. https://itwiki.kr/w/박스\_플롯#/media/파일:Boxplot2.png [↑](#footnote-ref-19)
20. https://matplotlib.org/stable/api/ticker\_api.html#matplotlib.ticker.ScalarFormatter [↑](#footnote-ref-20)
21. tips 데이터셋 설명: https://rdrr.io/cran/reshape2/man/tips.html [↑](#footnote-ref-21)
22. sns.barplot() : https://seaborn.pydata.org/generated/seaborn.barplot.html [↑](#footnote-ref-22)
23. sns.countplot() : https://seaborn.pydata.org/generated/seaborn.countplot.html [↑](#footnote-ref-23)
24. sns.stripplot(): https://seaborn.pydata.org/generated/seaborn.stripplot.html [↑](#footnote-ref-24)