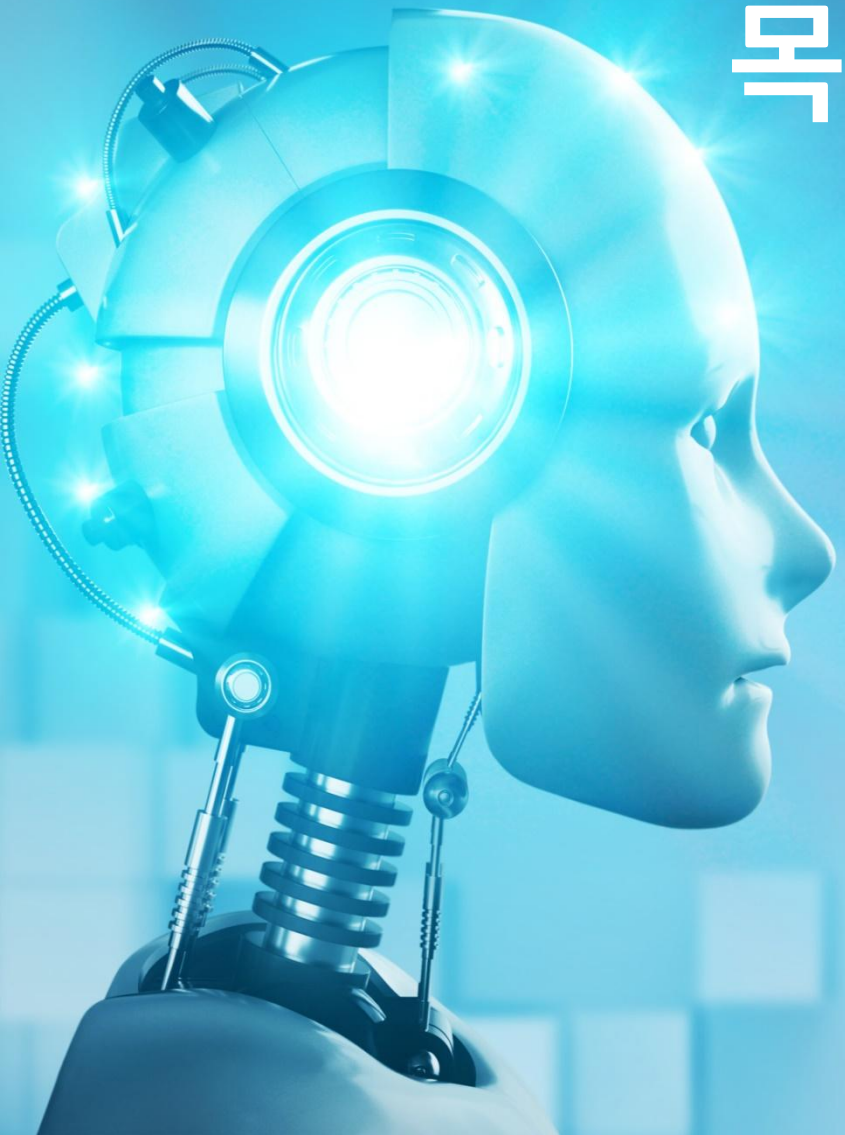
인공지능(AI) 개발자 양성 과정

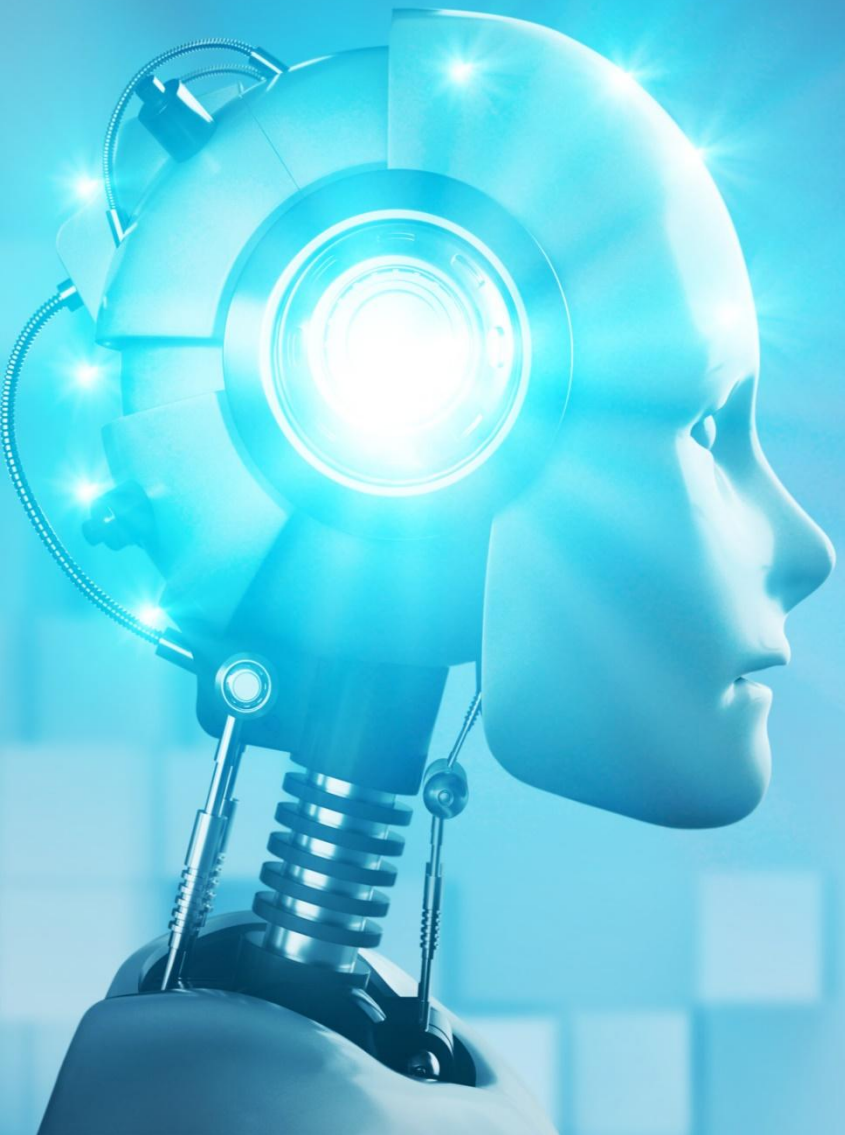
2차시

머신리닝을 위한 파이썬 기초

휴먼교육센터 안홍근 강사



목차 Contents



I 기초 파이썬 프로그래밍

1. 파이썬 기본문법
2. 함수
3. 메서드
4. 예외처리
5. 자원관리 자동화
6. 데이터 분석 및 자동화함수
7. 클래스(class)

들어가기 전에

파이썬과 자바 주요 차이점

특성	Python	Java
실행방식	인터프리터 방식 (코드를 한 줄씩 바로 해석하고 실행) cf. 동시통역가 역할과 유사	컴파일 방식 (전체 코드를 먼저 번역한 후 실행) cf. 책을 전부 번역한 후 읽는 것과 유사
문법	간단하고 영어 문장과 유사, 들여쓰기(콜론 :)로 코드 블록 구분	중괄호({})로 코드 블록 구분, 세미콜론(;)으로 문장 끝맺음, 엄격한 규칙
속도	상대적으로 느림	일반적으로 더 빠름
타입 시스템	동적 타입(변수 타입을 자유롭게 변경 가능)	정적 타입(변수의 타입이 고정됨)
메모리 관리	자동 (가비지 컬렉션)	자동 (가비지 컬렉션)
주요 용도	데이터 분석, 인공지능, 웹 개발, 자동화 스크립트	기업용 소프트웨어, 대규모 시스템
코드 작성 속도	빠름 (적은 코드로 많은 기능 구현 가능)	상대적으로 느림(더 많은 코드 필요)

1. 파이썬 기본문법

1.1 기본문법 요약 (1/2)

항목	요소	정의	설명	예시
변수		데이터를 저장하는 메모리 공간	숫자, 문자	x = 5 x = "사과"
자료형	정수형 (int)	소수점이 없는 숫자	산술 연산 가능	x = 5
	실수형 (float)	소수점이 있는 숫자	근사값 처리	x = 3.14
	문자열 (str)	문자들의 집합	큰 따옴표 또는 작은 따옴표	s = "사과" 또는 s = '사과 '
	불린 (bool)	참/거짓 값	True/False 만 가능	b = True
	리스트 (list)	순서가 있는 집합	수정 가능, 중복 허용	lst = [1, 2, 3]
	튜플 (tuple)	순서가 있는 집합	수정 불가, 중복 허용	tup = (1, 2, 3)
	딕서너리 (dict)	키-값 쌍의 집합	키는 중복 불가	d = {"a" : 1}

1. 파이썬 기본문법

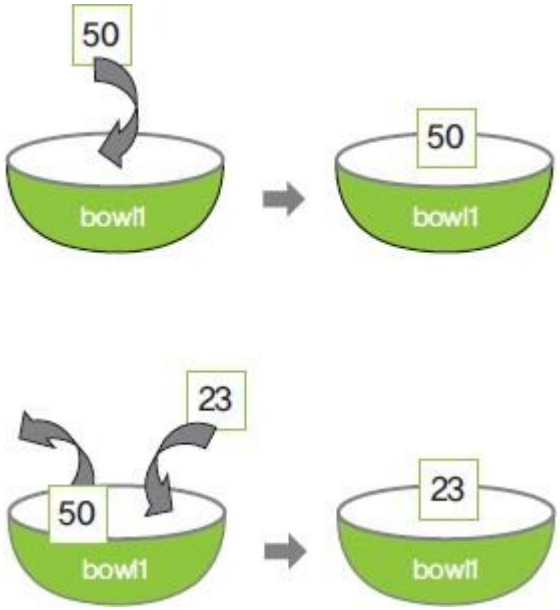
1.1 기본문법 요약 (2/2)

항목	요소	정의	설명	예시
조건문	if	조건에 따른 실행	조건이 참일 때 실행	if x > 0
	elif	다중 조건 실행	이전 조건이 거짓일 때 확인	elif x == 0:
	else	기본 실행	모든 조건이 거짓일 때 실행	else:
반복문	for	정해진 횟수 반복	시퀀스 객체 순회	for I in range(3):
	while	조건부 반복	조건이 참인 동안 반복	while x > 0:
	제어문	반복 제어	break, continue 사용	break
함수	기본함수	코드의 재사용 단위	def로 정의, 반환값 설정	def add(a, b):
	람다함수	익명 함수	단일 표현식 함수	lambda x: x*2
	내장함수	파이썬 기본 제공	print(), len() 등	print("hello")
메서드	리스트 메서드	리스트 조작 함수	append, remove, sort 등	lst.append(4)
	문자열 메서드	문자열 처리 함수	upper, lower, split 등	s.upper()
예외처리	try	예외 발생 가능 코드	오류 검사 블록	try:
	except	예외 처리 코드	오류 발생시 실행	except:
	finally	항상 실행 코드	예외 발생 여부와 무관	finally:

1.2 변수 ▶ 1.2.1 개요(1/2)

- 변수는 어떠한 값을 저장하는 메모리 공간 (ex. 그릇에 비유)
- 컴퓨터에 값을 담아두는 공간을 만들고 그 장소에 이름을 붙여 사용하는 것

변수에 대한 정의



물건을 담는 그릇처럼 값을 저장하는 공간 개념

변수 타입

- ✓ 정수형, 실수형, 문자형, 불린형이 대표적
- ✓ 변수 타입별로 그릇 모양이 다른 것처럼 쓰임새가 다름 (ex. 정수형에는 정수만 등록)
- ✓ 최초 선언한 변수 타입과 다른 값을 넣을 시에 에러가 발생
- ✓ 타입 변경이 필요할 경우 타입변경 함수 사용

1.2 변수 ▶ 1.2.1 개요(2/2)

- 파이썬은 자바와 달리 변수 선언 시 자료형을 명시하지 않음
- 파이썬은 동적 타이핑 언어이기 때문에, 변수에 값을 할당할 때 자동으로 자료형이 결정

파이썬 변수 선언 방식

- a = 10 (정수형 변수)
- b = “홍길동“ (문자열 변수)
- c = 3.14 (실수형 변수)
- d = True (불린형 변수)

✓ 변수 선언 시 자료형을 신경쓰지 않아도 됨
✓ 변수에 다른 자료형의 값을 다시 할당할 수 있음
✓ 주의사항 : 코드 실행 중에 변수의 자료형이 변경 시, 예상치 못한 오류 발생 가능성

자바 변수 선언 방식

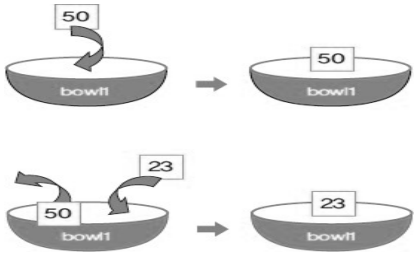
- int a; (정수형 변수)
- String b = “홍길동”; (문자열 변수)
- double c = 3.14; (실수형 변수)
- Boolean d = true; (불린형 변수)

✓ 컴파일 시점에 자료형 오류를 검출하여 안정적인 코드 작성을 도움
✓ 변수의 자료형을 명시적으로 확인할 수 있어 코드 가독성을 높임
✓ 파이썬에 비해 변수를 선언할 때 자료형을 명시해야 하므로 코드의 길이가 길어짐

1.2 변수 ▶ 1.2.2 변수의 사용 예시

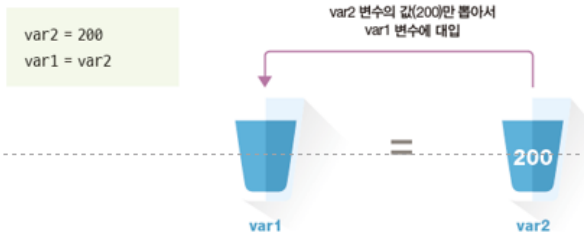
변수 값의 변경

- 최초 변수에 넣은 값을 인식함 (ex. a = 50)
- 변수에 새로운 값을 넣으면 기존값은 변경됨 (ex. a=50 → a = 23)



다른 변수의 값을 저장

- 다른 변수의 값을 저장할 수 있음
- var2 = 200
- var1 = var2 → var1 = 200



계산 결과 저장

- 계산 결과를 저장할 수 있음
- var1 = 100 + 100 → var1 = 200

숫자와 변수의 연산 값 저장

- 변수에는 숫자와 변수의 연산을 넣을 수 있음
- var2 = 200
- var1 = var2 + 100 → var1 = 300

1.2 변수 ▶ 1.2.2 변수 만드는 규칙

☑ 대소문자 구분
- 파이썬에서는 대소문자를 달리 구분함
- num, NUM 두개의 변수는 전혀 다른 변수

☑ 숫자로 시작 불가
- 문자, 숫자, 언더바(_)를 포함 할 수 있으나 숫자를 앞에 사용할 수 없음
- var2(O), _var(O), var_2(O), 2Var(X)

☑ 파이썬 키워드 사용 불가
- 파이썬 명령문이나 내장함수를 사용할 수 없음
- if, while, for, append, sum, print 등 사용불가 : ex. if = 10 (X), sum = 50 (X)

☑ 명확하고 의미있는 이름 사용
- 의미를 잘 전달할 수 있는 이름 선택
- 짧고 명확한 단어 선택이 좋음 : ex. a =10 보다 age = 10 이 좋음

1.3 자료형 ▶ 1.3.1 숫자형

숫자형

- 파이썬의 숫자형은 다양한 수학적 연산을 지원하며 , 정수형과 실수형으로 나눌수 있음
- 복합 연산자(+=, -+, *=, /= 등)를 사용하여 코드를 더 간결하게 작성 가능

정수형 (Integer)

- 정의 : 숫자 중에서 소수점이 없는 정수를 나타내는 자료형
- 특징 : 파이썬에서는 정수의 크기에 제한이 없으며 원하는 크기의 정수 사용이 가능

[예시]

```
num1 = 10   # 양의 정수
num2 = -20  # 음의 정수
print(num1, num2)  # 출력: 10 -20
```

실수형 (floating point)

- 정의 : 소수점을 포함하는 실수 값을 나타냄
- 특징 : float는 유한한 크기의 메모리에서 부동 소수점 연산을 지원

[예시]

```
num1 = 3.14   # 양의 실수
num2 = -2.718   # 음의 실수
print(num1, num2) # 출력: 3.14 -2.718
```

1.3 자료형 ▶ 1.3.2 문자열 (1/5)

문자열(string)

- 문자의 집합을 나타내는 데이터 타입으로 작은 따옴표(‘ ’) 또는 큰 따옴표(“ ”) 로 문자를 감싸서 표현함
- 문자열은 불변하며, 각 문자는 인덱스를 통해 접근 가능

문자열 생성

[예시]

```
str1 = 'Hello’
str2 = "Python"
print(str1, str2) # 출력: Hello Python
```

멀티라인 문자열

- 멀티라인 문자열은 세 개의 작은따옴표(‘’’ ‘’’) 또는 큰따옴표(“”” “””)를 사용하여 여러 줄에 걸친 문자열을 생성할 수 있음

[예시]

```
str3 = '''This is
a multi-line
string.'''
print(str3)
```

인덱싱

- 문자열의 각 문자는 0부터 시작하는 인덱스를 통해 접근 가능

[예시]

```
text = "Python"
print(text[0]) # 출력: P
print(text[-1]) # 출력: n (뒤에서 첫 번째 문자)
```

슬라이싱

- [start:end:step] 형식을 사용해 문자열의 일부를 추출할 수 있음

[예시]

```
text = "Python"
print(text[0:3]) # 출력: Pyt
print(text[::2]) # 출력: Pto (2칸씩 건너뜀)
```

1.3 자료형 ▶ 1.3.2 문자열 (3/5)

메서드 (문자변환)

- upper() : 모든 문자를 대문자로 변환
- lower() : 모든 문자를 소문자로 변환

[예시]

```
text = "Python"
print(text.upper()) # 출력: PYTHON
print(text.lower()) # 출력: python
```

메서드 (문자찾기)

- find() : 특정 문자의 인덱스 반환(없으면 -1 반환)
- count() : 특정 문자의 개수 세기

[예시]

```
text = "banana"
print(text.find('a')) # 출력: 1 (첫 번째 'a'의 위치)
print(text.count('a')) # 출력: 3
```

메서드

(공백 제거)

- strip() : 문자열 양쪽 끝 공백 제거
- lstrip(), rstrip() : 왼쪽 또는 오른쪽 공백만 제거

[예시]

```
text = " Hello World "
print(text.strip()) # 출력: "Hello World"
```

메서드

(문자열 결합 및 분할)

- split() : 특정 구분자로 문자열을 나누어 리스트로 반환
- join() : 리스트의 문자열을 특정 구분자로 연결하여 하나의 문자열로 반환

[예시]

```
text = "Python is fun"
words = text.split()  # 공백 기준으로 분할
print(words)  # 출력: ['Python', 'is', 'fun']
joined_text = ' '.join(words) # 공백을 구분자로 연결
print(joined_text)  # 출력: "Python is fun"
```

1.3 자료형 ▶ 1.3.2 문자열 (5/5)

포맷팅 (formatting)

- 파이썬은 문자열 내에 다양한 방법으로 변수를 삽입할 수 있음
- f- strings : 파이썬 3.6부터 도입된 간결한 문자열 포매팅 방식, 문자열 앞에 f를 붙여 사용하며, 중괄호 {} 안에 변수나 표현식을 직접 넣어 문자열에 포함시킬 수 있음

[예시: 파이썬 3.6이상, f-strings 방식]

```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

[예시: 구버전]

```
text = "My name is %s and I am %d years old." % (name, age)
print(text) # 출력: My name is Alice and I am 25 years old.
```

특수문자

- \n : 줄바꿈
- \t : 탭
- \\ : 백슬래시

[예시]

```
print("Hello\nPython")    # 출력: Hello (줄바꿈) Python
print(“탭을\t사용한\t예시")   # 출력: 탭을 사용한 예시
print("백슬래시는 이렇게 \\ 출력")  # 백슬래시는 이렇게 \  출력
```

불린(bool)

- 참(True)과 거짓(False) 두가지 값만을 갖는 데이터 유형
- 주로 조건문에서 조건의 참/거짓을 판별하는 용도로 사용

불린연산자

- and: 두 조건이 모두 참일 때 참, or: 둘 중 하나라도 참이면 참
- not: 조건의 값을 반전시킴

[예시]

```
a= True
b=False
print(a and b) # False
print(a or b) # True
print(not a) # False
```

불린활용 예제

```
is_authenticated = True
has_permission = False
if is_authenticated and has_permission:
    print("접근 허용")
else:
    print("접근 거부")
```

➔ 결과 : 접근 거부

1.3 자료형 ▶ 1.3.4 리스트 (1/3)

리스트(List)

- 여러 개의 값을 하나의 변수에 담을 수 있는 순서가 있는 데이터 자료형으로 **대괄호 [ ]** 안에 콤마(,)로 구분하여 정의
- **수정 가능**하고 **중복이 허용**되며 다양한 데이터 유형을 한 리스트에 담을 수 있음

리스트 생성과 초기화

[빈 리스트 생성]
- empty_list = []

[초기값 설정]
- numbers = [1, 2, 3, 4, 5]
- fruits = ["apple", "banana", "cherry"]

인덱스 접근 및 슬라이싱

[인덱스로 접근]
- fruits = ["apple", "banana", "cherry"]
- print(fruits[0]) # apple

[슬라이싱]
- fruits = ["apple", "banana", "cherry"]
- print(fruits[0:2]) # ['apple', 'banana']

리스트 활용예제

```
numbers = [3, 1, 4, 1, 5]

# 요소 추가 (append)
numbers.append(9)
print(numbers) # [3, 1, 4, 1, 5, 9]
# 요소 정렬 (sort)
numbers.sort()
print(numbers) # [1, 1, 3, 4, 5, 9]
# 요소 제거 (remove)
numbers.remove(1) # 첫 번째로 발견되는 1을 제거
print(numbers) # [1, 3, 4, 5, 9]
# 특정 위치의 요소 추출 및 삭제 (pop)
removed_item = numbers.pop(2) # 인덱스 2의 요소를 추출하고 삭제
print(numbers) # [1, 3, 5, 9]
print(removed_item) # 4
# 리스트 뒤집기 (reverse)
numbers.reverse()
print(numbers) # [9, 5, 3, 1]
```

리스트 활용예제

```
print(numbers)  # [9, 5, 3, 1]

# 리스트 요소 개수 세기 (count)
count = numbers.count(5)
print(count) # 1
# 리스트 복사 (copy)
new_numbers = numbers.copy()
print(new_numbers) # [9, 5, 3, 1]
# 리스트 확장 (extend)
numbers.extend([6, 7, 8])
print(numbers) # [9, 5, 3, 1, 6, 7, 8]
# 리스트 합치기 ( + 로 연결)
another_list = [10, 11, 12]
combined_list = numbers + another_list
print(combined_list) # [9, 5, 4, 3, 6, 7, 8, 10, 11, 12]
```

튜플(tuple)

- 여러 개의 값을 하나의 변수에 담을 수 있는 순서가 있는 데이터 자료형
- 리스트와는 달리 **수정할 수 없고 중복은 허용되며**, 튜플은 **소괄호( )** 안에 값을 콤마(,)로 구분하여 정의함

리스트 생성과 초기화

[빈 튜플 생성]
- empty_tuple = ()

[한 개의 요소를 가진 튜플 : 단일 요소일 때는 콤마가 필요]
- single_element = (5,)

[튜플의 불변성]
- my_tuple = (1, 2, 3, 4, 5) → my_tuple[0] = 10 # TypeError 발생

튜플 요소 접근 및 슬라이싱

[인덱스로 접근]
- colors = ("red", "green", "blue")
- print(colors[1]) # green

[슬라이싱]
- colors = ("red", "green", "blue")
- print(colors[0:2]) # ('red', 'green')

1.3 자료형 ▶ 1.3.5 튜플 (2/2)

튜플
활용예제

- **# 특정 값의 개수 세기 (count)**
- numbers = (1, 2, 3, 2, 4)
- print(numbers.count(2)) # 2 (2가 두 번 등장)
- **# 특정 값의 인덱스 찾기 (index)**
- print(numbers.index(3)) # 2 (3의 인덱스)

[예제]

- point = (3, 4)
- print("x 좌표:", point[0])
- print("y 좌표:", point[1])

➔ 결과 : x 좌표: 3, y 좌표: 4

1.3 자료형 ▶ 1.3.6 딕셔너리 (1/2)

딕셔너리(dict)

- 키(key)-값(value) 쌍으로 데이터를 저장하는 매핑 타입의 자료형
- **요소 추가, 수정, 삭제가 가능**하며, 표기는 **중괄호{}** 안에 key: value 형태로 값을 저장함

리스트 생성과 초기화

[빈 딕셔너리 생성]
- empty_tuple = {}

[초기값 설정]
- person = {"name": "Bob", "age": 30, "city": "New York"}

딕셔너리 요소 접근 및 슬라이싱

[키를 통해 값 접근]
- person = {"name": "Bob", "age": 30, "city": "New York"}
- print(person["name"]) # Bob

[값 수정]
- person["age"] = 31
- print(person["age"]) # 31

1.3 자료형 ▶ 1.3.6 딕셔너리 (2/2)

딕셔너리 활용예제

```
# 키에 대응하는 값 반환 (get)
info = {"name": "Charlie", "age": 22}
print(info.get("name")) # Charlie

# 모든 키를 반환 (keys)
print(info.keys()) # dict_keys(['name', 'age']

# 모든 값을 반환 (values)
print(info.values()) # dict_values(['Charlie', 22])

# 키-값 쌍을 튜플 형태로 반환 (items)
print(info.items()) # dict_items([('name', 'Charlie'), ('age', 22)])
```

1.4 조건문 ▶ 1.4.1 if문 (1/2)

if문

- 조건에 따라 코드 블록을 실행할 지 결정하는 제어문으로서 조건식이 True일 경우에만 특정 코드를 실행
- 조건식의 결과는 True 또는 False로 나타나야 하고, 코드 블록은 들여쓰기로 구분되고, 콜론(:)기호로 조건식의 끝을 표시

단순 if문

- 조건이 참일 때만 실행되는 단일 블록

[예시]

```
score = 85
if socre > 80:
    print("테스트에 합격하셨습니다.")
# 결과 : 테스트에 합격하셨습니다.
```

형식

```
if 조건식:
    실행문
```

if-else문

- 조건이 참일 때와 거짓일 때 각각 다른 코드로 실행

[예시]

```
number = 5
if number % 2 == 0:     # %는 나머지를 구하는 연산자
    print("짝수입니다.")
else:
    print("홀수입니다.")
# 결과 : 홀수입니다.
```

형식

```
if 조건식:
    실행문1
else:
    실행문2
```

1.4 조건문 ▶ 1.4.1 if문 (2/2)

if-elif-else

- 여러 조건 중 하나만 선택하여 실행

[예시]

```
score = 85
if score >= 90:
    print("A학점")
elif score >= 80:
    print("B학점")
else:
    print("C학점:")      # 결과  B학점
```

형식

```
if 조건식:
    실행문1
elif 조건식2:
    실행문2
else:
    실행문3
```

if문 활용 팁

- 조건식 간단화 : 복잡한 조건은 논리 연산자(and, or, not)를 사용
- 중첩 피하기 : 여러 조건은 elif로 나누어 가독성 향상
- 디버깅 편리 : print()문으로 조건 결과 확인

[예시]

```
x, y = 10, 20
if x > 5 and y < 30:
    print("조건을 만족합니다.")     # 결과 : 조건을 만족합니다.
```

1.5 반복문 ▶ 1.5.1 for문 (1/2)

for문

- 반복 가능한 객체(리스트, 튜플, 문자열 등)의 각 요소를 순회하며 코드를 실행하는 반복문
- 특정 작업을 반복적으로 수행할 때 사용하고, 반복 횟수를 명확히 제어 가능

기본 for문

- 반복 대상의 요소를 하나씩 순회하며 실행

[예시]

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num * 2)   # 결과 : 2, 4, 6, 8, 10
```

형식

```
for 변수 in 반복가능한 객체:
    실행문
```

range()와 함께 사용 시

- range() 함수로 반복 횟수를 지정
- range(시작값, 종료값, 증가값)

[예시]

```
for i in range(1, 6):
    print(f"현재 값: {i}")  # 결과 : 현재값 1, 현재값 2
                                ~ 현재값 5
```

형식

```
for 변수 in range():
    실행문
```

1.5 반복문 ▶ 1.5.1 for문 (2/2)

중첩 for문

- 반복문 안에 또 다른 반복문 포함
- 안쪽에 있는 for 문이 완전히 실행된 후에 바깥쪽 for문이 실행되면서 반복되는 구조

[예시]

```
for i in range(1, 4):
    for j in range(1, 3):
        print(f"i: {i}, j: {j}")
```

형식

```
for 변수1 in 반복객체1:
    for 변수2 in 반복개체2:
        실행문
```

enumerate() 함께 사용 시

- enumerate() 함수는 시퀀스(리스트, 튜플, 문자열 등)를 순회하면서 각요소의 **인덱스와 값을 튜플 형태로 함께 반환**하는 내장 함수
- 반복문을 돌면서 현재 어떤 인덱스의 값을 처리하고 있는 지 알고 싶을 때 용이

[예시]

```
items = ["A", "B", "C"]
for idx, item in enumerate(items):
    print(f"인덱스 {idx}: {item}")
```

형식

```
for 인덱스, 값 in enumerate():
    실행문
```

1.5 반복문 ▶ 1.5.2 while문 (1/3)

while문

- 조건이 참(True)인 동안 특정 코드를 반복 실행하는 반복문으로 **반복 횟수를 사전에 알 수 없을 때 주로 사용**
- **조건식의 값이 변하지 않으면 무한 루프**가 발생할 수 있음

기본 for문

- 종료 조건을 명시적으로 설정해야 안전
- **조건식이 False가 되면 반복 종료**

[예시]

```
count = 0
while count < 5:
    print(f"Count: {count}")
    count += 1
```

형식

```
while 조건식:
    실행문
```

무한 루프

- 종료 조건 없이 True를 조건식으로 설정
- **반드시 break문으로 종료 조건 설정**

[예시]

```
while True:
    user_input = input("종료하려면 'exit' 입력: ")
    if user_input == "exit":
        break
```

형식

```
while True:
    실행문
```

조건과 else 사용

- 조건이 False로 바뀌면 else 블록 실행

[예시]

```
x = 5
while x > 0:
    print(x)
    x -= 1
else:
    print("반복 종료")
```

형식

```
while 조건식:
    실행문
else:
    실행문
```

break & continue 사용

- continue : continue 아래의 코드는 실행되지 않고, 반복문의 조건 검사로 다시 돌아감
- break : break를 만나면 반복문을 벗어나게 되고 반복문 종료

[예시]

```
n = 0
while n < 10:
    n += 1
    if n % 2 == 0:
        continue    # 짝수는 출력 건너뜀
    print(n)
    if n == 7:
        break    # 7에서 반복 종료
```

1.5 반복문 ▶ 1.5.2 while문 (3/3)

input 함수 사용 예제

- input 함수 : 사용자로부터 입력을 받는 데 사용하는 내장함수, 문자열 형태로 반환
- name = input("이름을 입력하세요: ")
- print("안녕하세요, " + name + "님!")

- age = input("나이를 입력하세요: ")
- age = int(age)
- print("당신의 나이는", age, "살입니다. ")

[예시]

```
password = "password123"
attempt = 0
while atempt < 3:
    user_password = input("비밀번호를 입력하세요: ")
    attempt += 1
    if user_password == password:
        print("비밀번호가 일치합니다.")
        break
    else:
        print("비밀번호가 틀렸습니다.")
if attempt == 3:
    print("비밀번호 입력 횟수를 초과했습니다.")
```

2.1 기본함수 (1/8)

기본함수

- 함수란 특정 작업을 수행하는 코드의 블록으로 재사용 가능한 프로그램의 구성 단위
- 기본함수를 정의할 때는 def 를 사용하고, 함수 이름과 괄호 ( ) 가 필요하며, 괄호안에 매개변수를 정의하여 데이터를 전달받음

특징

- 코드의 재사용성 향상
- 프로그램의 구조화 및 모듈화 가능
- 유지보수의 용이성
- 코드의 가독성 향상

- 중복코드 감소
- 프로그램 개발시간 단축
- 디버깅 용이

형식

```
def 함수이름(변수1, 변수2):
    -- 함수 본문 --
    return 반환값
```

효율적인 함수설계

좋은 예시

```
def get_user_info(user_id):
“””사용자 정보를 조회하는 함수“””
    user = database.query(user_id)
    return {
        ‘name’ : user.name,
        ‘email’ : user.email,
        ‘status’ : user.status
}
```

좋지않은 예시

```
def process_user(user_id):
    “”” 여러가지 일을 한번에 하는 함수“””
    user = database.query(user_Id)
    send_email(user.email)
    update_status(user.status)
    generate_report(user)
```

2.1 기본함수 (2/8)

매개변수가 없는 함수

[예시]

```
def greet():
    print("안녕하세요!")

greet()     # 출력 : 안녕하세요
```

매개변수가 있는 함수

[예시]

```
def add(a, b):
    return a + b

result = add(3, 5)
print(result)     # 출력: 8
```

2.1 기본함수 (3/8)

기본값 매개변수 사용

[예시]

```
def greet(name="홍길동님"):
    print(f"안녕하세요, {name}!")

greet()     # 출력: 안녕하세요, 홍길동님!
greet("준호님")     #출력: 안녕하세요, 준호님!
```

가변 매개변수 사용

- *args : 임의의 개수의 숫자를 받아들임

[예시]

```
def multiply(*args):
    result = 1
    for num in args:
        result *= num
    return result

print(multipy(2, 3, 4))     # 출력: 24
```

2.1 기본함수 (4/8)

if문 활용 예시 (1)

[예시]

```
def check_number(num):
    """숫자가 양수, 음수, 또는 0인지 확인"""
    if num > 0:
        return "양수입니다."
    elif num < 0:
        return "음수입니다."
    else:
        return "0입니다."

print(check_number(5))     # 출력 : 양수입니다.
print(check_number(-3))    # 출력 : 음수입니다.
print(check_number(0))     # 출력 : 0입니다.
```

if문 활용 예시 (2)

[예시]

```
def login_check(username, password):
    """사용자 이름과 비밀번호를 확인"""
    if username == "admin" and password == "1234":
        return "로그인 성공"
    else:
        return "로그인 실패"

print(login_check("admin", "1234"))     # 출력 : 로그인 성공
print(login_check("admin", "5678"))     # 출력 : 로그인 실패
```

2.1 기본함수 (5/8)

for문 활용 예시 (1)

[예시]

```
# 리스트에서 최대값을 찾는 함수
def find_max(numbers):
    max_num = numbers[0]
    for num in numbers:
        if num > max_num:
            max_num = num
    return max_num

numbers = [1, 5, 3, 9, 2]
print(find_max(numbers))     # 출력 : 9
```

for문 활용 예시 (2)

[예시]

```
# 주어진 범위 내의 짝수만 반환하는 함수
def get_even_numbers(start, end):
    even_nums = []
    for num in range(start, end+1):
        if num % 2 == 0:
            even_nums.append(num)
    return even_nums

print(get_even_numbers(1, 10))     # 출력 : [2, 4, 6, 8, 10]
```

2.1 기본함수 (6/8)

while문 활용 예시 (1)

[예시]

```
# 목표 금액 모을 때까지 저축하는 시뮬레이션 함수
def saving_simulation(target_amount, monthly_saving):
    total = 0
    months = 0
    while total < target_amount:
        total += monthly_saving
        months += 1
        print(f"{months}개월 차: {total: ,}원 저축")     # {total: ,}에서 콤마는 1000단위 구분
    return f"{months}개월 만에 목표 금액 달성!"

print(saving_simulation(1000000, 200000))     # 출력 : 5개월 만에 목표 금액 달성
```

while문 활용 예시 (2)

[예시]

```
# 숫자 맞추기 게임 함수
def guess_number(answer):
    count = 0
    guess = None     # 아무 값도 할당하지 않은 상태
    while guess != answer:     # !=  '같지 않다' 의미의 비교 연산자
        guess = int(input("1부터 100 사이의 숫자를 맞춰보세요: "))
        count += 1
        if guess < answer:
            print("더 큰 숫자입니다!")
        elif guess > answer:
            print("더 작은 숫자입니다!")
    return f"정답입니다! {count}번 만에 맞추셨습니다."
print(guess_number(45))
```

2.1 기본함수 (7/8)

응용예시 (1)

[예시]

```
# 데이터 정규화 함수  (cf. 정규화: 서로 다른 범위의 값들을 비교할 때, 값들을 동일한 기준으로 맞춤
def normalize_data(numbers):
    """
    데이터를 0~1 사이의 값으로 정규화하는 함수
    정규화 = (값 – 최소값) / (최대값 – 최소값)
    매개변수:
        numbers: 정규화할 숫자 리스트
    반환값:
        정규화된 데이터 리스트 (0~1 사이의 값)
    """
    min_value = min(numbers)     # 데이터의 최소값
    max_value = max(numbers)     # 데이터의 최대값
    normalized = []     #정규화된 값을 저장할 리스트

    for num in numbers:
        # 정규화 공식 적용
        norm_value = (num – min_value) / (max_value – min_value)
        normalized.append(norm_value)
    return normalized

data = [10, 20, 30, 40, 50]
print("===== 데이터 정규화 예제 =====")
print("원본 데이터:", data)     # 출력 : 원본 데이터 : [10, 20, 30, 40, 50]
normalized = normalize_data(data)
print("정규화된 데이터:", normalized)     # 출력 : [0.0, 0.25, 0.5, 0.75, 1.0]
```

2.1 기본함수 (8/8)

응용예시 (2)

```
# 데이터 분할 함수
def split_data(data, train_ratio=0.8):
    """
    데이터를 학습용과 테스트용으로 분리하는 함수
    매개변수:
        data: 분할할 데이터 리스트
        train_ratio: 학습 데이터의 비율 (기본값 0.8 = 80%)
    반환값:
        train_data : 학습용 데이터
        test_data: 테스트용 데이터
    """
    # 학습 데이터 개수 계산
    split_index = int(len(data) * train_ratio)     # len함수 : 문자열, 리스트, 튜플 등 길이 측정
    # 데이터 분할
    train_data = data[:split_index]     # 처음부터 split_index까지
    test_data = data[split_index:]     # split_index부터 끝까지
    return train_data, test_data
# 데이터 분할 함수 사용 예시
full_data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("===== 데이터 분할 예제 =====")
print(" 전체 데이터:", full_data)     # 콤마(,)로 결과로 나오는 리스트의 요소 값을 구분
# 80:20으로 분할
train, test = split_data(full_data)
print("₩기본 비율(80%)로 분할:")
print("학습 데이터:", train}     # 학습 데이터 : [1, 2, 3, 4, 5, 6, 7, 8]
print("테스트 데이터:", test)     # 테스트 데이터 : [9, 10]
```

2.2 람다함수 (1/2)

람다함수

- 익명함수로 이름 없이 간단히 사용이 가능
- lambda 키워드를 사용해 작성하며, 한 줄로 작성되는 간단한 함수에 적합

특징

- 간결함
  - 한 줄로 함수의 동작을 표현 가능
  - 일반적으로 짧은 연산이나 표현식에 사용
- 이름이 없음
  - 람다 함수는 익명으로, 변수에 할당하거나 다른 함수의 인자로 전달될 때 주로 사용

형식

lambda 변수1, 변수2: 표현식

람다함수 vs 일반함수

람다함수

```
multiply = lambda x, y: x * y

print(multiply(3, 4))    # 출력: 12
```

일반함수

```
def multiply(x, y):
    return x * y

print(multiply(3, 4))    # 출력: 12
```

2.2 람다함수 (2/2)

단순 계산 및 여러 매개변수 사용

[예시 : 단순계산]

```
square = lambda x: x ** 2
print(square(5))     # 출력: 25
```

[예시: 여러 매개변수 사용]

```
add = lambda a, b: a + b
print(add(3, 7))     # 출력: 10
```

조건문 사용 및 다른 함수 인자로 사용

[예시 : 조건문 사용]

```
check_even = lambda x: "짝수" if x % 2 == 0 else "홀수"
print(check_even(4))     # 출력 : 짝수
```

[예시: 다른 함수의 인자로 사용]

```
numbers = [5, 2, 9, 1]
sorted_numbers = sorted(numbers, key=lambda x: x)
print(sorted_numbers)     # 출력: [1, 2, 5, 9]
```

- cf. sorted 함수 : 객체를 입력받아 정렬된 리스트를 반환하는 내장함수. 다른 메모리 사용
- cf. sort 함수 : 객체를 입력받아 정렬된 리스트를 반환하는 내장함수, 같은 메모리 사용

2.3 내장함수 (1/3)

내장함수

- 파이썬에서 기본적으로 제공하는 함수로, 임포트하지 않고 바로 사용 가능
- 코드를 간결하게 작성하고 다양한 작업을 쉽게 처리할 수 있도록 설계됨

특징

- 즉시 사용 가능
  - 별도의 설치나 임포트가 필요하지 않음
  - 예: print(), len() 등
- 다양한 기능 지원 : 데이터 변환, 수학 계산, 문자열 처리, 정렬, 입력/출력 등
- 효율적이고 최적화된 동작 : 대부분의 내장함수는 파이썬 C API로 구현되어 빠르고 안정적임

구분	설명	주요 내장함수
데이터 변환 및 확인 함수	타입 변환, True/False 반환 등에 사용	type(), int(), float(), str(), boo1()
데이터 처리 함수	길이, 합계, 최대, 최소, 정렬된 리스트반환에 사용	len(), sum(), max(), min(), sorted()
입력/출력 함수	화면에 출력, 사용자 입력, 문자열포맷팅에 사용	print(), input(), format()
수학함수	수학 연산에 사용	abs(), round(), pow()
함수관련 기능	각 요소에 함수적용, 필터링, 튜플로 반환 등에 사용	map(), filter(), zip(), enumerate()

2.3 내장함수 (2/3)

데이터 변환 및 확인 함수

함수	설명	예제
type()	객체의 자료형 반환	type(123) → int
int()	정수로 변환	int("10") → 10
float()	실수로 변환	float("3.14") → 3.14
str()	문자열로 변환	str(123) → '123'
bool()	참/거짓 여부 반환	bool(0) → False, bool(1) → True
isinstance()	객체가 특정클래스의 인스턴스인지 확인	isinstance(123, int) → True

데이터 처리 함수

함수	설명	예제
len()	객체의 길이(요소 수) 반환	len([1, 2, 3]) → 3
sum()	요소의 합계 반환	sum([1, 2, 3]) → 6
max()	최댓값 반환	max([1, 2, 3]) → 3
min()	최솟값 반환	min([1, 2, 3]) → 1
sorted()	정렬된 리스트 반환	sorted([3, 1, 2]) → [1, 2, 3]
reversed()	역순 반복자 반환	list(reversed([1, 2, 3])) → [3, 2, 1]

2.3 내장함수 (3/3)

입력/출력 함수

함수	설명	예제
print()	화면에 출력	print("Hello, world") → Hello, World!
input()	사용자 입력을 문자열로 반환	input("Enter name: ") → 사용자 입력 값
format()	문자열 포맷팅	"{: .2f}".format(3.14159) → 3.14 cf. {: .2f} : 소수점 아래 실수형 두자리

수학 함수

함수	설명	예제
abs()	절댓값 반환	abs(-5) → 5
round()	반올림	round(3.14159, 2) → 3.14
pow()	거듭제곱 계산	pow(2, 3) → 8

함수 관련 기능

함수	설명	예제
map()	각 요소에 함수를 적용	map(str, [1, 2, 3]) → ['1', '2', '3']
filter()	조건을 만족하는 요소만 필터링	filter(lambda x: x>2, [1, 2, 3]) → [3]
zip()	여러 이터러블을 묶어서 튜플로 반환	zip([1, 2], ['a', 'b']) → [(1, 'a'), (2, 'b')]
enumerate()	인덱스와 요소를 함께 반환	enumerate(['a', 'b']) → [(0, 'a'), (1, 'b')]

3. 메서드

3.1 메서드와 함수의 차이 비교

메서드

- 메서드(Method)는 특정 객체에 속하는 함수로, 해당 객체의 데이터를 조작하거나 동작을 수행하는데 사용됨
- 클래스 내부에 정의되며, **객체와 연관되어 호출**됨, 일반적으로 메서드는 첫번째 매개변수로 객체 자체인 self를 받음

특징

- 메서드는 특정 객체에 속해 있으며, 그 객체의 속성에 접근하거나 동작을 정의
- 모든 인스턴스 메서드는 첫번째 매개변수로 객체 자신을 전달받음
- 여러 객체에서 공통적으로 사용되는 동작을 정의하고 재사용할 수 있음

구분	메서드	내장함수
정의 위치	객체 또는 클래스에서 정의	클래스 내부나 독립적으로 정의
호출방식	객체.메서드명(인자)	함수명(인자)
첫번째 매개변수	self(인스턴스 메서드) 또는 cls(클래스 메서드)	추가 매개변수를 요구하지 않음
소속	객체나 클래스에 소속	특정 객체/클래스에 종속되지 않음
예시	str.upper(), list.append(), 사용자정의 메서드	len(), print(), 사용자 정의 함수

3.2 리스트(list) 메서드 (1/3)

리스트(List) 메서드

☑ 리스트 메서드는 리스트 객체에서 사용할 수 있는 함수들로, 리스트의 요소를 추가, 삭제, 수정하거나 정렬 등의 작업을 수행

사용 예시 (append)

- 리스트의 끝에 요소를 추가

- fruits = [‘apple’, ‘banana’]
- fruits.append(‘cherry’)
- print(fruits) # 출력 : [‘apple’, ‘banana’, ‘cherry’]

사용 예시 (extend)

- 리스트에 다른 리스트나 반복 가능한 객체의 모든 요소를 추가

- fruits = [‘apple’, ‘banana’]
- fruits.extend([‘cherry’, ‘strawberry’])
- print(fruits) # 출력 : [‘apple’, ‘banana’, ‘cherry’, ‘strawberry’]

사용 예시 (insert)

- 지정한 위치에 요소를 삽입

- fruits = [‘apple’, ‘cherry’]
- fruits.insert(1, ‘banana’)
- print(fruits) # 출력 : [‘apple’, ‘banana’, ‘cherry’]

3.2 리스트(list) 메서드 (2/3)

사용 예시 (remove)

- 첫 번째로 나타나는 특정 요소를 삭제

```
fruits = ['apple', 'banana', 'cherry']
fruits.remove('banana')
print(fruits)    # 출력 : ['apple', 'cherry']
```

사용 예시 (pop)

- 특정 인덱스의 요소를 삭제하고 반환, 인덱스를 생략하면 마지막 요소를 제거

```
fruits = ['apple', 'banana', 'cherry']
last_item = fruits.pop()
print(last_item)    # 출력: cherry
print(fruits)    # 출력: ['apple', 'banana']
```

사용 예시 (sort)

- 리스트를 오름차순으로 정렬(기본값). reverse=True를 설정하면 내림차순 정렬

```
numbers = [3, 1, 4, 1, 5]
numbers.sort()
print(numbers)    # 출력: [1, 1, 3, 4, 5]
```

3.2 리스트(list) 메서드 (3/3)

사용 예시 (reverse)

- 리스트의 순서를 반대로 변경

```
numbers = [1, 2, 3]
numbers.reverse()
print(numbers)    # 출력: [3, 2, 1]
```

사용 예시 (count)

- 리스트에서 특정 요소의 개수를 반환

```
numbers = [1, 2, 3, 1]
print(numbers.count(1))    # 출력: 2
```

사용 예시 (index)

- 특정 요소의 인덱스를 반환

```
fruits = ['apple', 'banana', 'cherry']
print(fruits.index('banana'))    # 출력: 1
```

3.3 문자열(String) 메서드 (1/3)

문자열(String) 메서드

☑ 문자열 메서드는 문자열 객체에서만 사용 가능한 함수로, 문자열을 직접 수정하지 않고 새 문자열을 반환(불변성 유지)

사용 예시 (lower)

- 문자열을 소문자로 변환

```
text = “Hello World”
print(text.lower())    # 출력: hello world
```

사용 예시 (upper)

- 문자열을 대문자로 변환

```
text = “Hello World”
print(text.upper())    # 출력: HELLO WORLD
```

사용 예시 (strip)

- 양쪽 끝의 공백 또는 특정문자 제거

```
text = “  Hello World  ”
 print(text.strip())    # 출력: Hello World
```

3.3 문자열(String) 메서드 (2/3)

사용 예시 (replace)

- 특정 문자열을 다른 문자열로 치환

```
text = “banana”
print(text.replace(“a”, “o”))    # 출력: bonono
```

사용 예시 (split)

- 문자열을 구분자를 기준으로 분리하여 리스트로 반환

```
text = “apple, banana, cherry”
print(text.split(‘,’))    # 출력: [‘apple’, ‘banana’, ‘cherry’]
```

사용 예시 (join)

- 리스트나 튜플의 요소를 문자열로 결합

```
fruits = [‘apple’, ‘banana’, ‘cherry’]
print(‘, ‘.join(fruits))    # 출력: apple, banana, cherry
```

3.3 문자열(String) 메서드 (3/3)

사용 예시 (find)

- 부분 문자열의 첫 번째 위치를 반환

- text = “Hello World”
- print(text.find(“World”)) # 출력: 6

사용 예시 (startswith/ endswith)

- 문자열이 특정 접두사/접미사로 시작하거나 끝나는지 확인
- text = “Hello World”
- print(text.startswith(“Hello”)) # 출력: True
- print(text.endswith(“World”)) # 출력: True

사용 예시 (isnumeric)

- 문자열이 숫자로만 구성되어 있는 지 확인

- text = “12345”
- print(text.isnumeric()) # 출력: True

사용 예시 (capitalize)

- 첫 번째 문자를 대문자로, 나머지는 소문자로 변환

- text = “hello world”
- print(text.capitalize()) # 출력: Hello world

4. 예외처리

4.1 예외처리 정의 및 필요성

예외처리

- 예외(Exception)는 프로그램 실행 중 발생할 수 있는 오류를 뜻하며, 실행을 중단시키는 원인이 됨
- 예외처리는 프로그램이 예외 상황에서도 정상적으로 동작하도록 대비하는 프로그래밍 기법

예외처리 필요성

- 프로그램 안정성 : 예외 상황이 발생하더라도 프로그램이 중단되지 않고 동작을 이어감
- 에러 원인 파악 : 적절한 에러 메시지 출력 및 디버깅에 도움
- 사용자 경험 개선 : 잘못된 입력이나 시스템 오류에도 사용자가 혼란을 겪지 않도록 처리

구분	설명	실행 여부
try	예외 발생 가능성이 있는 코드를 작성	예외 발생 시 종료
except	예외가 발생했을 때 실행할 코드를 작성	해당 예외가 발생한 경우
else	예외가 발생하지 않았을 때 실행할 코드를 작성	예외 없이 try 블록이 성공적으로 실행된 경우
finally	예외 발생 여부와 상관없이 항상 실행할 코드를 작성	항상 실행

4.2 파이썬 예외처리 구문 및 예시 (1/3)

기본 구조

```
try:
    # 예외가 발생할 가능성이 있는 코드
except 예외 타입 as 변수:
    # 예외가 발생했을 때 실행할 코드
else:
    # 예외가 발생하지 않았을 때 실행할 코드
finally:
    # 예외 발생 여부와 상관없이 항상 실행할 코드
```

try 블록

- 예외가 발생할 가능성이 있는 코드를 작성

```
try:
    num = int(input("숫자를 입력하세요: "))
    print(f'입력한 숫자는 {num}입니다.")
```

4.2 파이썬 예외처리 구문 및 예시 (2/3)

except 블록

- 특정 예외가 발생했을 때 실행되는 코드
- 예외 타입을 명시하거나, 생략하여 모든 예외를 처리할 수 있음

```
try:
    num = int(input("숫자를 입력하세요: "))
except ValueError:
    print("유효한 숫자를 입력해야 합니다.")
```

else 블록

- 예외가 발생하지 않은 경우 실행되는 코드

```
try:
    num = int(input("숫자를 입력하세요: "))
except ValueError:
    print("유효한 숫자를 입력해야 합니다.")
else:
    print(f"정상적으로 입력되었습니다: {num}")
```

4.3 파이썬 예외처리 구문 및 예시 (3/3)

finally 블록

- 예외 발생여부와 관계없이 항상 실행되는 코드로 주로 리소스해제(파일닫기, 네트워크 종료)에 사용

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
except FIleNotFoundError:
    print("파일을 찾을 수 없습니다.")
finally:
    file.close()
    print("파일을 닫았습니다.")
```

예외 객체 활용

- 예외를 변수로 받아서 상세 정보를 확인할 수 있음

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"예외 발생: {e}")    # 출력: 예외발생: division by zero
```

4.4 예외처리 유의사항 (1/2)

구체적인 예외타입 처리

- 일반적인 except 블록보다는 예외 타입을 명확히 지정

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("0으로 나눌 수 없습니다.")
```

finally 블록 활용

- 파일, 네트워크 등 리소스 해제는 반드시 finally에서 처리

```
try:
    file = open("example.txt", "r")
finally:
    file.close()
```

4.4 예외처리 유의사항 (2/2)

else와의 조합

- 예외가 발생하지 않은 경우와 발생한 경우를 명확히 구분하여 코드 작성

```
try:
    num = int(input("숫자를 입력하세요: "))
except ValueError:
    print("숫자가 아닙니다.")
else:
    print(f"정상 입력: {num}")
```

불필요한 예외남용 금지

- 모든 코드를 try 블록에 넣는 것은 좋지 않으므로, 예외 발생 가능성이 있는 코드만 처리

잘못된 예시

```
try:
    # 예외가 발생할 가능성 있는 코드까지 포함
    name = input("이름 입력: ")
    age = int(input("나이 입력: "))
    print(f"{name}님. 당신의 나이는 {age}살")
except ValueError:
    print("올바른 숫자를 입력하세요.")
```

name = input(..)은 예외를 발생시키지 않으므로 try 블록에 포함할 필요가 없음

개선된 예시

```
name = input("이름 입력: ")
try:
    age = int(input("나이 입력: "))
    print(f"{name}님. 당신의 나이는 {age}살")
except ValueError:
    print("올바른 숫자를 입력하세요.")
```

try 블록에는 **숫자 변환**처럼 예외 발생 가능성이 높은 코드만 포함

5. 자원관리 자동화

5.1 with문 특징

- 컨텍스트 매니저를 사용하는 코드 블록을 간편하게 관리하기 위해 제공되는 Python 문법
- 주로 파일 입출력, 리소스 관리, 데이터베이스 연결 등에 사용됨

특징

특징	설명
자동자원 관리	• with문 블록이 끝나면 파일이나 리소스가 자동으로 닫힘(__exit__ 메서드 호출) • 개발자가 직접 close()를 호출할 필요가 없음
예외 처리 간소화	• with문 내부에서 예외가 발생해도 리소스 정리(close(), release())가 보장됨
코드 간결화	• 명시적으로 try-finally를 사용하지 않아도 되므로 코드가 더 깔끔해짐

컨텐스트 매니저란?

- 컨텍스트 매니저(Context Manager)는 특정작업의 시작과 종료를 관리하는 도우미
- with 문에서 자동으로 동작
- Python에서 with문과 함께 사용되며, 리소스 관리를 더 쉽고 안전하게 처리하도록 지원
- 예를들어 파일을 열면 작업이 끝난 후 자동으로 파일을 닫아주고, 데이터베이스 연결을 열었다면 작업 후 자동으로 연결을 종료

5.2 with문 예시

방식비교

- with문 방식이 더 안전하고 가독성이 좋아 실수 가능성을 줄일 수 있음
- 파일이나 리소소를 다룰 때는 가능하면 with문을 사용하는 것을 권장

기존 방식

```
file = open("example.txt", "r")
try:
    print(file.read())
finally:
    file.close()
```

with문 방식

```
with open("example.txt", "r") as file
    print(file.read())
```

예시

- [파일 열고 읽기/쓰기]
- with open('dataset/human.txt', 'w') as f:
- f.write('안녕하세요 반갑습니다. 홍길동입니다.\n')
- f.write('파이썬 with문 예시입니다.\n')

6. 데이터 분석 및 시각화 함수

6.1 Numpy (1/2)

- numpy는 파이썬에서 과학 계산을 위한 필수 라이브러리
- 대규모 다차원 배열과 행렬 연산을 효율적으로 처리하는 기능을 제공하여 데이터 분석에서 널리 사용

특징

특징	설명
빠르고 효율적인 연산	Numpy 배열은 C로 구현되어 있어 파이썬 list 보다 훨씬 빠르고 효율적이 연산을 제공.
벡터화 연산	여러 개의 값에 대해 한 번에 연산을 수행하는 방식
다양한 수학 함수	삼각 함수, 지수 함수, 통계 함수 등 다양한 수학 함수를 제공
배열 생성 및 조작	다양한 방법으로 배열을 생성하고, shape을 변경하거나 원하는 요소를 추출
선형대수 연산	행렬 곱, 역행렬, 고유값 분해 등 선형대수 연산을 지원

6.1 Numpy (2/2)

Numpy 함수 예시

함수	설명	예시
np.array()	리스트를 Numpy 배열로 변환	import numpy as np arr = np.array([1, 2, 3]) arr

np.arange()	주어진 범위 내에서 일정한 간격의 값을 가지는 배열 생성	# arrange(시작값, 종료값, 건너뛰기)
		arr = np.arange(0, 10, 2) # 2씩 증가

np.random.rand()	0과 1사이의 균일 분포를 따르는 난수 생성	# np.ramdom : 난수생성, rand: 0~1난수
		arr = np.random.rand(3, 3)

np.reshape	배열의 shape 변경	arr = arr.reshape(1, 9)
np.sum()	배열 요소의 합	np.sum([1, 2, 3]) # 출력값 : 6
np.mean()	배열 요소의 평균	np.mean([1, 2, 3]) # 출력값 : 2
np.std()	배열 요소의 표준편차 계산	np.std([1, 2, 3]) # 출력값 : 0.816

※ 알아두어야 할 용어 (분산, 표준편차)

분산과 표준편차 예시

구분	1번	2번	3번	4번	5번
변량	175	177	179	181	183
평균	(175 + 177 + 179 + 181 + 183) / 5 = 179				
편차 (변량 – 평균)	175 – 179 = −4	177 – 179 = −2	179 – 179 = 0	181 – 179 = 2	183 – 179 = 4
편차제곱	$(-4)^2 = 16$	$(-2)^2 = 4$	0	$2^2 = 4$	$4^2 = 16$
분산	(16 + 4 + 0 + 4 + 16) / 5 = 40 /5 = 8				
표준편차	$\sqrt{8}$ = 2.828				

6.2 Pandas (1/2)

- Numpy를 기반으로 하여, **표 형태의 데이터**를 효율적으로 처리하고 분석할 수 있는 다양한 함수와 자료 구조를 제공
- Series (1차원 배열과 유사하며, 각 요소에 인덱스 부여), DataFrame (표 형태의 2차원 데이터를 나타내며, 행과 열로 구성)

특징

특징	설명
데이터 읽기/쓰기	다양한 형식의 데이터(CSV, Excel, SQL 등)를 읽고 쓸 수 있음
데이터 선택 및 필터링	원하는 데이터를 선택하고 조건에 맞는 데이터를 추출할 수 있음
데이터 변형	데이터를 정렬, 그룹화, 평균 등 다양한 방식으로 변형할 수 있음
결측치 처리	결측치를 찾고 처리하는 다양한 방법을 제공
데이터 시각화	Matplotlib와 연동하여 데이터를 시각화할 수 있음

6.2 Pandas (2/2)

Pandas 함수 예시

함수	설명	예시
read_csv()	CSV 파일 읽기	df = pd.read_csv('data.csv')
head(), tail()	데이터프레임의 처음 또는 마지막 부분 확인	df.head(), df.tail()
shape	데이터프레임의 행과 열 개수 확인	df.shape
info()	데이터프레임의 요약 정보 확인	df.info()
describe()	데이터프레임의 기본적인 통계 정보 확인	df.describe()
loc[], iloc[]	데이터 선택	df.loc['index_name'], df.iloc[0]
groupby	데이터 그룹화	df.groupby('column_name').mean()
sort_values()	데이터 정렬	df.sort_values('column_name')
fillna()	결측치 채우기	df.fillna(0)
dropna()	결측치 행 삭제	df.dropna()

6.3 Matplotlib (1/2)

- 다양한 종류의 그래프를 생성하여 데이터를 시각적으로 표현
- 데이터 분석 과정에서 얻은 결과를 효과적으로 전달하는 데 사용

특징

특징	설명
유연성	다양한 종류의 그래프를 생성할 수 있으며, 그래프의 스타일, 크기, 색상 등을 자유롭게 조절할 수 있음
확장성	다양한 서드파티 라이브러리와 연동하여 더욱 복잡하고 정교한 시각화를 구현할 수 있음
객체 지향 인터페이스	그래프의 각 요소(축, 선, 점 등)를 객체로 다루어 세밀하게 조작할 수 있음

6.3 Matplotlib (2/2)

Matplotlib 함수 예시

함수	설명	예시
plt.plot()	선 그래프 생성	plt.plot(x, y)
plt.scatter()	산점도 생성	plt.scatter(x, y)
plt.bar()	막대 그래프 생성	plt.bar(x, height)
plt.hist()	히스토그램 생성	plt.hist(data, bins=20)
plt.boxplot()	상자 그림 생성	plt.boxplot(data)
plt.pie()	원 그래프 생성	plt.pie(x)
plt.subplot()	여러 개의 그래프를 하나의 Figure에 배치	plt.subplot(2, 2, 1)
plt.xlabel()	x축 레이블 설정	plt.xlabel('X축 이름')
plt.ylabel()	y축 레이블 설정	plt.ylabel('Y'축 이름')
plt.title()	그래프 제목 설정	plt.title('그래프 제목')
plt.legend()	범례 설정	plt.legend()
plt.show()	그래프 출력	plt.show()

6.4 Seaborn (1/2)

- Matplotlib을 기반으로 개발된 파이썬 시각화 라이브러리
- 통계적 시각화를 위한 고급 인터페이스를 제공. Matplotlib보다 더욱 세련되고 통계적인 시각화를 쉽게 만들 수 있다는 장점

특징

특징	설명
고급 인터페이스	Matplotlib의 복잡한 설정 없이 간결한 코드로 다양한 시각화를 생성할 수 있음
통계적인 시각화	데이터 분포, 상관관계, 회귀 분석 등 통계적인 개념을 시각화하는데 특화됨
매력적인 스타일	기본적으로 제공되는 다양한 스타일 테마를 통해 시각적으로 매력적인 그래프를 생성할 수 있음
Seaborn과 Pandas의 원활한 통합	Pandas DataFrame을 직접 입력하여 시각화를 수행할 수 있음

6.4 Seaborn (2/2)

Seaborn 함수 예시

함수	설명	예시
sns.distplot()	히스토그램과 커널 밀도 추정을 함께 표현	sns.distplot(data.kde=True)
sns.boxplot()	상자 그림 생성	sns.boxplot(x='category', y='value', data=df)
sns.violinplot()	바이올린 플롯 생성	sns.violinplot(x='category', y='value', data=df)
sns.scatterplot()	산점도 생성	sns.scatterplot(x='x', y='y', data=df)
sns.pairplot()	여러 변수 간의 관계를 한 번에 시각화	sns.pairplot(df)
sns.heatmap()	히트맵 생성	sns.heatmap(corr_matrix)
sns. lmplot()	선형 회귀 모델 시각화	sns.lmplot(x='x', y='y', data=df)

7. 클래스(class)

7.1 클래스 정의 및 특징 (1/3)

- 객체를 만들기 위한 설계도
- 관련된 데이터(속성)와 함수(메서드)를 하나로 묶어서 관리할 수 있음

특징

특징	설명
속성	• 클래스 내부의 변수로, 객체가 가지는 데이터를 저장함 • self 키워드를 사용해 정의하고, 객체별로 독립적으로 관리됨 [예시] • self.name = name
메서드	• 클래스 내부의 함수로, 객체가 할 수 있는 행동(동작)을 정의 • 메서드는 항상 첫 번째 인자로 self를 받아 객체 자신을 참조함 [예시] • def greet(self): • print(“Hello!”)

7.1 클래스 정의 및 특징 (2/3)

특징

특징	설명
생성자	• _init_ 메서드는 객체 생성 시 자동으로 호출되어 초기화 작업을 함 • 객체 생성 시 전달받은 값을 속성에 할당하는 데 주로 사용됨 [예시] • def __init__(self, name, age): • self.name = name • self.age = age

캡슐화	• 속성이나 메서드를 외부에서 직접 접근하지 못하도록 보호
	• 이름 앞에 밑줄 두개(__)를 붙이면 비공개(private) 속성/메서드가 됨
	[예시]
	class Person:
	  def __init__(self, name, age):
	  self.__name = name # private 속성
	  self.age = age
	
	  def get_name(self):
	  return self.__name # private 속성에 접근하는 메서드

7.1 클래스 정의 및 특징 (3/3)

특징

특징	설명

상속	• 기존 클래스를 상속받아 새로운 클래스를 생성할 수 있음
	• 상속을 통해 기존 클래스의 기능을 재사용하고 확장
	• __init__ : 파이썬의 메서드중의 하나, 객체의 초기 상태를 설정하는 코드 포함
	[예시]
	class Student(Person): #Person 클래스를 상속
	  def __init__(self, name, age, student_Id):
	  super().__init__(name, age) # super(): 부모 클래스 참조 객체 반환
	  self.student_id = student_Id

클래스의 장점

클래스의 장점

- 코드 재사용 : 한 번 정의한 클래스를 여러 곳에서 재사용 가능
- 구조화된 코드 : 속성과 동작을 묶어 관리하므로 코드 가독성 증가
- 유지보수 용이 : 캡슐화와 상속을 통해 확장성과 유지보수성 강화

7.2 클래스 예시 (1/8)

클래스 예시

```
# 간단한 클래스 예제
class Dog:
    #초기화 메서드 (생성자)
    def __init__(self, name):
        self.name = name    # 인스턴스 변수

    def bark(self):
        print(f"{self.name}가 짖습니다!")

# 객체 생성
my_dog = Dog("멍멍이")
my_dog.bark()    # 출력 : 멍멍이가 짖습니다!
```

7.2 클래스 예시 (2/8)

클래스 예시 (상속)

```
# 클래스 예제(상속)
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

class Cat(Animal):    # Animal 클래스 상속
    def speak(self):
        return f"{self.name}가 야옹하고 웁니다."

# 객체 생성 및 사용
cat1 = Cat("나비")     # Cat 클래스의 인스턴스(객체) 생성
print(cat1.name)    # 출력 : 나비
print(cat1.speak())    # 출력 : 나비가 야옹하고 웁니다.

cat2 = Cat("미야")    # 다른 인스턴스 생성
print(cat2.name)    # 출력 : 미야
print(cat2.speak())    # 출력 : 미야가 야옹하고 웁니다.
```

7.2 클래스 예시 (3/8)

클래스 예시 (캡슐화)

```
# 클래스 예제(캡슐화)

class BankAccount:
    def __init__(self):
        self.__balance = 0    # 비공개 변수

    def deposit(self, amount):
        if amount 〉 0:
            self.__balance + = amount

    def get_balance(self):
        return self.__balance

# 실행 예제
account = BankAccount()
account.deposit(1000)
print(account.get_balance())    # 출력 : 1000
# print(account._balance)    # 오류 발생 : 직접 접근 불가
```

7.2 클래스 예시 (4/8)

클래스 예시 (클래스 변수와 인스턴스 변수 예제)

```
# 클래스 변수와 인스턴스 변수 예제
class Student:
    school_name = "파이썬 고등학교"    # 클래스 변수
    student_count = 0    # 클래스 변수

    def __init__(self, name):
        self.name = name # 인스턴스 변수
        Student.student_count += 1

# 실행 예제
student1 = Student("김철수")
student2 = Student("이영희")

print(Student.school_name)    # 출력: 파이썬 고등학교
print(student1.school_name)    # 출력: 파이썬 고등학교
print(student2.school_name)    # 출력: 파이썬 고등학교
print(Student.student_count)    # 출력: 2
```

7.2 클래스 예시 (5/8)

클래스 예시 (계산기)

```
# 계산기 클래스 예제
class Calculator:
    # 초기화 메서드: 아무 상태 없이 계산만 수행
    def __init__(self):
        print("계산기를 초기화합니다.")

    # 두 숫자를 더하는 메서드
    def add(self, a, b):
        return a + b

    # 두 숫자를 곱하는 메서드
    def multiply(self, a, b):
        return a * b

# 객체 생성 및 메서드 호출
calc = Calculator()    # 출력 : 계산기를 초기화합니다.
print("덧셈 결과:", calc.add(10, 20))    # 출력 : 덧셈결과 : 30
print("곱셈 결과:", calc.multiply(10, 20))    # 출력: 곱셈 결과 : 200
```

7.2 클래스 예시 (6/8)

클래스 예시 (학생정보 관리)

```
# 학생정보 관리 클래스 예제
class Student:
    # 생성자 : 학생 이름과 나이를 초기화
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # 학생 정보 출력 메서드
    def display_info(self):
        print(f"이름: {self.name}, 나이: {self.age}")

# 객체 생성 및 메서드 호출
student1 = Student("홍길동", 16)
student2 = Student("김철수", 18)

student1.display_info()     # 출력: 이름: 홍길동, 나이: 16
student2.display_info()     # 출력: 이름: 김철수, 나이: 18
```

7.2 클래스 예시 (7/8)

클래스 예시
(영화티켓할인)

```
# 영화 티켓 할인 관리 클래스 예제
class MoiveTicket:
    # 생성자 : 영화 제목과 기본 티켓 가격 초기화
    def __init__(self, title, price):
        self.title = title
        slef.price = price

    # 할인 적용 메서드
    def apply_discount(self, discount):
        self.price -= discount
        print(f"{self.title} 영화의 할인된 가격: {self.price}원")

# 객체 생성 및 메서드 호출
ticket = MovieTicket("어벤져스", 12000)
ticket.apply_discount(2000)    # 출력 : 어벤져스 영화 의 할인된 가격 : 10000원
```

7.2 클래스 예시 (8/8)

클래스 예시 (은행계좌)

```
# 간단한 은행 계좌 클래스

class BankAccout:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance
    # 입금
    def deposit(self, amount):
        self.balance += amount
        print(f"{amount}원이 입금되었습니다. 현재 잔액: {self.balance}원")
    # 출금
    def withdraw(self, amount):
        if amount 〉 self.balance:
            print("잔액이 부족합니다.")
        else:
            self.balance -= amount
            print(f"{amount}원이 출금되었습니다. 현재 잔액: {self.balance}원")

# 사용 예시
account = BankAccount("홍길동",  10000)
account.deposit(5000)    # 출력 : 5000원이 입금되었습니다. 현재 잔액: 15000원
account.withdraw(2000)   # 출력 : 20000원이 출금되었습니다. 현재 잔액: 13000원
```

7.3 Flask 구조 및 예제

Flask 애플리케이션 기본 파일 구조

- app.py 또는 main.py : 메인 애플리케이션 파일로 사용
- templates 폴더 : HTML 템플릿 파일들을 저장하는 곳으로 Flask가 기본적으로 이 폴더를 찾음
- static 폴더 : css, Javascript, 이미지 등의 정적 파일들을 저장

```
my_flask_app/
├── app.py
├── templates/
│   ├── index.html
│   └── ...
├── static/
│   ├── css/
│   ├── js/
│   └── images/
└── ...
```

7.3 Flask 구조 및 예제

체중과 신장을 입력하면 BMI를 계산하고, 결과를 DB에 저장한 후 결과 페이지를 보여주는 웹 애플리케이션

```
project/
├── app.py
├── bmi.py
├── db.py
├── schema.sql
├── templates/
│   ├── index.html
│   ├── result.html
│   └── history.html
└── static/
    └── style.css
```

기능 설명

1.app.py: Flask 웹 애플리케이션의 주요 파일로, 라우트와 요청 처리를 담당합니다.

2.bmi.py: BMI 계산과 관련된 로직을 포함하는 클래스를 정의합니다.

3.db.py: 데이터베이스 연결 및 종료 클래스를 정의합니다.

4.templates/index.html: 사용자가 체중과 신장을 입력할 수 있는 폼을 제공합니다.

5.templates/result.html: 계산된 BMI 결과를 표시합니다.

6.templates/history.html : 입력된 값을 저장하여 표시합니다.

7.static/style.css: 웹 페이지의 스타일을 정의합니다.

실행방법

- 위의 파일 구조와 같이 프로젝트 폴더 및 각 파일 생성
- 패키지 설치: pip install flask pymysql
- 터미널에서 프로젝트 폴더로 이동하여 다음 명령어 실행 : python app.py
- 웹 브라우저에서 http://localhost:5000 으로 접속