

2교시 - 파이썬 프로그래밍 빠른 요약

1. 파이썬 프로그램의 구조

1.1 들여쓰기로 프로그램의 구조가 결정됩니다.

- 프로그램의 최상위 블록은 들여쓰기를 하지 않습니다.
- 프로그램 코드가 콜론 `:`으로 끝나면 그 다음 줄부터 하위 블록
- 하위 블록은 상위 블록보다 조금 더 들여 써야 하며
- 같은 수준의 블록은 같은 들여쓰기를 맞춰야 합니다.
- 하위 블록이 종료되면 원래의 들여쓰기로 돌아갑니다.
- 하위 블록 아래 또 다른 하위 블록이 위치할 수 있습니다.
- 하위 블록은 다음과 같은 경우의 다음 행 부터 시작됩니다.
 - 함수를 선언한 후 - `def` 로 시작하는 코드 줄
 - 클래스를 선언한 후 - `class` 로 시작하는 코드 줄
 - 조건 제어에서 조건에 따라 실행되는 블록 - `if`, `elif`, `else` 로 시작하는 코드 줄
 - 반복문에서 루프 내에서 실행할 코드 블록 - `for`, `while` 로 시작하는 코드 줄
 - 예외 처리의 대상이 되는 코드 블록 - `try` 로 시작하는 코드 줄
 - 예외 발생 여부에 따라 실행할 코드 블록 - `except`, `finally` 로 시작하는 코드 줄
 - 컨텍스트 매니저에서 컨텍스트가 유효한 범위의 코드 블록 - `with` 로 시작하는 코드 줄
 - 기타 강의에 다루지 않은 특수한 경우 - `async`, `await` 키워드 또는 데코레이터 등의 특수 구문 등에 서도 만들어질 수 있습니다.

1.2 주석 또는 코멘트

- 프로그램 코드에서 문자 `#`이 등장하면, `#` 부터 그 줄의 마지막까지는 프로그램에 아무런 영향을 미치지 않습니다.
- 이를 활용해 유용한 기록을 프로그램에 남겨둘 수 있습니다.
 - 함수 또는 코드의 설명
 - 함수 또는 코드를 자신 또는 다른 개발자가 읽을 때 필요한 부가 정보
 - 파이썬 파일의 정보
 - 일부 특수한 용도
 - 기타 필요하다면 무엇이든...
- 단 문자열 내의 `#`은 주석을 생성하지 않습니다.

1.3 파이썬 프로그램 코드를 둘로 구분하면? - 표현식과 표현식이 아닌 것

표현식 (expression)

- 값을 반환하는 코드를 말합니다.
- 변수, 상수, 함수 호출, 산술 연산, 비교 연산, 논리 연산 등 값을 반환하는 부분은 모두 표현식에 해당됩니다.
- 반환값이 없는 함수를 호출할 때도 `None` 을 반환합니다.

문(또는 구문, statement)

- 값을 반환하지 않고 프로그램의 흐름을 제어하거나 상태를 변경하는 구문을 말합니다.
- 파이썬 프로그램에서 구문에 해당되는 요소는 다음과 같습니다.
 - 할당 연산, 복합 할당 연산
 - `if`, `elif`, `else`, `match`, `case` - 조건 제어
 - `for`, `while`, `continue`, `break` - 반복 제어
 - `try`, `except`, `finally`, `raise` - 예외 처리
 - `with`, `as` - 컨텍스트 관리
 - `import`
 - `def` (함수 또는 메서드 정의), `class` (클래스 정의)
 - `return` - 함수의 결과를 반환 또는 종료
 - `del` (변수 또는 객체를 삭제하는 키워드)
- 파이썬에서 연산에 사용되는 일부 키워드 (`True`, `False`, `lambda`, `and`, `or`, `not`, `None`, `is`, `in` 등)을 제외한 나머지 키워드는 모두 문을 구성하는 요소입니다.

1.4 파이썬 프로그램의 실행 순서

- 위에서 아래로 순차적으로 실행이 기본 원칙
 - 인터프리터 언어인 파이썬은 코드를 코드행 단위로 읽고 실행합니다.
- 파이썬 프로그램이 실행될 때의 일반적인 순서
 1. 파이썬 프로그램 파일(.py 파일)을 읽어 드리고 구문을 분석합니다.
 - 문법 오류(SyntaxError)가 있으면 프로그램은 실행되지 않고 오류가 발생합니다.
 2. 한 줄씩 처리
 - 함수 정의, 클래스 정의는 '기억'만 해 둡니다.
 - 나머지는 실제로 처리합니다.
 - 첫 줄부터 차례대로 처리하는 것이 원칙입니다.
 - 단 조건 제어, 반복 제어, 예외 처리 등 제어 컨트롤 영역에서는 컨트롤 방식에 따라 처리 순서가 조절됩니다.
 - 처리 순서가 조절되면 그 위치부터 한 줄씩 차례대로 처리합니다.
 3. 함수나 메서드가 호출되면

- 함수나 메서드를 호출하면 그 때 '기억'해 둔 위치로 찾아가 한 줄씩 차례대로 처리합니다.
- 함수나 메서드 내부에서 다른 함수나 메서드를 호출하면 다시 그 위치로 찾아가 한 줄씩 차례대로 처리합니다.
- 함수나 메서드가 종료되면, 직접 호출한 곳으로 돌아간 후, 그 다음의 명령을 실행합니다.

4. 객체를 생성하면

- 클래스의 객체를 생성하면 그 때 '기억'해 둔 위치를 참조해 객체를 생성합니다.

5. 다음의 경우 프로그램이 종료됩니다.

- 처리되지 않은 오류가 발생할 때
- `sys.exit()` 함수를 호출할 때
- 더 이상 실행할 코드 행이 없을 때

2. 파이썬의 변수, 연산, 자료형

2.1 변수와 할당 연산

리터럴과 변수

- 리터럴 - 한 번 사용하면 사라지는 각 자료형의 고정된 값
 - 참거짓값 리터럴 : `True`, `False`
 - 숫자 리터럴 : `10`, `10.0`, `(5+10j)`
 - 문자열 리터럴 : `"Hello, Python!"`
 - 리스트 리터럴 : `[1, 2, 3, 4, 5]`
- 변수
 - 리터럴을 메모리에 저장해 계속 사용할 수 있도록 만든 '이름을 가진 그릇'
 - 변수의 값은 컴퓨터 메모리 어딘가에 저장해 두고 변수의 이름을 통해서 변수의 값을 불러와 사용합니다.

변수 사용법

- 할당 연산으로 변수를 만들고 값을 저장합니다. (변수의 정의)
 - `변수의_이름 = 저장할_값`
 - 저장할 값은 리터럴, 변수, 리터럴과 변수를 사용하는 표현식 모두 사용할 수 있습니다.
 - 할당 연산은 연산자 `=` 를 기준으로 오른쪽을 모두 계산한 후 왼쪽 이름의 변수에 값을 저장합니다.
- 정의된 변수는 변수의 이름을 사용하면 값으로 바뀌어 사용됩니다.
 - 할당되지 않은, 즉 정의되지 않은 변수는 사용할 수 없습니다.
- 할당 연산으로 이미 정의된 이름의 변수의 값을 바꿀 수 있습니다.
 - 실제로는 같은 이름을 가진 새로운 변수가 만들어집니다.

- `temperature = temperature + 1` 할당 연산에서 왼쪽 `temperature` 와 오른쪽 `temperature` 는 이름은 같지만 알고 보면 실체는 다른 변수입니다.
- 복합 할당 연산을 사용해 변수의 값을 간편하게 바꿀 수 있습니다.
 - n만큼 증가 : `+= n`
 - n만큼 감소 : `-= n`
 - n배로 변경 : `*= n`
 - n으로 나눈 값으로 변경 : `/= n` (나눗셈 연산), `//= n` (내림나눗셈 연산)
- 변수의 재할당
 - 불변 자료형은 값을 바꿀 수 없는 자료형입니다. (숫자 자료형, 문자열 자료형, 튜플 등)
 - 가변 자료형은 값을 바꿀 수 있는 자료형입니다. (리스트, 딕셔너리, 세트, 사용자가 정의한 객체)
 - 불변 자료형 변수를 다른 변수에 재할당하면 새로운 그릇(변수)이 만들어지고 새로운 그릇(변수)에 값을 복사해 넣습니다.
 - 가변 자료형 변수를 다른 변수에 재할당하면 원래의 그릇(변수)에 재할당 하는 변수의 이름이 하나 더 만들어집니다. (참조)

변수와 함수의 이름을 만드는 법

- 기본 규칙
 - 한 글자 이상의 문자열로 구성됩니다.
 - 특수 문자 중에는 `_` 만 사용 가능하며, 눈에 보이지 않는 문자를 사용할 수 없습니다. (공백 문자, 이스케이프 시퀀스 등)
 - 첫 글자로 숫자를 사용할 수 없습니다.
 - 파이썬의 키워드는 변수의 이름으로 사용할 수 없습니다.
- PEP 8 - Style Guide for Python Code
 - 바람직한 파이썬 프로그래밍 스타일 가이드
 - 변수 이름 짓는 방법 등 파이썬 프로그램의 형태에 대한 일종의 모범적 표준을 제시합니다.
- 바람직한 변수 이름의 구성 (PEP 8 + 강사의 추천)
 - 변수 이름은 소문자만 사용합니다.
 - 변수 이름은 너무 길지만 읽기 쉽도록 대상을 잘 설명할 수 있도록 작성합니다.
 - 두 단어 이상의 연결은 `_` 을 사용하고 가급적 명사형 단어를 사용합니다. (snake case 작성법)
 - `o` 와 `l` 은 오해의 여지가 있다면 사용하지 않습니다.
 - 한 두자 정도의 짧은 이름의 변수는 특정한 용도로 사용합니다. (루프 변수 등)
 - `__` 또는 `_` 로 시작하는 변수 이름은 특정한 용도가 정해져 있는 경우가 대부분이므로 이에 맞게 사용합니다.
 - 영문 키보드에 없는 글자는 변수에 사용하지 않습니다.
- 모든 규칙에 앞서는 절대 규칙

- 함께 일하는 분들과 합의된 사항이 다른 모든 원칙에 우선합니다.

2.2 불변 기본 자료형

참거짓형 데이터 (Boolean 자료형)

- 논리적으로 참이면 `True`, 거짓이면 `False` 두 종류의 값이 속합니다.
- `if`, `while` 등의 조건 기준의 제어와 `and`, `or`, `not`의 논리 연산자에서 사용합니다.
 - 이 때 `False`, `0`, `0.0`, `''`, `[]`, `()`, `{}`, `set()`, `range(0)`, `None`을 사용하면 조건의 기준이나 논리 연산은 이를 `False`로 평가합니다.
 - `True`를 포함한 이외의 값은 모두 `True`로 평가합니다.

숫자 자료형

- 정수 자료형, 실수 자료형, 복소수 자료형을 사용할 수 있습니다.
- `1`은 정수 자료형이며, `1.0`은 실수 자료형입니다.
 - `1`과 `1.0`은 값은 같습니다. 즉 `1 == 1.0`은 `True`입니다.
 - 하지만 `1`과 `1.0`은 동일하지 않습니다.

```
int_one = 1
real_one = 1.0
print(int_one == real_one)          # True
print(type(int_one), type(real_one)) # <class 'int'> <class 'float'>
```

- 숫자형 데이터끼리는 다음의 연산을 할 수 있습니다.
 - 덧셈 연산 - `+`
 - 뺄셈 연산 - `-`
 - 곱셈 연산 - `*`
 - 나눗셈 연산과 나머지 연산
 - `/` 연산은 항상 결과가 실수가 되는 나눗셈입니다.
 - `//` 연산은 내림나눗셈 연산입니다. 정수끼리 `//` 연산을 하면 결과는 항상 정수입니다.
 - `%` 연산은 나머지 연산입니다.
 - 거듭제곱 연산 - `**`
 - 사칙 연산의 우선 순위
 - 거듭제곱 연산 → 곱셈, 나눗셈, 나머지 연산 → 덧셈, 뺄셈 연산
 - 괄호를 사용해서 연산 순서를 조정할 수 있습니다. 단, 이 용도로는 소괄호 `()`만 사용할 수 있습니다.
 - 연산 결과의 자료형은 미리 결정되어 있습니다.
 - 덧셈, 뺄셈, 곱셈, 내림나눗셈 `//`, 나머지 연산 `%` - 피연산자가 모두 정수형인 경우에만 정수형 (나머지는 모두 실수형)

- 나머지 `/` 연산은 무조건 실수형입니다.
- 거듭 제곱 연산은 밑이 정수, 지수가 0 이상의 정수인 경우에만 정수형 (나머지는 모두 실수형)
- 비교 연산
 - 두 숫자에 대해서 `==` (같으면 `True`), `!=` (다르면 `True`), `>` (앞 피연산자가 크면 `True`), `<` (앞 피연산자가 작으면 `True`), `>=` (앞 피연산자가 같거나 크면 `True`), `<=` (앞 피연산자가 같거나 작으면 `True`) 등의 비교 연산을 할 수 있습니다.
 - 같거나 크다, 같거나 작다에서 등호의 순서에 주의해야 합니다. (`>=`, `<=` 사용 불가)

문자열 자료형

- 문자들의 연속으로 이루어진 데이터이며 텍스트를 다루는데 사용
- 문자열 데이터의 형식
 - `"` 또는 `'` 로 시작하고, 동일한 따옴표로 끝나는 형식의 데이터
 - 워드나 한글 워드 프로세스 등에서 코드를 작성한 후 복사-붙여넣기를 하면 안됩니다.
 - 길이가 0인 문자열도 만들 수 있고, 이를 빈 문자열이라고 합니다.
 - `""` 또는 `''`
 - 문자열은 반드시 한 줄로 구성되어야 합니다.
 - 문자열을 프로그램 코드에 두 줄에 걸쳐서 만들고 싶다면 이스케이프 시퀀스 `\` + 엔터를 사용합니다. 단 문자열에 줄바꿈이 추가되지 않습니다.
 - 문자열 내에 줄바꿈을 포함하려면 이스케이프 시퀀스 `\n` 을 사용합니다.
 - 작은 따옴표 `'` 가 포함된 문자열을 만들 때는 큰 따옴표 `"` 를 사용합니다.
 - 큰 따옴표 `"` 가 포함된 문자열을 만들 때는 작은 따옴표 `'` 를 사용합니다.
- 이스케이프 시퀀스 : `\` 로 시작하는 특수 효과 기호
 - `\"` - 큰 따옴표
 - `\'` - 작은 따옴표
 - `\\` - 백슬래시
 - `\` + 엔터 - 문자열을 다음 줄에 이어서 작성 (줄바꿈이 추가되지 않음)
 - `\t` - 탭 문자
 - `\b` - 백스페이스
- 문자열의 길이
 - 내장함수 `len()` 을 사용해서 문자열의 길이를 구할 수 있습니다.
 - 공백 문자 (빈칸)도 한 글자로 계산해 길이에 합산됩니다.
 - 빈 문자열의 길이는 0 입니다.
 - 보이는 것과 문자열의 길이는 다를 수 있습니다. → 이스케이프 시퀀스도 길이 1만큼 합산됩니다. (`\` + 엔터는 예외)
- 문자열 리터럴 접두사

- 원시 문자열 - 따옴표 앞에 `r` 을 덧붙이면 이스케이프 시퀀스를 무시하는 문자열을 만들 수 있습니다.
- 형식 문자열 - 따옴표 앞에 `f` 를 덧붙이면 문자열 내에 중괄호 `{ }` 를 열고 닫아, 그 안의 표현식의 결과 값을 문자열에 반영할 수 있습니다.
- 삼중 따옴표 문자열
 - `'''` 로 시작해 `'''` 로 끝나거나 `"""` 로 시작해 `"""` 로 끝나는 문자열
 - 문자열 내에 자유롭게 줄바꿈을 할 수 있고, 이 줄바꿈은 `\n` 으로 문자열에 반영
- 문자열 연산
 - 문자열 끼리 덧셈 연산이 가능합니다. (이어 붙이기)
 - 문자열과 정수를 서로 곱할 수 있습니다. (정수 만큼 문자열 반복해 이어 붙이기)
 - 문자열끼리 비교 연산을 사용할 수 있습니다.
 - 사전 순 배열 기준으로 앞에 오는 문자열이 더 작습니다.
 - 단 대문자와 소문자는 구분되며, 모든 대문자가 모든 소문자보다 작습니다.
 - 공백문자 → `!@#$%^&'()*+,-./` → `0123456789` → `;<=>?@` → 대문자 → `[\]^_`` → 소문자 → `{|}~` → 기타 언어 및 대부분의 특수 기호 순서
 - 대문자와 소문자를 구분하지 않고 문자열의 크기를 비교하려면 문자열의 `lower()` 또는 `upper()` 메서드를 사용해 대문자 또는 소문자로 일치시킨 후 비교하면 됩니다.
 - 문자열과 문자열이 아닌 것을 비교할 수 없습니다. 단 `==` 와 `!=` 연산은 가능합니다.
- 문자열은 인덱싱과 슬라이싱이 가능합니다.
 - 인덱스는 문자열의 순서대로의 번호입니다. 단 첫 번째 글자의 인덱스 번호는 1이 아니라 0입니다.
 - 음수 인덱스는 문자열의 역순대로의 번호입니다. 길이 n인 문자열의 마지막 글자의 음수 인덱스 번호는 -1이고, 첫 번째 글자의 음수 인덱스 번호는 -n 입니다.
 - 길이가 n인 문자열의 인덱스의 범위는 0에서 n - 1까지(양수 인덱스), -n에서 -1까지 (음수 인덱스)입니다.
 - 인덱싱에서 절대 이 범위 밖의 값을 사용하면 안됩니다.
 - 문자열 뒤에 `[k]` 를 덧붙이면 이 문자열의 k + 1번째 글자 하나로 구성된 문자열이 만들어집니다. (인덱싱)
 - 문자열 뒤에 `[s:e]` 를 덧붙이면 이 문자열에서 s + 1번째 글자부터 e 번째 글자까지로 구성된 부분 문자열이 만들어집니다. (인덱스 기준으로 s부터 e - 1까지, 슬라이싱)
 - `[s:e:d]` 를 덧붙이면 시작과 끝 기준은 동일하고, d 간격으로 부분 문자열을 만듭니다. 이때 d는 음수도 가능합니다. 음수인 경우 모든 기준의 방향이 반대로 바뀝니다.
 - 시작 및 종료 기준의 인덱스의 값에 음수 인덱스 값을 사용할 수 있습니다.
 - 시작 기준 `s` 는 생략할 수 있습니다. 이 경우 문자열의 처음부터 슬라이싱합니다.
 - 음수 간격 슬라이싱의 경우 문자열의 마지막부터를 의미합니다.
 - 종료 기준 `e` 도 생략할 수 있습니다. 이 경우 문자열의 마지막까지 슬라이싱합니다.
 - 음수 간격 슬라이싱의 경우 문자열의 제일 앞까지를 의미합니다.

- 슬라이싱의 범위 지정은 인덱스 범위 밖의 값을 사용할 수 있습니다.
- 단, 인덱싱으로 문자열 내의 글자를 바꾸는 것은 불가능합니다. “불변형” 데이터입니다.

2.3 기본 컬렉션 자료형

주요 컬렉션 자료형의 특성 일람

	리스트(list)	튜플(tuple)	세트(set)	딕셔너리(dictionary)
리터럴 형태	[요소0, 요소1, 요소2]	(요소0, 요소1, 요소2)	{요소0, 요소1, 요소2}	{키1:값1, 키2:값2, 키3:값2}
빈 객체	[]	()	set()	{}
항목의 순서	있음	있음	없음(1)	있음 (3)
항목 값 중복	가능	가능	불가능	가능 (4)
수정 가능	가능	불가능	가능 (제한적) (2)	가능
저장 자료형 제한	없음	없음	있음	있음 (5)

1. 일반적으로 사용되는 파이썬 릴리즈에서 세트는 자동으로 오름차순 정렬된 순서를 가진 것 처럼 보입니다. 하지만 세트의 정의 기준으로는 순서를 보장하지 않습니다.
2. 세트는 값을 바꿀 수 있는 가변형 객체입니다. 하지만 순서가 없는 세트의 특성 상 값을 추가, 삭제하는 것은 가능하지만 인덱싱으로 값을 변경하는 것은 불가능합니다.
3. 딕셔너리 항목의 순서는 파이썬 3.7 이후부터 공식적으로 보장됨
4. 딕셔너리의 값은 중복이 가능하지만 키는 중복될 수 없음
5. 딕셔너리의 값은 자료형 제한이 없으나 키는 해시가능한 객체만 사용할 수 있음

리스트

- 리스트(list)의 정의와 특징
 - 여러 데이터를 묶어서 관리하는 자료형 → 이를 컬렉션이라고 부릅니다.
 - 각각의 데이터를 '요소' 또는 '항목'이라고 부릅니다.
 - 길이와 순서가 있습니다.
 - 파이썬의 어떤 종류의 데이터든 리스트의 항목이 될 수 있습니다.
- 리스트의 형태
 - 여는 대괄호 `[` 로 시작, 닫는 대괄호 `]` 로 종료
 - 각 항목은 쉼표 `,` 로 구분
 - 빈 리스트 `[]` 도 만들 수 있습니다.
 - 하나의 리스트에 여러 종류의 자료형이 포함될 수 있습니다.
- 리스트의 길이(또는 리스트의 크기)
 - 포함된 항목의 수를 말합니다.
 - 내장함수 `len()` 을 사용해 리스트의 길이를 구할 수 있습니다.

- 빈 리스트는 길이가 0입니다.
- 리스트의 인덱싱과 슬라이싱
 - 인덱스는 리스트 항목의 순서대로의 번호입니다. 단 첫 번째 항목의 인덱스 번호는 1이 아니라 0입니다.
 - 음수 인덱스는 리스트 항목의 역순대로의 번호입니다. 길이 n 인 리스트의 마지막 항목의 음수 인덱스 번호는 -1 이고, 첫 번째 항목의 음수 인덱스 번호는 $-n$ 입니다.
 - 길이가 n 인 리스트의 인덱스의 범위는 0에서 $n - 1$ 까지(양수 인덱스), $-n$ 에서 -1 까지(음수 인덱스)입니다.
 - 인덱싱에서 절대 이 범위 밖의 값을 사용하면 안됩니다.
 - 리스트 뒤에 `[k]`를 덧붙이면 이 리스트의 $k + 1$ 번째 항목을 돌려줍니다. (인덱싱)
 - 리스트 변수에서 `인덱싱 = 값` 형식으로 해당 항목의 값을 바꿀 수 있습니다.
 - 문자열에서는 불가
 - 리스트 뒤에 `[s:e]`를 덧붙이면 이 리스트에서 $s + 1$ 번째 항목부터 e 번째 항목까지로 구성된 부분 리스트가 만들어집니다. (인덱스 기준으로 s 부터 $e - 1$ 까지, 슬라이싱)
 - `[s:e:d]`를 덧붙이면 시작과 끝 기준은 동일하고, d 간격으로 부분 리스트를 만듭니다. 이때 d 는 음수도 가능합니다. 음수인 경우 모든 기준의 방향이 반대로 바뀝니다.
 - 시작 및 종료 기준의 인덱스의 값에 음수 인덱스 값을 사용할 수 있습니다.
 - 시작 기준 `s`는 생략할 수 있습니다. 이 경우 리스트의 첫 번째 요소부터 슬라이싱합니다.
 - 음수 간격 슬라이싱의 경우 리스트의 마지막 요소부터를 의미합니다.
 - 종료 기준 `e`도 생략할 수 있습니다. 이 경우 리스트의 마지막 요소까지 슬라이싱합니다.
 - 슬라이싱의 범위 지정은 인덱스 범위 밖의 값을 사용할 수 있습니다.
 - `[:]`를 사용해 슬라이싱하면 원래 리스트와 동일하지만 참조가 분리된 새로운 리스트가 만들어집니다. (중요!)
- 리스트의 연산
 - 리스트 끼리 덧셈 연산이 가능합니다. (이어 붙이기)
 - 리스트와 정수를 서로 곱할 수 있습니다. (정수 만큼 리스트 반복해 이어 붙이기)
 - 리스트 끼리 크기를 비교할 수 있습니다.
 - 첫 번째 항목부터 차례대로 비교합니다. 두 리스트의 크기가 같고 포함된 모든 항목이 순서대로 동일하면 두 리스트의 크기는 같습니다.
 - 항목의 크기를 차례대로 비교하되, 항목이 서로 다르면 그 항목의 크기를 기준으로 두 리스트의 크기가 결정됩니다. (그 뒤의 항목의 크기는 고려하지 않습니다.)
 - 길이가 다른 두 리스트에서 비교할 수 있는 항목까지의 값이 모두 같다면, 길이가 긴 리스트의 크기가 더 큼니다.
 - 리스트와 리스트가 아닌 것을 아닌 것을 비교할 수 없습니다. 단 `==`와 `!=` 연산은 가능합니다.
 - 리스트끼리의 비교라 할 지라도 항목간 크기 비교가 불가능하다면 비교할 수 없습니다.
- 리스트의 조작

- 리스트의 특정 항목의 값을 인덱싱에 할당 연산을 사용해 바꿀 수 있습니다.
- 리스트에 항목을 추가하기
 - `append()` 메서드를 사용해 리스트에 값을 하나씩 추가할 수 있습니다. (제일 뒤에 추가)
 - 덧셈 연산으로 리스트에 값을 하나 또는 여러 개를 추가할 수 있습니다. (제일 뒤에 추가)
 - `insert()` 메서드를 사용해 리스트에 값을 추가할 위치를 지정해 값을 하나씩 추가할 수 있습니다.
- 리스트의 항목을 삭제하기
 - `remove()` 메서드를 사용해 지정한 값을 리스트에서 삭제할 수 있습니다.
 - 지정한 값이 여러 개라면 첫 번째 하나만 삭제합니다.
 - 지정한 값이 리스트에 없으면 예외가 발생합니다.
 - `del` 키워드를 사용해 특정 인덱스의 항목을 삭제할 수 있습니다.
 - `del` 은 키워드로 변수의 이름으로는 사용할 수 없습니다.
- 다차원 리스트
 - 리스트는 리스트를 요소로 가질 수 있습니다.
 - 리스트에 포함된 요소로서의 리스트가 아무리 많은 값을 가지더라도 바깥쪽 리스트의 관점에서는 단 하나의 요소에 불과합니다.
 - 리스트 원소의 리스트를 인덱싱할 수 있습니다.
 - 대괄호 `[]` 를 여러 번 중첩 사용해 찾아갈 수 있습니다.
 - 리스트 뿐 아니라 튜플도 리스트를 요소로 가질 수 있고, 딕셔너리도 값으로 리스트를 가질 수 있습니다.

튜플 - 다른 컬렉션과 달리 '불변형 객체'입니다.

- 튜플(tuple)의 정의와 특징
 - 여러 데이터를 묶어서 관리하는 자료형
 - 길이, 순서가 있고, 포함된 각각의 데이터를 '요소' 또는 '항목'이라고 부릅니다.
 - 인덱스, 인덱싱, 슬라이싱 등을 리스트와 동일하게 사용할 수 있습니다.
 - 단 인덱싱으로 값을 바꿀 수는 없습니다.
 - 수정이 불가능합니다.
 - 데이터 무결성 보장이 필요하거나
 - 효율적인 프로그램 동작에 도움이 됩니다.
- 리스트와의 차이점
 - 리스트와 달리 수정이 불가능한 자료형입니다.
 - 인덱싱으로 튜플의 요소의 값을 바꿀 수 없습니다.
 - 튜플에 요소를 추가하거나, 제거할 수 없습니다.
 - 딕셔너리의 키로 사용할 수 있습니다.

- 튜플의 형태

- 여는 소괄호 (로 시작, 닫는 소괄호) 로 종료
- 각 요소는 쉼표 , 로 구분
- 빈 튜플 () 도 만들 수 있습니다.
- 요소가 하나 뿐인 튜플도 만들 수 있습니다. 이 경우 첫 번째 요소 뒤에 쉼표 , 를 붙입니다.

```
# 3 하나의 요소를 가지는 튜플을 다음과 같이 만들 수 있습니다.
only_element_tuple = (3,)
# 다음은 (3)을 표현식이 계산한 결과로 정수 3이 변수에 저장됩니다.
not_valid_tuple = (3)
```

- 하나의 튜플에 여러 종류의 자료형이 포함될 수 있습니다.

- 튜플의 길이(또는 튜플의 크기)

- 포함된 요소의 수를 말합니다.
- 내장함수 len() 을 사용해 튜플의 길이를 구할 수 있습니다.
- 빈 튜플은 길이가 0입니다.

세트

- 세트(set)의 정의와 특징

- 여러 데이터를 묶어서 관리하는 자료형
- 각각의 데이터를 '요소' 또는 '항목'이라고 부를 수도 있고 집합처럼 '원소'라고 부를 수도 있습니다.
- 길이는 있지만 순서는 없습니다.
- 파이썬의 불변 객체만 값으로 가질 수 있습니다.
 - 정수, 실수, 문자열, 튜플은 가능
 - 리스트, 딕셔너리, 세트는 불가능
- 세트는 수학의 집합 개념과 매우 흡사합니다.

- 세트의 형태

- 여는 중괄호 { 로 시작, 닫는 중괄호 } 로 종료
- 각 항목은 쉼표 , 로 구분
- 빈 세트 set() 도 만들 수 있습니다.
 - 빈 세트는 {} 이 아닙니다. {} 는 빈 딕셔너리입니다.
- 하나의 세트에 여러 종류의 자료형이 포함될 수 있습니다.

- 세트의 길이(또는 세트의 크기)

- 포함된 항목의 수를 말합니다.
- 내장함수 len() 을 사용해 세트의 길이를 구할 수 있습니다.
- 빈 세트는 길이가 0입니다.

- 세트는 순서가 없으므로 인덱스의 개념이 없고, 인덱싱, 슬라이싱이 불가능합니다.
- 세트의 집합 연산 (아래의 메서드 표현과 연산 표현은 동일하게 동작합니다.)
 - 합집합 - 두 세트를 합친 세트를 만듭니다. 이때 중복된 요소는 제거하고 하나 만 유지합니다.
 - 메서드 표현 : `set1.union(set2)`
 - 연산 표현 : `set1 | set2`
 - 교집합 - 두 세트의 중복된 공통 요소 만으로 세트를 만듭니다.
 - 메서드 표현 : `set1.intersection(set2)`
 - 연산 표현 : `set1 & set2`
 - 차집합 - 한 세트에서 다른 세트에 포함된 요소를 제거한 세트를 만듭니다.
 - 메서드 표현 : `set1.difference(set2)`
 - 연산 표현 : `set1 - set2`
 - 여집합 - 두 세트에서 중복되지 않은 요소 만으로 세트를 만듭니다.
 - 메서드 표현 : `set1.symmetric_difference(set2)`
 - 연산 표현 : `set1 ^ set2`

```
set1 = {1, 2, 3, 4, 5}
set2 = {1, 3, 5, 7, 9}
print(f"합집합 : {set1 | set2}")      # 중복을 제거하고 합침 : {1, 2, 3, 4, 5, 7, 9}
print(f"교집합 : {set1 & set2}")      # 겹치는 원소의 세트 : {1, 3, 5}
print(f"차집합 : {set1 - set2}")      # set1에만 있는 원소의 세트 : {2, 4}
print(f"여집합 : {set1 ^ set2}")      # 두 세트에서 겹치지 않는 원소의 세트 : {2, 4, 7, 9}
print(set1, set2)                    # 집합 연산은 원래 세트를 바꾸지 않습니다.
```

- 세트의 조작
 - 세트는 인덱싱이 불가하므로 인덱싱의 형태로 특정 항목의 값을 바꿀 수 없습니다.
 - 세트에 요소를 추가하기
 - `add()` 메서드를 사용해 세트에 하나의 요소를 추가할 수 있습니다.
 - `update()` 메서드를 사용해 세트에 여러 개의 요소를 추가할 수 있습니다.
 - `update()` 메서드의 인자는 리스트, 세트, 튜플 등을 사용할 수 있습니다.
 - 세트의 요소를 삭제하기
 - `remove()` 메서드를 사용해 지정한 값을 세트에서 삭제할 수 있습니다.
 - 만약 지정한 값이 없으면 예외가 발생합니다.
 - `discard()` 메서드를 사용해 지정한 값을 세트에서 삭제할 수 있습니다.
 - 지정한 값이 없더라도 예외가 발생하지 않습니다.

- `pop()` 메서드 - 세트에서 무작위로 아무 값이나 하나를 제거하고, 제거한 값을 반환합니다.
- `clear()` 메서드 - 세트의 모든 원소를 삭제하고 빈 세트를 만듭니다.

딕셔너리

- 딕셔너리(dictionary)의 정의와 특징
 - 여러 데이터를 묶어서 관리하는 자료형
 - 각각의 데이터(이를 값이라고 합니다.)에는 고유한 키가 부여됩니다.
 - 이 키를 사용해 검색이 가능합니다.
 - 즉 딕셔너리는 키-값의 쌍을 묶어서 관리하는 자료형입니다.
 - 길이와 순서가 있습니다.
 - 딕셔너리의 순서는 딕셔너리에 키-값 쌍을 추가한 순서이며 파이썬 3.7 버전 이후부터 보장됩니다.
 - 단, 리스트, 튜플 처럼 인덱스 기준으로 인덱싱을 할 수 없습니다.
 - 키는 반드시 불변 객체여야 합니다.
 - 해시가능Hashable이라고도 하며, 이 문제로 예외가 발생하면 예외의 정보에 "Hashable"이라는 표현이 포함되므로 이 단어를 기억해 둡시다.
 - 값의 형식은 제한이 없습니다.
- 리스트의 형태
 - 여는 중괄호 `{` 로 시작, 닫는 중괄호 `}` 로 종료
 - 각 요소의 키와 값은 콜론 `:` 으로 분리
 - 각 요소는 쉼표 `,` 로 구분
 - 빈 딕셔너리 `{}` 도 만들 수 있습니다.
 - 하나의 딕셔너리에 여러 종류 자료형의 키, 값이 포함될 수 있습니다.
- 딕셔너리의 길이(또는 딕셔너리의 크기)
 - 포함된 요소(키-값 쌍)의 수를 말합니다.
 - 내장함수 `len()` 을 사용해 딕셔너리의 길이를 구할 수 있습니다.
 - 빈 딕셔너리는 길이가 0입니다.
- 딕셔너리의 인덱싱
 - 리스트의 인덱싱과 달리 인덱스 번호는 없습니다.
 - 인덱스 번호 대신 키의 값을 대괄호 `[]` 안에 사용하면 이 키에 해당되는 값을 반환합니다.
 - 존재하지 않는 키를 사용해 인덱싱을 하면 예외가 발생합니다.
 - 키의 값을 사용한 인덱싱을 대상으로 값을 할당하면 키의 존재 여부에 따라 다음과 같이 딕셔너리가 변경됩니다.
 - 키가 존재하지 않으면 - 이 키에 해당하는 키-값 쌍을 새롭게 추가됩니다.
 - 키가 이미 존재하면 - 이 키에 해당되는 값이 변경됩니다. (동일한 값의 키가 추가될 수 없습니다.)

- 딕셔너리로는 슬라이싱을 할 수 없습니다.
- 딕셔너리의 정보를 가지고 오는 메서드
 - `key()` 메서드는 딕셔너리에 포함된 모든 키의 목록을 `dict_keys` 자료형 데이터로 반환하며, 이 자료형은 리스트로 형변환할 수 있습니다.
 - `values()` 메서드는 딕셔너리에 포함된 모든 값의 목록을 `dict_values` 자료형 데이터로 반환하며, 이 자료형은 리스트로 형변환할 수 있습니다.
 - `items()` 메서드는 딕셔너리에 포함된 모든 키와 값의 쌍을 튜플로 만들어 구성된 목록을 `dict_items` 자료형 데이터로 반환하며, 이 자료형은 리스트로 형변환할 수 있습니다.
 - `get()` 메서드는 인자로 지정한 키에 해당하는 값을 반환합니다. 만약 해당하는 키가 딕셔너리에 없는 경우는 `None` 을 반환합니다.
 - `get()` 메서드에 두 번째 인자를 지정할 수 있습니다. 두 번째 인자를 지정하면 키가 없는 경우 이 두 번째 인자를 반환합니다.
- 딕셔너리의 조작
 - 딕셔너리의 항목을 추가하거나 수정하기
 - 존재하지 않는 키를 사용한 인덱싱에 할당 연산을 하면 딕셔너리에 새로운 항목을 추가할 수 있습니다.
 - 존재하는 키를 사용한 인덱싱에 할당 연산을 하면 딕셔너리의 내용을 변경할 수 있습니다. 해당 키의 값을 바꿉니다.
 - `update()` 메서드를 사용해 인자로 주어진 다른 딕셔너리를 원래 딕셔너리에 더할 수 있습니다.
 - 이때 키가 중복되어 충돌이 발생하면 인자로 사용한 딕셔너리의 값으로 원래 딕셔너리의 값을 변경합니다.
 - 딕셔너리의 항목을 삭제하기
 - `pop()` 메서드를 사용해 인자로 주어진 키의 키-값 쌍을 삭제할 수 있습니다.
 - 이때 삭제하는 키의 값을 반환합니다.
 - `del` 키워드로 키-값 쌍을 삭제할 수도 있습니다.
- 딕셔너리로 루프 생성
 - `for 루프변수 in dict.keys():` 형태를 사용해 딕셔너리 `dict` 의 각각의 키를 순서대로 사용하는 루프를 만들 수 있습니다.
 - `for 루프변수 in dict.values():` 형태를 사용해 딕셔너리 `dict` 의 각각의 값을 순서대로 사용하는 루프를 만들 수 있습니다.
 - `for 루프변수 in dict.items():` 형태를 사용해 딕셔너리 `dict` 의 각각의 키-값 튜플을 순서대로 사용하는 루프를 만들 수 있습니다.
 - 이 경우 `루프변수` 의 자료형은 튜플이 됩니다.
 - `for 루프변수_키, 루프변수_값 in dict.items():` 형태를 사용해 딕셔너리 `dict` 의 각각의 대응되는 키와 값을 순서대로 사용하는 루프를 만들 수 있습니다.
 - 이 경우 `루프변수_키` 에는 키가 할당되며, `루프변수_값` 에는 값이 할당됩니다.

- `for 루프변수 in dict:` 는 `for 루프변수 in dict.keys():` 와 동일합니다.

컴프리헨션

- 리스트 컴프리헨션
 - 리스트를 간결하고 효율적으로 생성하기 위한 문법입니다.
 - 기존의 리스트를 기반으로 다른 리스트를 만들거나, 특정 조건의 요소만 선택해서 생성할 수 있습니다.
 - `[표현식 for 요소변수 in 이터러블객체 if 조건표현식]` 의 기본형태를 가집니다.
 - 이터러블객체(리스트, 튜플, 문자열 등)의 각각의 요소가 차례대로 요소 변수에 할당됩니다.
 - 이 요소변수를 사용한 조건표현식의 참거짓을 평가합니다.
 - 참인 경우 : 요소변수를 사용한 표현식이 새로운 리스트의 요소에 포함됩니다.
 - 거짓인 경우 : 아무 일도 일어나지 않습니다.
 - 새로운 리스트의 요소를 모아서 리스트를 생성합니다.
 - `if 조건표현식` 은 생략할 수 있습니다. 이 경우 이터러블객체의 요소 전체를 사용합니다.
- 세트 컴프리헨션
 - 리스트 컴프리헨션과 거의 동일합니다.
 - `{표현식 for 요소변수 in 이터러블객체 if 조건표현식}` 의 기본형태를 가집니다.
 - 리스트 컴프리헨션과 거의 동일하게 세트를 생성하는데 단 값이 중복되면 하나만 원소로 갖습니다.
 - `if 조건표현식` 은 생략할 수 있습니다. 이 경우 이터러블객체의 요소 전체를 사용합니다.
- 딕셔너리 컴프리헨션
 - 리스트, 세트 컴프리헨션과 거의 동일합니다.
 - `{키_표현식:값_표현식 for 요소변수 in 이터러블객체 if 조건표현식}` 의 기본형태를 가집니다.
 - 리스트 컴프리헨션과 거의 동일하게 딕셔너리를 생성합니다.
 - 키 표현식으로 만들어진 키와 값 표현식으로 만들어진 값의 쌍으로 딕셔너리를 구성합니다.
 - 주의사항 1 : 키가 중복된 경우, 키 - 값 쌍 생성 순서 기준으로 가장 마지막에 만들어진 값으로 생성됩니다. (덮어 쓰기)
 - 주의사항 2 : 리스트 등 키로 사용할 수 없는 자료형을 키로 지정하려 하면 예외가 발생합니다.
 - `if 조건표현식` 은 생략할 수 있습니다. 이 경우 이터러블객체의 요소 전체를 사용합니다.
- 조건 표현식 연산(`if ~ else` 연산)을 컴프리헨션과 함께 사용하면 유용합니다.

2.4 표현식, 연산과 연산자

표현식(Expression)

- 의미
 - 값을 반환하는 코드의 일부를 뜻합니다.

- 표현식은 파이썬 프로그램의 기본 구성 요소입니다.
- 표현식은 아무리 복잡하더라도 처리되고 나면 하나의 '값'이 됩니다.
- 표현식의 구성
 - 리터럴, 변수가 하나만 있는 경우도 이를 표현식이라 말할 수 있습니다.
 - 함수를 호출하는 부분도 표현식입니다.
 - 이 경우 호출된 함수의 동작과 별개로 함수가 반환한 값이 표현식의 결과입니다.
 - 연산자를 사용해 계산하는 식도 표현식입니다.
 - 비교 연산이나 논리 연산의 결과도 `True` 또는 `False` 둘 중 하나를 결과로 하는 표현식입니다.
 - 함수를 호출한 후, 그 반환 결과를 사용해 추가의 연산을 하는 경우도 표현식이라 할 수 있습니다.

괄호에 대해서

- 괄호는 표현식 내에서 사용
- 대괄호 `[]` 의 용도
 - 리스트 리터럴을 정의할 때 사용합니다.
 - 값으로 리스트를 정의하는 경우
 - 컴프리헨션으로 리스트를 정의하는 경우
 - 리스트를 표현할 때 사용합니다.
 - 리스트나 문자열 등의 인덱싱에 사용합니다.
 - 리스트나 문자열 등의 슬라이싱에 사용합니다.
 - 딕셔너리에서 키를 사용해 값에 접근할 때 사용합니다.
- 중괄호 `{ }` 의 용도
 - 딕셔너리 리터럴을 정의할 때 사용합니다.
 - 값으로 딕셔너리를 정의하는 경우
 - 컴프리헨션으로 딕셔너리를 정의하는 경우
 - 세트 리터럴을 정의할 때 사용합니다.
 - 값으로 세트를 정의하는 경우 - 단 빈 세트는 중괄호를 사용해 정의할 수 없습니다.
 - 컴프리헨션으로 세트를 정의하는 경우
 - 딕셔너리나 세트를 표현할 때 사용합니다.
- 소괄호 `()` 의 용도
 - 함수를 호출할 때 인자를 전달하기 위해서 사용합니다.
 - 함수를 정의할 때 매개변수를 선언하기 위해서 사용합니다.
 - 튜플을 정의할 때 사용합니다.
 - 값으로 튜플을 정의하는 경우

- 튜플은 컴프리헨션으로 직접 생성할 수 없습니다. 단, 컴프리헨션과 유사한 표현으로 제너레이터를 만들 수 있고, 이를 형변환 하여 튜플로 만들 수 있습니다. - 이번 강의 범위에 포함되지 않습니다.
- 클래스를 상속할 때 슈퍼 클래스를 지정하는 용도로 사용합니다.
- 연산자의 우선순위를 조정하는 용도로 사용합니다.

단항 연산, 이항 연산과 삼항 연산

- 피연산자를 하나만 가지는 연산을 단항 연산이라고 합니다.
 - `+` (단항 플러스 연산자), `-` (단항 마이너스 연산자), `not` (논리 not 연산자), `~` (비트 반전 연산자)
 - 항상 연산자 뒤에 피연산자가 위치합니다.
 - 연산자는 항상 뒤에 위치한 피연산자와 결합합니다.
- 피연산자가 셋인 연산을 삼항 연산이라고 합니다.
 - 파이썬의 삼항 연산은 `if ~ else` 연산 하나 밖에 없습니다.
 - 여기서 `if ~ else` 는 조건 제어와 다른 연산자로 사용되는 `if ~ else` 입니다.
 - `피연산자1 if 피연산자2 else 피연산자3` 형태로 사용하며
 - `피연산자2` 이 참 `True` 이면 `피연산자1` 가 결과가 되고
 - `피연산자2` 이 거짓 `False` 이면 `피연산자3` 이 결과가 됩니다.
- 피연산자가 둘인 연산을 이항 연산이라고 합니다.
 - 위에서 언급한 단항 연산과 삼항 연산을 제외한 모든 연산은 이항 연산입니다.
 - 거의 모든 이항 연산의 연산 순서는 왼쪽 피연산자 → 오른쪽 피연산자 순서입니다. (연산의 방향이 왼쪽에서 오른쪽)
 - 예외 1. 거듭제곱 연산의 연산 순서는 오른쪽 피연산자 → 왼쪽 피연산자입니다. (연산의 방향이 오른쪽에서 왼쪽)
 - 예외 2. 할당 연산, 복합 할당 연산의 연산 순서는 오른쪽 피연산자 → 왼쪽 피연산자입니다. (연산의 방향이 오른쪽에서 왼쪽)

산술 연산

- 다음과 같은 산술 연산자를 사용할 수 있습니다.

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
1	<code>**</code>	이항연산	거듭제곱 연산	<code>2 ** 3</code>	2를 세제곱하는 연산
2	<code>+</code>	단항연산	단항 플러스 연산	<code>+operand</code>	<code>operand</code> 의 부호를 유지
2	<code>-</code>	단항연산	단항 마이너스 연산	<code>-operand</code>	<code>operand</code> 의 부호를 바꿈
3	<code>*</code>	이항연산	곱셈 연산	<code>2 * 3</code>	2 곱하기 3
3	<code>/</code>	이항연산	나눗셈 연산	<code>4 / 2</code>	4 나누기 2 (결과는 2.0)

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
3	//	이항연산	내림나눗셈 연산	4 // 2	4 나누기 2 후 내림 (결과는 2)
3	%	이항연산	나머지 연산	5 % 2	5를 2로 나눈 나머지
4	+	이항연산	덧셈 연산	2 + 3	2 더하기 3
4	-	이항연산	뺄셈 연산	2 - 3	2 빼기 3

비트 연산



강의 환경 문제로 자세히 설명하지 않습니다. 동영상 강의를 참조하세요.

- 다음과 같은 비트 연산자를 사용할 수 있습니다.

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
2	~	단항연산	비트 반전 연산	~10	10의 이진수 표현에서 0과 1을 1과 0으로 바꿈 (결과는 -11)
5	<<	이항연산	비트 왼쪽 시프트 연산	35 << 2	35의 이진수 표현에서 낮은 자리 (오른쪽)에 0을 2개 덧붙임 (결과는 140)
5	>>	이항연산	비트 오른쪽 시프트 연산	35 >> 2	35의 이진수 표현에서 낮은 자리 (오른쪽) 2개를 삭제 (결과는 8)
6	&	이항연산	비트 and 연산	36 & 57	36과 57의 이진수 표현에서 자릿수를 맞춘 후 양쪽 모두 1인 자리는 1, 나머지 자리는 0인 값으로 계산 (결과는 32)
7	^	이항연산	비트 xor 연산	38 ^ 44	38과 44의 이진수 표현에서 자릿수를 맞춘 후 두 자리의 값이 같으면 0, 다르면 1 (결과는 10)
8		이항연산	비트 or 연산	38 44	38과 44의 이진수 표현에서 자릿수를 맞춘 후 양쪽 모두 0인 자리는 0, 나머지 자리는

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
					1인 값으로 계산 (결과는 46)

비교 연산

- 다음과 같은 비교 연산자를 사용할 수 있습니다.

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
9	<code>in</code>	이항연산	포함 검사 연산 (또는 멤버십 검사 연산)	<code>5 in [1, 2, 3, 4, 5]</code>	<code>[1, 2, 3, 4, 5]</code> 에 <code>5</code> 가 포함되어 있는가? (결과는 <code>True</code>)
9	<code>not in</code>	이항연산	포함 검사 연산	<code>5 not in [1, 2, 3, 4, 5]</code>	<code>[1, 2, 3, 4, 5]</code> 에 <code>5</code> 가 포함되어 있지 않은가? (결과는 <code>False</code>)
9	<code>is</code>	이항연산	동일성 검사 연산자	<code>x is y</code>	x와 y가 같은 대상을 참조하는가?
9	<code>is not</code>	이항연산	동일성 검사 연산자	<code>x is not y</code>	x와 y가 다른 대상을 참조하는가?
9	<code><</code>	이항연산	크기 비교 연산	<code>x < y</code>	x가 y보다 작은가?
9	<code><=</code>	이항연산	크기 비교 연산	<code>x <= y</code>	x가 y보다 작거나 같은가?
9	<code>></code>	이항연산	크기 비교 연산	<code>x > y</code>	x가 y보다 큰가?
9	<code>>=</code>	이항연산	크기 비교 연산	<code>x >= y</code>	x가 y보다 크거나 같은가?
9	<code>!=</code>	이항연산	크기 비교 연산	<code>x != y</code>	x와 y의 값이 다른가?
9	<code>==</code>	이항연산	크기 비교 연산	<code>x == y</code>	x와 y의 값이 같은가?

- 비교 연산의 결과는 항상 `True`, `False` 둘 중 하나입니다.
- `is` 연산과 `is not` 연산, `in` 연산자와 `not in` 연산의 결과는 항상 각각 반대입니다.
- `is not` 연산자는 `not`이 뒤에, `not in` 연산자는 `not`이 앞에 와야 합니다.
- 크기 비교 연산
 - `==`, `!=` 두 연산은 피연산자의 종류가 달라도 사용할 수 있습니다.
 - 나머지 크기 비교 연산은 두 피연산자의 종류가 다르면 대부분의 경우 사용할 수 없습니다.
 - 문자열과 리스트의 크기를 비교하는 방식은 문자열, 리스트의 설명을 참고하세요
 - 숫자 간의 크기 비교 연산은 값을 기준으로 비교합니다.
 - `1 == 1.0`은 정수형과 실수형의 비교이지만 `True`입니다.

- 포함 검사 연산 (또는 멤버십 연산)
 - 포함 검사 연산의 오른쪽 피연산자에는 여러 요소를 포함할 수 있는 대상이 위치합니다.
 - 문자열, 리스트, 튜플, 딕셔너리, 세트, range 객체 등
 - 오른쪽 피연산자에 왼쪽 값이 포함되어 있는지를 검사합니다.
 - 문자열을 대상으로 검사할 때는 부분 문자열 여부를 검사합니다.

```
print("p" in "python") # True
print("P" in "python") # False - 대소문자 구분
print("py" in "python") # True
print("pn" in "python") # False
# 부분문자열 검사와 글자의 포함여부는 다릅니다.
```

- 나머지는 요소가 포함되어 있는지의 여부를 검사합니다.

```
print(1 in [1, 2, 3]) # True
print([1] in [1, 2, 3]) # False
print([1, 2] in [1, 2, 3]) # False
# [1, 2, 3]에 1이 포함되어 있지 [1]은 포함되어 있지 않습니다.
# [1, 2]도 마찬가지입니다.
print([1, 2] in [[1, 2], 3]) # True
```

- 동일성 검사 연산
 - 두 객체가 동일한지를 검사하는 연산이며 값이 같은지는 중요하지 않습니다.

```
arr = [1, 2, 3]
other_arr = [1, 2, 3]
referenced_arr = arr
print(arr == other_arr) # 값이 같으므로 True
print(arr == referenced_arr) # 값이 같으므로 True
print(arr is other_arr) # 값이 같아도 다른 객체이므로 False
print(arr is referenced_arr) # 같은 객체이므로 True

arr.append(4)
print(arr) # [1, 2, 3, 4]
print(other_arr) # [1, 2, 3] -> 다른 객체이므로 arr 변경에 영향을 받지 않음
print(referenced_arr) # [1, 2, 3, 4] -> 같은 객체이므로 arr 변경에 영향을 받음
```

논리 연산

- 다음과 같은 논리 연산자를 사용할 수 있습니다.

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
10	<code>not</code>	단항연산	not 연산	<code>not x</code>	x가 <code>True</code> 로 평가되면 <code>False</code> , <code>False</code> 로 평가되면 <code>True</code>
11	<code>and</code>	이항연산	포함 검사 연산	<code>x and y</code>	x와 y 모두 <code>True</code> 로 평가되면 <code>True</code> , 이외의 경우 <code>False</code>
12	<code>or</code>	이항연산	동일성 검사 연산자	<code>x or y</code>	x와 y 모두 <code>False</code> 로 평가되면 <code>False</code> , 이외의 경우 <code>True</code>

- `not`, `and`, `or` 은 우선순위가 다릅니다!
- `and` 연산의 참거짓표(진리표) - “그리고”의 의미

x	y	x and y
<code>False</code>	<code>False</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>False</code>
<code>True</code>	<code>False</code>	<code>False</code>
<code>True</code>	<code>True</code>	<code>True</code>

- `or` 연산의 참거짓표(진리표) - “또는”의 의미

x	y	x or y
<code>False</code>	<code>False</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>True</code>
<code>True</code>	<code>True</code>	<code>True</code>

- `and` 연산의 참거짓표(진리표)

x	not x
<code>False</code>	<code>True</code>
<code>True</code>	<code>False</code>

- 논리 연산은 ‘평가’를 사용합니다.
 - 논리 연산의 피연산자는 참거짓값이 아니어도 됩니다.
 - 피연산자가 참거짓값이 아니면 피연산자를 다음과 같이 평가하고, 그 결과를 연산에 반영합니다.
 - `False` 로 평가 (Falsy라고 함) : `False`, `None`, `0`, `0.0`, `""` (빈 문자열), `[]` (빈 리스트), `()` (빈 튜플), `{}` (빈 딕셔너리), `set()` (빈 세트) 등
 - `True` 로 평가 (Truthy라고 함) : `True`, `0` 이 아닌 숫자, 비어 있지 않은 문자열, 비어 있지 않은 리스트, 비어 있지 않은 튜플, 비어 있지 않은 딕셔너리, 비어 있지 않은 세트 등

```
def evaluation(x):
    if x:
        print(f"{x}는 Truthy")
    else:
        print(f"{x}는 Falsy")

empty = []
pseudo_empty = [[]]
evaluation(empty)      # []는 Falsy
evaluation(pseudo_empty) # [[]]는 Truthy
```

- **and** 연산이 진짜로 하는 일
 - 왼쪽 피연산자가 Truthy이면 오른쪽 피연산자의 값을 반환
 - 왼쪽 피연산자가 Falsy이면 오른쪽 피연산자는 무시하고 (실행도 하지 않고) 왼쪽 피연산자의 값을 반환

```
print(10 and 20)      # 10이 Truthy이므로 20을 반환
print(bool(10 and 20)) # 20은 True로 평가
print(0 and 10)       # 0이 Falsy이므로 0을 반환
print(bool(0 and 10))  # 0은 False로 평가
print(0 and 10 / 0)    # 0이 Falsy이므로 10 / 0은 무시됨 -> 0으로 나눌 때
                        # 의 예외가 발생하지 않음
print(5 and 10 / 0)    # 왼쪽 피연산자가 Truthy이므로 오른쪽을 확인해야 함 -
                        # > 예외 발생
```

- **or** 연산이 실제로 하는 일
 - 왼쪽 피연산자가 Falsy이면 오른쪽 피연산자의 값을 반환
 - 왼쪽 피연산자가 Truthy이면 오른쪽 피연산자는 무시하고 (실행도 하지 않고) 왼쪽 피연산자의 값을 반환

```
print(0 or 0)         # (왼쪽)0이 Falsy이므로 (오른쪽)0을 반환
print(bool(0 or 0))   # 0은 False로 평가
print(5 or 0)         # 5는 Truthy이므로 5를 반환
print(bool(5 or 0))   # 5는 True로 평가
print(5 or 10 / 0)    # 5가 Truthy이므로 10 / 0은 무시됨 --> 0으로 나눌 때
                        # 의 예외가 발생하지 않음
```

- 그래서 논리 연산 일람표를 다음과 같이 고쳐 쓸 수 있습니다.

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
10	<code>not</code>	단항연산	not 연산	<code>not x</code>	x가 <code>True</code> 로 평가되면 <code>False</code> ,

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
					<code>False</code> 로 평가되면 <code>True</code>
11	<code>and</code>	이항연산	포함 검사 연산	<code>x and y</code>	x, y 모두 <code>True</code> 인 경우에만 <code>True</code> , 이외에는 <code>False</code> 실제로는 x가 <code>True</code> 로 평가되면 y, x가 <code>False</code> 로 평가되면 x
12	<code>or</code>	이항연산	동일성 검사 연산자	<code>x or y</code>	x, y 모두 <code>False</code> 인 경우에만 <code>False</code> , 이외는 <code>True</code> 실제로는 x가 <code>True</code> 로 평가되면 x, <code>False</code> 로 평가되면 y

기타 연산, 할당 연산

- 기타로 다음 사항도 알아두면 좋습니다.

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
13	<code>if ~ else</code>	삼항연산	조건 표현식 연산	<code>condition if x else y</code>	x가 <code>True</code> 로 평가되면 <code>False</code> , <code>False</code> 로 평가되면 <code>True</code>
14	<code>lambda</code>	익명함수 표현식	없음 (연산이 아님)	<code>lambda x: x ** 2</code>	인자를 제공한 결과를 반환하는 익명함수
15	<code>:=</code>	이항연산	할당 표현식 연산	<code>variable := x</code>	<code>variable</code> 에 x를 할당하고, x의 값을 반환
마지막	<code>=</code>	이항연산	할당 연산	<code>x = 10</code>	x에 정수 10을 할당
마지막	<code>+=</code>	이항연산	덧셈 복합할당 연산	<code>x += 5</code>	<code>x + 5</code> 의 값을 변수 x에 할당
마지막	<code>-=</code>	이항연산	뺄셈 복합할당 연산	<code>x -= 5</code>	<code>x - 5</code> 의 값을 변수 x에 할당
마지막	<code>*=</code>	이항연산	곱셈 복합할당 연산	<code>x *= 5</code>	<code>x * 5</code> 의 값을 변수 x에 할당
마지막	<code>/=</code>	이항연산	나눗셈 복합할당 연산	<code>x /= 5</code>	<code>x / 5</code> 의 값을 변수 x에 할당
마지막	<code>//=</code>	이항연산	내림나눗셈 복합할당 연산	<code>x //= 5</code>	<code>x // 5</code> 의 값을 변수 x에 할당

우선순위	연산자	종류	연산의 이름	예시	예시의 의미
마지막	<code>%=</code>	이항연산	나머지 복합할당 연산	<code>x %= 5</code>	<code>x % 5</code> 의 값을 변수 x에 할당
마지막	<code>**=</code>	이항연산	거듭제곱 복합할당 연산	<code>x **= 5</code>	<code>x ** 5</code> 의 값을 변수 x에 할당
마지막	<code>&=</code>	이항연산	비트 and 복합할당 연산	<code>x &= 5</code>	<code>x & 5</code> 의 값을 변수 x에 할당
마지막	<code> =</code>	이항연산	비트 or 복합할당 연산	<code>x = 5</code>	<code>x 5</code> 의 값을 변수 x에 할당
마지막	<code>^=</code>	이항연산	비트 xor 복합할당 연산	<code>x ^= 5</code>	<code>x ^ 5</code> 의 값을 변수 x에 할당
마지막	<code><<=</code>	이항연산	비트 왼쪽시프트 복합할당 연산	<code>x <<= 5</code>	<code>x << 5</code> 의 값을 변수 x에 할당
마지막	<code>>>=</code>	이항연산	비트 오른쪽시프트 복합할당 연산	<code>x >>= 5</code>	<code>x >> 5</code> 의 값을 변수 x에 할당

- `lambda` 는 이번 강의에서 다루지 않으며, 연산자도 아닙니다. 다만 표현식 내에 위치했을 때의 우선 순위를 설명하기 위해서 추가합니다.
- `:=` 는 할당 연산과 유사하지만, 할당한 값을 반환하는 실제 연산자로 동작합니다.
 - 이 할당 표현식 연산은 왈러스 연산(Walrus 연산)이라고도 하며, 파이썬 3.8 버전에 도입되어 그 이전 버전에서는 사용할 수 없습니다.

```
if (x := 5) < 10:
    print(f"역시 {x}는 10보다 크다.")
```

- 이에 비해 할당연산 `=` 과 복합 할당 연산은 할당 용도 이외로는 표현식 내에서 사용할 수 없습니다.

```
# 그냥 할당 연산은 표현식 중간에 사용할 수 없습니다.
if (x = 5) < 10:
    print("어짜피 예외가 발생할건데 아무말이나...")
```

- 대신 할당 표현식 연산은 할당 연산의 용도로만 사용하는 경우 예외가 발생합니다.

```
y = (x := 10) + 5    # x에 10을 할당하고, 그 값인 10에 5를 더하는 용도로 사용
                    # 하므로 정상
print(x, y)          # 10 15 출력
x := 10              # 할당 용도로만 사용했으므로 예외
y = x := 10 + 5      # := 우선순위가 +보다 낮으므로 할당 용도로만 사용됩니다. -
> 예외 발생
```

연산의 우선순위

- 괄호 `()` 가 가장 먼저 계산됩니다.
 - 괄호가 중첩된 경우 안쪽 괄호가 먼저 계산됩니다.

- 괄호의 영향을 받지 않는 한 파이썬 연산은 각 연산자가 가진 우선 순위에 따라 연산 순서가 결정됩니다.
- 같은 연산 순위를 가진 연산자끼리는 연산자의 연산 방향을 반영합니다.
 - 거의 모든 연산자는 왼쪽에서 오른쪽 방향으로 연산이 진행됩니다.
 - 단항 연산자, 그리고 거듭제곱 연산자는 오른쪽에서 왼쪽 방향으로 연산이 진행됩니다.
- 연산자는 일반적으로 산술 연산 → 비트 연산 → 비교 연산 → 논리 연산 → 할당 표현식 연산 → 할당 연산 순으로 우선 순위를 갖습니다.
 - 할당 연산은 연산 여부를 떠나 표현식에서 가장 낮은 우선순위를 가지는 연산입니다.
 - 예외로 비트 반전 연산은 비트 연산임에도 불구하고 산술 연산의 `+`, `-` (단항 연산)과 동일한 우선 순위를 갖습니다.
 - `and`, `or` 두 연산은 (아닌 것 같지만) 우선 순위가 다릅니다. 이를 고려하지 않으면 의도와 다른 결과가 나올 수 있습니다.
 - 그러므로 `and`, `or` 연산자를 둘 이상 사용하는 경우 괄호를 사용해 우선 순위를 결정해 두는 것이 안전합니다.

```
print(True or False and False)
# False and False를 먼저 계산하고 (결과는 False)
# 그 다음 True or False를 계산하므로 최종 결과는 True
print((True or False) and False)
# 괄호를 우선으로 True or False를 먼저 계산하고 (결과는 True)
# 그 다음 True and False를 계산하므로 최종 결과는 False
```

- 연산의 우선 순위를 기준으로 연산자 표를 발췌 취합하면 다음과 같습니다.

우선순위	연산의 요소	연산의 종류	연산의 분류	연산의 방향
0	<code>()</code>	괄호	연산이 아님	없음
1	<code>**</code>	이항연산	산술 연산	오른쪽 → 왼쪽
2	<code>+</code> , <code>-</code> , <code>~</code>	단항연산	산술 연산, 비트 연산	오른쪽 → 왼쪽
3	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>	이항연산	산술 연산	왼쪽 → 오른쪽
4	<code>+</code> , <code>-</code>	이항연산	산술 연산	왼쪽 → 오른쪽
5	<code><<</code> , <code>>></code>	이항연산	비트 연산	왼쪽 → 오른쪽
6	<code>&</code>	이항연산	비트 연산	왼쪽 → 오른쪽
7	<code>^</code>	이항연산	비트 연산	왼쪽 → 오른쪽
8	<code> </code>	이항연산	비트 연산	왼쪽 → 오른쪽
9	<code>in</code> , <code>not in</code> , <code>is</code> , <code>is not</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>!=</code> , <code>==</code>	이항연산	비교 연산	왼쪽 → 오른쪽
10	<code>not</code>	단항연산	논리 연산	
11	<code>and</code>	이항연산	논리 연산	왼쪽 → 오른쪽
12	<code>or</code>	이항연산	논리 연산	왼쪽 → 오른쪽
13	<code>if ~ else</code>	삼항연산	조건 표현식 연산	왼쪽 → 오른쪽

우선순위	연산의 요소	연산의 종류	연산의 분류	연산의 방향
14	<code>lambda</code>	익명함수 표현식	연산이 아님	왼쪽 → 오른쪽
15	<code>:=</code>	이항연산	할당 표현식 연산	오른쪽 → 왼쪽
마지막	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>//=</code> , <code>%=</code> , <code>**=</code> , <code>&=</code> , <code> =</code> , <code>^=</code> , <code><<=</code> , <code>>>=</code>	이항연산	할당 연산	오른쪽 → 왼쪽

- 연산의 방향을 설명하는 예시입니다.
 - 거듭제곱 연산의 방향은 오른쪽에서 왼쪽입니다.
 - 그러므로 두 개의 거듭제곱 연산이 나란히 있으면 오른쪽을 먼저 계산한 후, 왼쪽을 계산합니다. (일반적인 산술 연산과는 다릅니다.)

```
print(2 ** 3 ** 2)      # 3 ** 2를 먼저 계산(9), 그 다음 2 ** 9을 계산 :
                        결과 512
print((2 ** 3) ** 2)    # 2 ** 3을 먼저 계산(8), 그 다음 8 ** 2를 계산 :
                        결과 64
```

3. 파이썬 제어 컨트롤

3.1 아무것도 하지 않음

자리 채우기 (`pass`)

- 정해진 들여쓰기의 위치에 `pass` 가 오면 "아무것도 하지 않음"을 실행합니다.
- 전후 코드 행의 실행에 아무런 영향을 미치지 않습니다.
 - `pass` 다음 행도 실행됩니다.
- 미리 프로그램의 구조를 잡아 두는 용도로 사용합니다.
- 주석과 함께 사용하는게 바람직합니다.
 - `# 아직 구현하지 않았음`
 - `# 다음 버전에 구현 예정`

3.2 조건 제어

조건에 따라 분기 (`if ~ elif ~ else` 분기)

- `if` 키워드
 - `if` 키워드 다음에는 조건 표현식이 오고, 이어 콜론 `:` 으로 마무리 됩니다.
 - 콜론 `:` 으로 마무리된 다음 행부터 하위블록(추가 들여쓰기 블록)이 시작되어야 합니다.
 - 이 하위블록은 `if` 다음의 표현식이 참 `True` 로 평가되면 실행됩니다. 이후 이어지는 같은 수준의 `elif`, `else` 및 여기에 속한 하위블록은 모두 실행되지 않습니다.

- `if` 다음의 표현식이 거짓 `False` 로 평가되면, `if` 와 같은 수준의 블록에서 이어지는 `elif`, `else` 가 순서대로 단 하나만 실행되거나, 아무 일도 하지 않습니다.
- `elif` 키워드
 - 항상 `if` 키워드에 이어지는 하위블록이 종료된 다음 위치합니다.
 - `elif` 키워드 다음에는 표현식이 오고, 이어 콜론 `:` 으로 마무리 됩니다.
 - 콜론 `:` 으로 마무리된 다음 행부터 하위블록(추가 들여쓰기 블록)이 시작되어야 합니다.
 - 이 하위블록은 `elif` 다음의 표현식이 참 `True` 로 평가되면 실행됩니다. 이후 이어지는 같은 수준의 `elif`, `else` 및 여기에 속한 하위블록은 모두 실행되지 않습니다.
 - `elif` 다음의 표현식이 거짓 `False` 로 평가되면, `if`, `elif` 와 같은 수준의 블록에서 이어지는 `elif`, `else` 가 순서대로 단 하나만 실행되거나, 아무 일도 하지 않습니다.
 - `elif` 키워드는 같은 수준에서 `if` 이후에 여러 번 사용할 수 있습니다.
- `else` 키워드
 - 항상 `if` 키워드, `elif` 키워드에 이어지는 하위블록이 종료된 다음 위치합니다.
 - 이전의 `if` 키워드와 `elif` 키워드의 표현식이 모두 `False`인 경우 무조건 실행됩니다.
 - 하나의 `if` 조건 분기에서 2개의 `else` 키워드 및 하위블록을 사용할 수 없습니다.
 - `else` 키워드는 `if` 조건 분기의 제일 마지막에만 사용할 수 있습니다.
 - `else` 이후 같은 수준의 `elif` 키워드는 사용할 수 없습니다.
- `if ~ elif` 로 이어지는 하나의 `if` 분기에 속한 여러 개의 하위블록은 동시에 둘 이상이 실행될 수 없습니다.
- `if ~ elif ~ else` 로 이어지는 하나의 `if` 분기에 속한 여러개의 하위블록은 무조건 그 중 하나만 실행됩니다.
 - 단 수준이 다른 `if` 분기 또는 서로 다른 `if` 분기에 걸쳐서 이 원칙은 적용되지 않습니다.
- `if`, `elif`, `else` 로 만들어지는 하위 블록 내부에는 또 다른 `if ~ elif ~ else` 분기가 포함될 수 있으며, 이 분기로 인한 더 하위 수준의 하위블록이 만들어질 수 있습니다.

```
# case 1
if True:           # 조건 확인
    pass           # 실행
pass              # 실행

# case 2
if False:          # 조건 확인
    pass           # 실행되지 않음
pass              # 실행

# case 3
if True:           # 조건 확인
    pass           # 실행
elif True:         # 무시
```

```

    pass                # 실행되지 않음
pass                    # 실행

# case 4
if False:               # 조건 확인
    pass                # 실행되지 않음
elif True:              # 조건 확인
    pass                # 실행

# case 5
if False:               # 조건 확인
    pass                # 실행되지 않음
elif False:             # 조건 확인
    pass                # 실행되지 않음

# case 6
if False:               # 조건 확인
    pass                # 실행되지 않음
elif False:             # 조건 확인
    pass                # 실행되지 않음
elif True:              # 조건 확인
    pass                # 실행

# case 7
if False:               # 조건 확인
    pass                # 실행되지 않음
elif True:              # 조건 확인
    pass                # 실행
elif True:              # 무시
    pass                # 실행되지 않음

# case 8
if True:                # 조건 확인
    pass                # 실행
else:                   # 무시
    pass                # 실행되지 않음

# case 9
if False:               # 조건 확인
    pass                # 실행되지 않음
else:                   # 통과
    pass                # 실행

# case 10
if False:               # 조건 확인
    pass                # 실행되지 않음

```

```

elif False:      # 조건 확인
    pass          # 실행되지 않음
else:            # 통과
    pass          # 실행

# case 11
if False:        # 조건 확인
    pass          # 실행되지 않음
elif True:       # 조건 확인
    pass          # 실행
elif True:       # 무시
    pass          # 실행되지 않음
else:            # 무시
    pass          # 실행되지 않음

# case 12
if True:         # 조건 확인
    if False:    # 조건 확인
        pass     # 실행되지 않음
    pass         # 실행
    else:        # 오류 발생!
        pass     # 오류로 인해서 실행되지 않음
else:            # 오류로 인해서 실행되지 않음
    pass         # 오류로 인해서 실행되지 않음

# case 13
if True:         # 조건 확인
    if False:    # 조건 확인
        pass     # 실행되지 않음
    else:        # 무시
        pass     # 실행되지 않음

# case 14
if True:         # 조건 확인
    if True:     # 조건 확인
        pass     # 실행
    else:        # 무시
        pass     # 실행되지 않음
    if False:    # 조건 확인
        pass     # 실행되지 않음
    elif True:   # 조건 확인
        pass     # 실행
    else:        # 무시
        pass     # 실행되지 않음

```

```
else:                # 무시
    pass             # 실행되지 않음
```

조건에 따라 선택 (`match` 분기)



python 3.10 이후 버전에서만 사용 가능합니다!

- `match` 키워드에 지정한 변수가 `case` 키워드에 지정한 값과 일치하는 경우 일치하는 `case` 키워드의 하위 블록을 실행합니다.
 - `case` 키워드에 지정가능한 대상은 값 뿐 아니라 패턴의 형태도 가능하지만, 일단 값만 기억해 둡시다.
 - 두 개 이상의 `case` 키워드에 지정한 값이 일치하는 경우, 그 중 첫 번째 `case`의 하위블록만 실행됩니다. (하위블록은 `if ~ elif ~ else` 와 마찬가지로 배타적으로 실행됩니다.)
 - `case` 키워드에 값 대신 `_` 를 지정하면 어떠한 `case` 의 지정값에도 해당되지 않는 경우 실행도 합니다.

```
match variable:
    case v1:
        pass                # variable의 값이 v1인 경우 실행
    case v2:
        pass                # variable의 값이 v2인 경우 실행
    case v2:
        pass                # variable의 값이 v2라도 실행되지 않음
    case _:
        pass                # variable의 값이 v1, v2 둘 다 아닌 경우 실행
```

- `case` 키워드에 여러 개의 값을 지정할 수 있습니다.
 - 이 경우 값과 값은 `|` 로 구분합니다.
 - 이 값 중 하나라도 일치하면 이 `case` 의 하위블록이 실행됩니다.

3.3 반복 제어 (루프 생성)

조건에 따라 반복 루프 생성 (`while` 루프)

- `while` 키워드
 - `while` 키워드 다음에는 조건 표현식이 오고, 이어 콜론 `:` 으로 마무리 됩니다.
 - 콜론 `:` 으로 마무리된 다음 행부터 하위블록(추가 들여쓰기 블록)이 시작되어야 합니다.
 - 이 하위블록은 `while` 다음의 표현식이 참 `True` 로 평가되면 실행됩니다.
 - 하위블록에서는 조건표현식에 사용한 변수의 값을 바꿀 수 있습니다.
 - 하위블록이 모두 실행되고 나면 `while` 키워드가 있는 코드 행으로 돌아갑니다.
 - `while` 키워드의 코드 행에서는 다시 표현식을 평가하고 참 `True` 이면 다시 하위블록을 실행합니다.

- 조건 표현식이 거짓 `False` 로 평가되면 하위블록을 실행하지 않고, 그 다음 코드행으로 실행을 이어 나갑니다.
- `continue` 와 `break` 키워드
 - `while` 키워드로 만들어진 하위블록에서 `continue` 키워드를 만나면 즉시 하위블록의 실행을 중단하고 다시 `while` 키워드의 코드 행으로 돌아가 표현식을 평가합니다. (이번 반복만 종료)
 - `while` 키워드로 만들어진 하위블록에서 `break` 키워드를 만나면 즉시 반복을 멈추고 하위블록 다음 코드행으로 실행을 이어 나갑니다. (반복을 완전 종료)

정해진 만큼 반복 루프 생성 (`for ~ in` 루프)

- 내장함수 `range()` - 조건에 따른 수열을 만드는 함수입니다.
 - 인자가 정수값 하나 (n)일 때 - 0, 1, ..., n - 1의 수열을 만듭니다.
 - 인자가 두 개의 정수값 (k, n) 일 때 - k, k + 1, ..., n - 1의 수열을 만듭니다.
 - 인자가 세 개의 정수값 (k, n, s)일 때 - k에서 시작해서 n - 1까지의 수열을 만들되 각 항의 차이가 s인 등차 수열을 만듭니다.
 - 예1) `range(1, 11, 2)` 는 1, 3, 5, 7, 9 수열을 만듭니다.
 - 예2) `range(6, -3, -3)` 은 6, 3, 0 수열을 만듭니다.
 - 내장함수 `range()` 의 인자는 반드시 정수 자료형이어야 합니다.
- `for ~ in` 키워드
 - 기본형


```
for loop_variable in arr:
    pass
```

 - `arr` (리스트)에 포함된 요소를 순서대로 하나씩 `loop_variable` (루프 변수)에 할당하면서 리스트의 길이 만큼 반복하는 반복 루프를 생성합니다.
 - `range()` 결합형


```
for loop_variable in range(...):
    pass
```

 - `range()` 함수로 만들어진 수열의 값을 순서대로 하나씩 `loop_variable` (루프 변수)에 할당하면서 리스트의 길이 만큼 반복하는 반복 루프를 생성합니다.
- `continue` 와 `break` 키워드 (`while` 반복 구문과 동일한 방식으로 동작)
 - `for` 키워드로 만들어진 하위블록에서 `continue` 키워드를 만나면 즉시 하위블록의 실행을 중단하고 리스트의 다음 요소를 루프 변수에 할당하고 하위블록의 처음부터 실행합니다.
 - `for` 키워드로 만들어진 하위블록에서 `break` 키워드를 만나면 즉시 반복을 멈추고 하위블록 다음 코드행으로 실행을 이어 나갑니다. (반복을 완전 종료)

for와 while로 루프를 생성할 때 알아둬야 하거나 알아두면 좋은 것

- 중첩된 루프

- `while` 키워드로 생성된 루프 하위블록 내에 또다른 `while` 키워드와 이에 딸린 하위블록이 위치할 수 있습니다.
 - `for` 키워드 루프도 마찬가지이며, `while` 과 `for` 를 섞어서도 중첩된 루프를 만들 수 있습니다.
- 중첩된 루프 내에서 `break` 와 `continue` 키워드는 가장 가까운 루프의 `while` 또는 `for` 키워드를 대상으로 동작합니다.

- 무한루프

- 잘못된 조건을 사용하는 `while` 키워드, 잘못된 형태로 사용하는 `for` 키워드는 종종 종료되지 않고 영원히 반복하는 상황을 만듭니다. 이를 무한루프라고 합니다.

- case 1 : 조건 표현식이 항상 참이 되는 경우

```
while True:
    pass
```

- case 2 : 조건을 검사하는데 사용하는 변수의 값을 바꾸는 것을 잊은 경우

```
def countdown(i):
    while i >= 0:
        print(i)
        # i -= 1 를 실수로 누락한 경우
```

- case 3 : 조건을 잘못 설정한 경우

```
def countdown(i)
    while True:
        print(i)
        if i == 0:
            break
        i -= 1

countdown(-1)
```

- case 4 : `for` 루프를 만드는 리스트를 잘못 조작한 경우

```
def accumulated_sum(arr):
    sum = 0
    for value in arr:
        sum += value:
        arr.append(sum)

arr = [1, 2, 3, 4, 5]
accumulated_sum(arr)
print(arr)                                # 누적 합으로 값을 바꾼 [1, 3, 6, 10, 15]
```


- 필요에 따라 만들어 사용할 수 있지만, 의도치 않은 무한루프는 절대로 만들어서는 안됩니다.
- 프로그램이 무한루프 등으로 종료되지 않을 때 프로그램을 종료시키려면 CTRL키 + C키를 누릅니다.
- `for` 와 `while` 의 선택
 - 반복하는 상황이 루프 시작 전 정해져 있다면 `for`
 - 반복하는 상황이 실행의 결과로 정해지는 경우 `while`
 - 단 보통 그러하다는 것이며 반드시 그렇지는 않음
- 리스트의 요소로 루프를 만들 때 현재 사용하는 요소의 인덱스 번호가 필요하다면?
 - 방법 1

```
students = ['김', '박', '이', '최', '한']
idx = 0
for name in students:
    print(idx, "번 학생의 이름은", name, "입니다.")
    idx += 1
```

- 방법 2

```
students = ['김', '박', '이', '최', '한']
for idx in range(len(students)):
    print(idx, "번 학생의 이름은", students[idx], "입니다.")
```

3.4 예외 처리

예외

- 예외란?
 - 파이썬 프로그램 실행 중에 발생하는 오류의 한 종류로
 - 코드가 정상적으로 실행되지 못하는 경우 발생
 - 예외에 속하지 않는 오류도 발생할 수 있습니다. (잘못된 알고리즘 등)
 - 프로그램에 둘 이상의 예외가 발생할 가능성이 있더라도 동시에 둘 이상의 예외가 발생하지 않고 파이썬 인터프리터가 실행되는 순서 기준으로 먼저 발생하는 예외가 먼저 발생합니다.
 - 단, 이번 강의에서 다루지 않는 멀티 프세스, 멀티 쓰레드 등 동시성 프로그래밍 환경에서는 둘 이상의 예외가 하나의 프로그램에서 동시에 발생할 수는 있습니다.
- 파이썬 예외의 특징
 - 예외도 하나의 객체입니다.
 - 모든 예외는 예외에 대응되는 클래스로부터 만들어집니다.
 - 모든 예외 클래스는 `Exception`이라는 이름의 최상위 예외 클래스의 서브 클래스입니다.
 - 오류에 대한 정보가 포함되어 있습니다.

- 트레이스백 메시지
 - 일반적인 경우, “처리되지 않은 오류”가 발생했을 때, 파이썬은 어떤 메시지를 출력하고 프로그램을 종료 시킵니다.
 - 이때 출력되는 메시지가 트레이스백 메시지입니다.
 - 트레이스백 메시지에는 다음 정보가 포함되어 있습니다.
 - 예외가 발생한 파일
 - 예외가 발생한 위치
 - 예외가 속한 클래스
 - 예외에 대한 설명
- 대표적으로 만날 수 있는 예외의 예외 객체는 다음과 같은 클래스로부터 만들어집니다.
 - `SyntaxError` - 문법 오류로 발생하는 예외
 - `NameError` - 정의되지 않은 변수나 함수 등을 사용할 때 발생하는 예외
 - `TypeError` : 자료형과 관련된 예외
 - `ValueError` : 값이 잘못되어 발생하는 예외
 - `IndexError` : 리스트, 문자열 등에서 잘못된 값을 인덱스로 사용할 때 발생하는 예외
 - `ZeroDivisionError` : 나눗셈, 내림나눗셈, 나머지 연산에서 0으로 나눌 때 발생하는 예외
 - `FileNotFoundError`, `PermissionError` : 내장함수 `open()` 등으로 파일을 열 때 파일을 사용할 수 없거나 생성할 수 없으면 발생하는 예외
 - `FileNotFoundError` 는 지정한 디렉토리나 파일이 없을 때 발생합니다.
 - `PermissionError` 는 지정한 디렉토리 또는 파일에 적절한 사용 권한이 없을 때 발생합니다.

예외의 처리

- 예외의 전파
 - 예외가 발생하면, 예외가 발생한 위치의 함수를 호출한 상위 함수로 예외가 거슬러 전파됩니다.
 - 더 이상 거슬러 올라갈 수 없는 최상위 위치까지 예외가 전파되면 그때 프로그램이 종료되고 트레이스백 메시지가 출력됩니다.
- 예외 처리 : `try ~ except ~ finally` 분기 사용
 - `try:`
 - `try:` 로 만들어지는 하위블록은 실행될 때 예외 감시의 대상이 됩니다.
 - 이 하위블록에서 예외가 발생하면 프로그램은 즉시 중단되고 예외 처리 분기로 넘어갑니다.
 - `except 예외_클래스_이름:`
 - `try:` 의 하위 블록에서 발생한 예외가 지정한 예외 클래스에 속하는 경우, 해당하는 `except` 에 속한 하위블록이 실행됩니다.
 - 발생한 예외의 부모에 해당되는 상위 예외 클래스는 자식에 해당되는 하위 예외 클래스의 예외를 잡아서 처리할 수 있습니다.

- 여러 개의 `except` 구문을 하나의 `try:` 하위블록 이후에 배치해 감시 코드 블록에서 발생하는 여러 종류의 예외를 처리할 수 있습니다.
 - 단 `except` 하위 블록은 각각 배타적으로 실행되며, `except` 에서 찾고자 하는 예외를 기준으로 가장 먼저 등장하는 예외 처리 블록에서 예외를 처리합니다.
- 적절한 예외 클래스를 지정해 예외를 처리하게 되면 프로그램은 종료되지 않고 계속 실행됩니다.
- 적절한 예외 클래스를 지정하지 못해서 발생한 예외를 처리하지 못한 경우 예외는 계속 상위 호출 함수로 전파되며, 결국 프로그램이 종료됩니다.
- `except 예외_클래스_이름 as 예외_객체_변수:`
 - `except` 의 하위 블록에서 예외 객체를 사용할 필요가 있으면 `as` 키워드와 함께 변수명을 지정하면, 이 이름의 변수에 예외 객체가 할당됩니다.
 - 이 변수를 사용해 발생한 예외와 관련된 여러 정보를 활용할 수 있습니다.
 - 이외의 사항은 `as` 키워드를 사용하지 않는 `except` 구문과 동일하게 동작합니다.
- `finally:`
 - 예외를 처리하건 처리하지 못했건 간에 무조건 예외 처리의 마지막 단계에서 `finally:` 이하의 하위 블록이 실행됩니다.
 - `finally:` 하위 블록의 유무는 프로그램의 종료 여부와 무관합니다. 즉 적절한 예외 클래스를 `except` 키워드가 지정하지 못해 예외가 처리되지 못하면 `finally:` 하위 블록을 실행한 후 프로그램이 종료됩니다.

```
# case 0 - 예외가 발생하지 않은 경우
try:
    10 / 1                # 예외가 발생하지 않음
except ZeroDivisionError: # 처리할 예외가 없음
    pass                  # 실행되지 않음
pass                      # 실행 됨

# case 1 - 예외가 정상 처리되는 경우
try:
    10 / 0                # 0으로 나누기 때문에 ZeroDivisionError 예외
    발생
    pass                  # try 블록에 속한 이전 코드행에서 예외가 발생했으
    므로 실행되지 않음
except ZeroDivisionError: # 적절한 예외 클래스 지정으로 예외 처리
    pass                  # 실행됨
pass                      # 예외가 처리 되었으므로 계속 실행됨

# case 2 - 발생한 예외가 처리되지 않는 경우
try:
    10 / 0                # ZeroDivisionError 예외 발생
except NameError:         # NameError를 지정해서는 ZeroDivisionError
    를 처리하지 못함
    pass                  # 실행되지 않음
```

[illegible]

```

# case 7 - 예외 객체를 사용하는 경우
try:
    10 / 0
except ZeroDivisionError as e:
    print(f"예외가 발생했습니다. 예외의 정보는 다음과 같습니다. : {e}")

# case 8 - finally 구문을 사용하고 예외가 발생하지 않은 경우
try:
    10 / 1
except ZeroDivisionError:
    pass
finally:
    pass
pass

# case 9 - finally 구문을 사용하고 발생한 예외를 처리한 경우
try:
    10 / 0
except ZeroDivisionError:
    pass
finally:
    pass
pass

# case 8 - finally 구문을 사용하고 예외가 발생하지 않은 경우
try:
    10 / 0
except NameError:
    pass
finally:
    pass
pass

```

예외가 발생하지 않음

잡을 예외가 없음

실행되지 않음

예외 발생 여부와 무관하게 실행

실행됨

예외가 발생하지 않았으므로 실행됨

예외가 발생하지 않음

예외가 발생했고, 예외를 잡음

실행됨

예외 발생 여부와 무관하게 실행

실행됨

발생한 예외가 처리되었으므로 실행됨

ZeroDivisionError 예외가 발생

NameError로 ZeroDivisionError를 처리할 수 없음

실행되지 않음

예외 발생 여부와 무관하게 실행

실행됨

발생한 예외가 처리되지 않았으므로 프로그램이 종료되고, 실행되지 않음

4. 함수

4.1 파이썬 함수 소개

- 함수의 정의
 - 특정 작업을 수행하기 위해 작성된 코드 블록
 - 수학의 함수와 비슷한 개념

- 아무것도 내놓지 않을 수도 있습니다.
- 함수를 사용하는 이유
 - 반복적인 작업을 전담해서 수행합니다. (코드 재사용)
 - 적절한 이름으로 복잡한 과정을 가릴 수 있습니다. (코드 가독성 향상)
 - 함수를 사용해 부품으로 기계를 조립하듯 프로그램을 만들 수 있습니다. (논리적 구조화)
 - 문제 점검, 수정 시에 기능 단위로 작성된 함수를 찾기 쉽고 수정도 쉬워집니다. (관리 편의성)

4.2 함수의 사용법

- 함수는 항상 먼저 정의한 후 사용합니다.
- 정의하지 않아도 되는 함수도 있습니다. 내장함수가 여기에 속합니다.

함수를 정의하는 방법 (간단 요약)

- `def 함수이름(매개변수목록):` 형태로 정의합니다. (함수의 서명)
- 이어서 이 함수에 속한 코드를 하위블록에 작성합니다.
 - 이 하위블록에서 정의한 변수는 함수가 종료될 때 사라집니다. 함수 종료 이후 함수 밖에서 사용할 수 없습니다.
 - 반대로 함수 밖에서 정의한 변수는 함수 안에서 사용할 수 있지만, 사용하지 맙시다.
- 함수를 종료할 때 무엇인가를 반환해야 한다면 `return` 키워드를 사용합니다.

함수를 사용하는 방법 (간단 요약)

- `함수이름(인자목록)` 형태로 사용합니다.
 - 이를 보통 '함수를 호출한다'라고 부릅니다.
- 인자의 목록은 매개변수 목록에 대응됩니다.
- 함수를 호출하면, 프로그램의 실행은 함수에 속한 하위 블록의 첫 번째 코드 행으로 이동합니다.
- 함수에 속한 블록이 모두 종료되면 함수를 호출한 그 위치를 기준으로 이어서 실행됩니다.
 - 반드시 다음 행이 아닐 수 있습니다.

인자(argument)와 매개변수(parameter)

- 인자의 매개변수의 정의
 - 인자 : 함수를 실행할 때의 실행 조건
 - 내장함수 `print()` 에서 출력할 값
 - 매개변수 : 함수가 조건을 처리하기 위해서 이를 저장하는 변수
 - `print()` 함수가 값을 출력하기 위해서 출력할 값을 저장해 두는 그릇
- 함수를 호출할 때 인자를 지정하는 방법
 - 괄호 안에 값(리터럴), 값이 저장된 변수, 리터럴과 변수 등을 사용한 표현식 등으로 인자를 지정할 수 있습니다.

- 두 개 이상의 인자를 지정할 수 있으며, 이때는 각 인자는 쉼표 , 로 구분합니다.
- 지정된 인자는 함수의 매개변수에 저장되는데, 이때 매개변수에 전달되는 데이터는 값 또는 참조입니다.
 - 불변 자료형을 전달하면 값이 전달됩니다.
 - 가변 자료형을 전달하면 참조가 전달됩니다.
- 인자 - 매개변수 매칭 원칙
 1. 인자와 매개변수는 순서대로 매칭됩니다.
 - 위치 인자(Positional argument)라고 부릅니다.
 2. 함수를 호출할 때 매개 변수에 대응되는 모든 인자의 값을 전달해야 합니다.
 3. 매개변수의 이름을 사용해서 함수를 호출할 수 있습니다.
 - 1번 원칙의 예외
 - 함수 호출을 호출할 때 괄호 안 인자목록에 **매개 변수의 이름 = 인자의 값** 형태로 지정합니다.
 - 키워드 인자(Keyword argument)라고 부릅니다.
 - 키워드 인자로 지정하는 인자는 매개변수와 순서를 맞추지 않아도 됩니다.
 4. 함수를 지정할 때 매개변수의 기본값을 지정할 수 있습니다.
 - 2번 원칙의 예외
 - 함수를 정의할 때 괄호 안 매개변수 목록에 **매개 변수의 이름 = 기본값** 형태로 지정합니다.
 - 함수를 호출할 때 기본값이 주어진 매개변수에 대응하는 인자는 사용하지 않아도 됩니다.
 - 이 경우 주어진 기본값을 기준으로 함수가 실행됩니다.
 - 호출할 때 기본값이 주어진 매개변수에 대응되는 값을 사용하면 주어진 인자의 값으로 함수가 실행됩니다.
 - 여러 개의 매개변수를 사용하는 함수에서 두 개 이상의 매개변수에 기본값을 지정할 수 있습니다.
 - 매개변수의 목록의 순서에서 기본값을 지정하는 변수는 반드시 기본값을 지정하지 않는 변수의 뒤에 위치해야 합니다.
 5. 위치 인자, 키워드 인자, 기본값 매개 변수를 섞어 쓸 수 있습니다.
 - 단, 함수를 호출할 때 인자 목록에서 키워드 인자를 사용한 후 위치 인자를 배열할 수 없습니다.
 6. 여러 개의 인자를 모아 하나의 변수에 저장할 수 있습니다.
 - 가변 인자라고 하며 인자의 수, 종류가 고정되어 있지 않을 때 사용할 수 있습니다.
 - 내장함수 `print()` 의 예
 - 위치 인자에 대응되는 가변 인자는 튜플에 저장됩니다.
 - 키워드 인자에 대응되는 가변 인자는 딕셔너리에 저장됩니다.

함수의 반환값

- 함수는 결과를 함수 밖으로 전달할 수 있습니다.

- 이를 '반환한다'라고 표현합니다.
- 결과를 반환하지 않는 함수도 만들 수 있습니다.
- 두 개 이상의 값을 반환할 수도 있습니다.
- `return` 키워드
 - 함수를 종료할 때 사용합니다.
 - `return` 키워드 뒤에 코드가 남아 있건 남아 있지 않건 상관없이 종료됩니다. 다만 제어 컨트롤에 따라 `return` 키워드가 포함된 코드 행이 실행되지 않는 경우는 당연히 함수가 종료되지 않습니다.
 - 키워드 뒤에 이어 기입한 값을 반환합니다.
 - 이 값은 리터럴, 변수, 표현식 모두 가능하지만, 실제로 반환되는 대상은 값 또는 참조입니다.
 - 불변 자료형을 반환하면 값이 전달됩니다.
 - 가변 자료형을 반환하면 참조가 전달됩니다.
 - 값을 지정하지 않으면 아무것도 반환하지 않고 함수가 종료됩니다.
 - `len()` 함수는 하나의 값을 반환하는 함수입니다.
 - `print()` 함수는 아무런 값을 반환하지 않는 함수입니다.

함수에 대해서 알아둬야 할 몇 가지 사실

- 함수가 다른 함수를 호출할 수 있습니다.
- 함수가 자기 자신을 호출할 수도 있습니다.
- 함수는 반복해서 호출되면 메모리에 함수의 탑이 쌓이게 됩니다.
 - 프로그램이 시작될 때, 이 탑의 높이는 1층입니다.
 - 나중에 호출되는 함수는 위층에 쌓입니다.
 - 함수가 종료되면 탑의 가장 위층을 제거합니다.
 - 고층 건물을 규제하듯 파이썬도 시스템 보호를 위해서 지나치게 탑이 높게 쌓지 못하도록 규제합니다.
- 메서드도 일종의 함수입니다. 다만 메서드는 주어가 객체로 정해진 함수입니다.

4.3 내장함수

형변환 내장함수

- `bool()` : 인자의 값을 평가 기준 참거짓값으로 변환합니다.
- `float()` : 인자의 값을 실수 자료형으로 변환합니다.
 - 변환할 수 없으면 예외가 발생합니다. (형변환 함수 공통)
- `int()` : 인자의 값을 정수 자료형으로 변환합니다.
- `list()`, `tuple()` : 인자의 값을 리스트 자료형이나 튜플 자료형으로 변환합니다.
 - 리스트 자료형이나 튜플 자료형으로 변환할 수 있는 자료형은 모두 이터러블 `iterable`이라는 속성이 있습니다.

- 문자열, 리스트, 튜플, 딕셔너리, 집합, `range` 객체 등이 여기에 해당됩니다.
- `set()` : 이터러블 속성 자료형 인자를 세트 자료형으로 변환합니다.
 - 이때 중복된 항목은 하나만 포함합니다.
- `dict()` : 특정한 형태의 이터러블 속성 자료형 인자를 딕셔너리 자료형으로 변환합니다.

값을 찾아주는 내장함수

- `len()` : 객체의 크기를 알려주는 내장함수
- `max()`, `min()` : 여러 인자 또는 컬렉션에 포함된 항목 중에서 최대값 또는 최소값을 골라주는 내장함수
- `sum()` : 컬렉션에 포함된 값의 합을 반환하는 내장함수
 - 여러 인자를 더하는 형태로 사용 불가

리스트 정렬 내장함수

- `sorted()` : 인자로 전달한 리스트를 오름차순으로 정렬한 리스트를 반환합니다.
 - 리스트 자체를 변경하지 않음
- `sorted(arr, reverse = True)` : 리스트 `arr` 을 내림차순으로 정렬한 리스트를 반환합니다.

내장함수 `print` 와 `input`

- `print()`
 - 인자를 화면에 출력하는 내장함수
 - 문자열, 숫자형 뿐 아니라 파이썬의 '모든' 자료형을 출력할 수 있습니다.
 - 반환값은 없습니다.
 - 여러 개의 위치 인자를 사용하면 이를 모두 출력합니다.
 - 키워드 인자 `end` : 기본값은 `'\n'` (줄바꿈)이며 모든 인자를 출력한 후 덧붙여 출력합니다.
 - 키워드 인자 `sep` : 기본값은 `' '` (한칸 공백)이며 여러 인자를 출력할 때 인자 사이에 출력합니다.
- `input()`
 - 사용자로부터 키보드 입력을 받는 내장함수
 - 프로그램이 잠시 멈추고 사용자의 입력을 기다립니다.
 - 사용자가 값을 입력하고 엔터키를 누르면, 엔터키를 제외한 입력한 내용을 문자열 자료형으로 반환합니다.
 - 인자를 사용하면 화면에 인자를 출력하고 사용자의 입력을 기다립니다.

5. 클래스와 객체

5.1 객체지향 설계

- 객체 (파이썬의 객체와는 다른 일반적인 객체)
 - 현실 세계의 사물, 개념을 프로그래밍에서 표현한 것입니다.

- 속성과 기능을 가집니다.
- 클래스
 - 객체의 청사진입니다.
 - 클래스를 기반으로 다수의 객체를 생성할 수 있습니다.
- 객체와 클래스의 예시
 - Human 클래스를 기반으로 다음 3개의 객체를 생성할 수 있습니다.
 - 김철수 객체
 - 속성 (성별 - 남, 나이 - 40, 취미 - 모름)
 - 기능 (잔다, 일어난다, 먹는다.)
 - 이영희 객체
 - 속성 (성별 - 여, 나이 - 45, 취미 - 수영)
 - 기능 (잔다, 일어난다, 먹는다.)
 - 박동수 객체
 - 속성 (성별 - 남, 나이 - 11, 취미 - 게임)
 - 기능 (잔다, 일어난다, 먹는다.)
- 객체지향 설계의 특징
 - 추상화
 - 복잡한 시스템의 필요한 부분만 추출해 표현합니다.
 - '잔다'는 기능에 대해서 잠의 생리적 메커니즘의 구체적인 부분은 몰라도 설계도인 Human 클래스에서 이를 구현해 두었다면 '잔다'는 추상화된 행위만으로 사용할 수 있습니다.
 - 캡슐화
 - 기능이나 속성 중 노출할 필요가 없는 것을 캡슐화해서 감출 수 있습니다.
 - '속성에 저장된 비밀번호를 입력한다'는 기능을 사용하면 비밀번호를 일일이 노출하지 않고도 인증이 가능합니다.
 - 상속
 - 상위 클래스를 재사용해서 새로운 하위 클래스를 만들 수 있습니다.
 - Human 클래스를 사용해 하위 클래스 Student를 만들면 Human 클래스에 정의된 잔다, 일어난다, 먹는다 등의 기능을 다시 구현할 필요가 없고 '공부한다' 등 새로 추가하는 클래스에 필요한 기능만 추가하면 됩니다.
 - 다형성
 - 상속 과정에서 실제로는 다르지만 같은 이름으로 부를 수 있는 기능을 다양하게 구성할 수 있습니다.
 - Human의 하위 클래스인 Driver와 Student 클래스를 만들 때 '일한다'라는 기능을 클래스 별로 운전하는 과정과 공부하는 과정으로 구성하면 '일한다'는 같은 명칭의 기능을 객체를 생성하는데 사용한 클래스별로 다른 형태로 구성할 수 있습니다.

- 객체지향 설계의 장점
 - 재사용성 - 상위클래스를 재사용하여 빠르고 효율적으로 하위클래스를 만들 수 있고, 각각의 객체마다 객체에 대한 함수를 재정의하지 않고도 설계도 기준으로 동작하게끔 만들 수 있습니다.
 - 확장성 - 다중 상속의 개념을 도입해 새로운 기능을 쉽게 추가할 수 있습니다.
 - 관리 및 유지가 쉬움 - 코드가 잘 구조화되고, 논리적으로 프로그램의 각 부분이 분리되어 프로그램에 문제가 발생하거나 기능을 업그레이드하기 용이합니다.

5.2 파이썬 클래스

파이썬 클래스 만들기

- 파이썬 클래스의 기본 형태
 - 키워드 `class` 와 클래스의 이름, 그리고 콜론:을 사용해 클래스를 정의함을 선언합니다.
 - 콜론 다음 줄 부터 하위블록이 만들어지며, 이 하위블록에서 클래스의 여러 메서드를 정의합니다.
 - 클래스의 객체를 대상으로 동작하는 함수를 메서드라고 부릅니다.
 - 하위블록이 끝날 때 까지가 클래스의 정의입니다.

```
class Human:          # Human 라는 이름의 클래스
    def sleep(self):
        pass

    def eat(self):
        pass

    def get_up(self):
        pass
# 하위블록 종료 - 여기까지가 Human 클래스의 정의

class Student:        # Student 라는 이름의 클래스
    pass
```

- 클래스 이름의 일반적인 모범 명명법
 - 함수나 변수의 이름짓기 규칙과 동일하지만 snake_case 대신 CamelCase를 사용합니다.
 - 대문자로 시작합니다.
 - 두 단어 이상을 사용해 클래스 이름을 만드는 경우, 각 단어의 첫 글자를 대문자로, 나머지 글자는 소문자로 하고 이를 공백없이 이어 씁니다.
 - ex) Destroyer, NavyArmy, SuperPowerShip

파이썬 클래스의 객체를 만들고 사용하기

- 마치 함수를 사용하듯 클래스를 호출하는 형태로 사용하면 객체가 만들어지고 반환됩니다.

```
Human()          # Human 클래스의 객체 생성
Student()        # Student 클래스의 객체 생성
```

- 이렇게 객체를 생성한 후, 계속 사용하려면 변수에 할당해야 합니다.

```
first_human = Human()      # 생성된 객체를 first_human 변수에 할당
```

- 이 때 생성되는 객체는 가변형 객체이며, 즉 이 변수는 객체를 참조하게 됩니다.
- 변수에 할당된 객체를 재할당하면 '복사'가 아닌 '참조 추가'가 발생합니다.

파이썬 클래스의 메서드와 속성 정의하기

- `self` - 인스턴스 객체 변수
 - 클래스 메서드의 첫 번째 매개변수는 일부 특수한 경우를 제외하고 대상이 되는 객체의 참조를 의미합니다.
 - 이를 인스턴스 객체라고 부르며, 이 매개변수는 관례상 `self` 라는 이름을 사용합니다.
- 파이썬 클래스의 메서드를 정의할 때, 클래스의 속성이나 메서드를 사용하려면 반드시 인스턴스 객체 매개 변수(`self`)에 점 표기법을 붙여서 사용합니다.
 - 점 표기법으로 표현하지 않으면 일반 변수나 일반 함수를 찾게 됩니다.
 - 클래스 메서드에서 다른 클래스 메서드를 호출할 때도 인스턴스 객체 인자(`self`)는 점 표기법으로 지정하지 인자에 지정하지 않습니다.

```
class Human():
    def get_up(self, wake_up_time):
        # wake_up_time과 self.wake_up_time은 구분됩니다.
        # wake_up_time - get_up 메서드의 매개변수
        # self.wake_up_time - Human 클래스의 속성
        self.wake_up_time = wake_up_time

    def sleep(self, sleeping_time):
        self.sleeping_time = sleeping_time

    def get_up_and_sleep(self, wake_up_time, sleeping_time):
        # self.get_up()은 클래스 내에 정의한 get_up 메서드입니다.
        # 만약 그냥 get_up()을 호출하면 클래스의 메서드가 아닌 일반 함수를 찾습니다
        self.get_up(wake_up_time)
        self.sleep(sleeping_time)
```

생성된 객체의 메서드를 호출하고 속성 사용하기

- 클래스의 메서드를 사용하려면, 점 표기법을 사용합니다.
 - `생성된_객체의_이름.클래스에_정의한_메서드의_이름()`

- 클래스의 메서드를 만들고 호출할 때 인자와 매개변수의 관계는 함수를 만들고 호출할 때의 인자와 매개변수의 관계와 거의 같습니다. 단, 클래스 정의에서 첫 번째 매개변수로 지정한 인스턴스 객체(일반적으로 매개변수명 `self`)는 인자로 지정하지 않습니다.

```
# Human 클래스의 객체 lazy_human 생성
lazy_human = Human()
# 이 시점에 lazy_human 객체는 wake_up_time 속성이 없습니다.
lazy_human.get_up("10시 30분")
# 이제서야 비로소 lazy_human 객체는 wake_up_time 속성을 가집니다.
# lazy_human.get_up(self, "10시 30분") 또는
# get_up(lazy_human, "10시 30분")은 모두 잘못된 메서드 호출 방식입니다.

# Human 클래스의 새로운 객체 diligent_human 생성
diligent_human = Human()
diligent_human.get_up_and_sleep("06시 00분", "23시 30분")
```

- 점 표기법에 사용한 객체(위의 예에서는 `first_human`)가 첫 번째 인자의 역할을 하며, 클래스 매개 변수 (`self`)의 값이 됩니다.
- 메서드와 유사하게 점 표기법으로 속성명을 지정해 속성의 값을 사용할 수 있습니다.
 - 객체가 속성을 가지게 되는 시점에 주의해야 합니다.

```
lazy_human = Human()
# 여기서 lazy_human.wake_up_time 속성 변수를 사용하면 예외가 발생합니다.
lazy_human.get_up("10시 30분")
print(lazy_human.wake_up_time)      # 출력 - 10시 30분

diligent_human = Human()
diligent_human.get_up_and_sleep("06시 00분", "23시 30분")
print(diligent_human.wake_up_time, sleeping_time)
# 출력 - 06시 00분 23시 30분
```

파이썬 클래스의 생성자

- 생성자
 - 클래스의 객체가 생성될 때 자동으로 호출되는 메서드
 - 생성자의 이름은 항상 `__init__`
 - 밑줄 `_`은 앞 뒤 각각 2개씩입니다.
 - 생성자는 특정한 값을 반환하지 않습니다.
 - `return` 키워드를 사용할 수는 있지만, 특정한 값을 지정해 반환할 수는 없습니다.
 - 단 `None`을 반환할 수는 있습니다. (결국 아무것도 반환하지 않는다는 의미입니다.)
- 생성자 만들고 사용하기
 - 생성자 메서드도 다른 메서드와 마찬가지로 첫 번째 매개변수는 인스턴스 객체(`self`)여야 합니다.

- 생성자 메서드도 다른 메서드와 마찬가지로 매개변수를 가질 수 있습니다.
 - 이 경우, 클래스의 객체를 생성할 때 클래스 이름 뒤 괄호 안에 생성자 매개변수에 대응되는 인자 (`self`)를 제외하고 제시해야 합니다.
- 생성자 메서드의 매개변수도 함수와 마찬가지로 기본값을 가질 수 있으며, 클래스를 생성할 때의 인자도 키워드 인자, 위치 인자 등의 방식을 모두 사용할 수 있습니다.

```
class Human():
    def __init__(self, name, age, gender = "여자", hobby = "모름"):
        self.name = name
        self.gender = gender
        self.age = age
        self.hobby = hobby

    def __str__(self):
        return f"{self.name}({self.age}, {self.gender}), 취미는 {self.hobby}"
# return f"{name}({age}, {gender}), 취미는 {hobby}"는 틀린 사용법입니다.

# first_human = Human() 형태로 생성하면 예외가 발생합니다.
first_human = Human("김철수", 40, "남자") # 기본값 인자 생략
second_humen = Human("이영희", 45, hobby = "수영") # 키워드 인자 사용
print(first_human)
print(second_humen)
```

알아둬야 할 것 : `None` 객체와 `__str__()` 메서드

- `None`
 - “아무것도 아닌 것”을 의미하는 객체
 - 아무것도 아닌 것을 의미하거나
 - 아무 값도 없음을 의미하는
 - ‘객체’입니다.
 - 조건식에 사용하면 거짓 `False`로 평가됩니다.
 - 다른 자료형의 데이터와 산술연산에 사용할 수 없습니다.
 - 클래스의 속성 또는 변수를 미리 초기화해 둘 때 또는 함수의 반환 값이 없음을 나타낼 때 등에 사용됩니다.
 - `None`의 값이라도 초기화해 두지 않으면 경우에 따라 예외가 발생할 수 있습니다.
 - `return`과 `return None`은 동일합니다.
 - 두 개 이상의 값을 반환할 때 둘 중 하나가 유효한 반환값이 아닐 때 유용
 - `None` 값을 반환하면, 반환값을 `if` 조건 구문에 사용해 함수가 정상적으로 종료되었는지를 확인하는 용도로 사용할 수 있습니다.
 - 초기화에 사용하는 예시

```

class Human:
    def __init__(self, name = None):
        self.name = name

    def print_name(self):
        if self.name == None:                # if not self.name: 과 동일함
            print("성명 불상")
        else:
            print(f"이름은 {self.name} 입니다.")

human_with_name = Human("아무개")
human_without_name = Human()
human_with_name.print_name()
human_without_name.print_name()

```

- 반환값의 유효성을 확인하는 용도의 예시

```

def sum_and_divide(operand_1, operand_2):
    sum_result = operand_1 + operand_2
    if operand_2 != 0:
        return sum_result, operand_1 / operand_2
    else:
        return sum_result, None

result_sum, result_divide = sum_and_divide(10, 0)
print(f"덧셈 결과는 {result_sum}이며, ", end = "")
if not result_divide:
    print("0으로 나눌 수 없습니다.")
else:
    print(f"나눗셈 결과는 {result_divide}입니다.")

```

- `__str__()` 메서드

- 파이썬의 모든 객체는 내장함수 `str()` 을 사용해 문자열 형태로 변형할 수 있습니다.

```

class Human:
    def __init__(self, name):
        self.name = name

int_vvariable = 10
list_variable = [1, 2, 3, 4, 5]
class_object = Human("아무개")

int_variable_to_str = str(int_vvariable)
list_variable_to_str = str(list_variable)

```

```
class_object_to_str = str(class_object)
# "<__main__.Human object at 0x7f37aeedad50>"와 유사한 형태의 문자열로 변

print(int_variable_to_str)
print(list_variable_to_str)
print(class_object_to_str)
```

- 클래스에 `__str__()` 메서드를 정의하면, 이 메서드의 반환값의 형태로 문자열이 만들어집니다.

```
class Human:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"이름은 {self.name}입니다."

class_object = Human("아무개")
class_object_to_str = str(class_object)
# "이름은 아무개입니다."의 문자열로 형변환
print(class_object_to_str)
```

- 일부러 형변환을 하지 않아도 내장함수 `print()` 는 자동으로 문자열로 형변환한 결과를 출력합니다.

```
class Human:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"이름은 {self.name}입니다."

class_object = Human("아무개")
print(class_object)
```

- 이와 유사하게 메서드의 이름에 따라 약속된 역할을 하는 몇 종류의 메서드가 더 있습니다. 이러한 메서드를 **매직 메서드**라고 부릅니다.

5.3클래스의 상속

부모 클래스와 자식 클래스

- 더 넓은 범위의 추상화를 담당하는 상위 클래스를 부모 클래스 또는 슈퍼 클래스라고 부릅니다.
- 조금 더 구체적인 하위 클래스를 자식 클래스 또는 서브 클래스라고 부릅니다.
- 부모 클래스 - 자식 클래스 관계 예시
 - 사람 - 학생
 - 동물 - 포유류

- 포유류 - 영장류
- 영장류 - 사람

상속으로 클래스 만들기

- 클래스의 상속
 - 부모 클래스를 사용해 자식 클래스를 만드는 과정을 상속이라고 합니다.
 - 자식 클래스를 선언할 때 클래스의 이름 옆에 부모 클래스의 이름을 괄호로 묶어 나타냅니다.

```
class SubClassName(SuperClassName):
    pass
```

- 자식 클래스는 다음에 해당되는 코드를 추가합니다.
 - 부모 클래스에는 없지만 자식 클래스에는 필요한 메서드와 속성
 - 부모 클래스에 있지만, 자식 클래스에는 다르게 동작하는 메서드, 자식 클래스에서는 값이 다른 속성 등
 - 이 경우 부모 클래스에서 정의한 바는 무시되고, 자식 클래스에서 정의한 바가 적용되며 이러한 구현 방식을 "오버라이딩(overriding)"이라고 부릅니다.
- 내장 함수 `super()`
 - 부모 클래스에 대한 참조가 필요할 때 사용하는 내장함수
 - 자식 클래스에서 `super()` 함수를 호출하면 부모 클래스의 참조가 반환됩니다.
 - `super()` 함수에 점 표기법을 사용해 메서드나 속성을 호출하면 부모 클래스의 메서드나 속성을 사용할 수 있습니다.
 - 부모 클래스의 생성자 메서드를 `super()` 함수를 사용해서 호출할 수 있습니다.

```
class SubClassName(SuperClassName):
    def __init__(self):
        super().__init__()
    pass
```

6. 모듈

6.1 모듈, 라이브러리, 패키지

용어 정리

- 모듈
 - 일반적인 의미
 - 파이썬 코드가 들어 있는 각각의 .py 파일을 말합니다.
 - 함수, 클래스, 변수, 기타 실행 가능한 코드로 구성됩니다.

- 흔히 사용되는 의미
 - 파이썬 프로그래밍 과정에서 어떤 기능을 직접 개발하지 않고 가져다 쓸 때 이 때의 기능의 그룹을 말합니다. (ex - 무작위 데이터를 다루는 '랜덤 모듈')
- 패키지
 - 여러 모듈(.py 파일)이 포함된 하나의 디렉토리를 말합니다.
 - 모듈의 집합에 해당됩니다.
- 라이브러리
 - (파이썬이 아닌 일반적인 프로그래밍에서 통용되는) 일반적인 의미
 - 프로그램을 만들 때 외부에서 제공한 기능을 사용하는 경우, 이 기능을 제공하는 무엇인가 (ex. 윈도우의 DLL 파일)
 - 파이썬에서의 라이브러리의 의미
 - 특정 목적을 위해 작성된 여러 모듈 및 패키지의 모음입니다.
 - 위에서 설명한 일반적인 의미로도 통용됩니다.

표준 라이브러리와 서드파티 모듈, 패키지, 라이브러리

- 파이썬 표준 라이브러리
 - 파이썬을 설치할 때 파이썬을 구동하는 인터프리터와 함께 설치되는 모듈과 패키지의 집합을 말합니다.
 - 다양한 프로그래밍 작업에 널리 사용되는 '기본 기능'과 '기본 구조'의 집합입니다.
 - 즉 파이썬의 '공식' 기본 라이브러리를 말합니다.
- 파이썬 표준 라이브러리의 특징
 - 거의 모든 기본적인 프로그래밍 기능을 포함합니다.
 - 공식 라이브러리인 만큼 안정적으로 동작하며, 개발에 참고할 수 있는 자료가 풍부합니다.
 - <https://docs.python.org/3/library/index.html>
 - 일부만 번역되어 있지만 한국어 기준 리소스도 활용 가능합니다.

```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

모든 비 키워드 인자는 `str()` 이 하듯이 문자열로 변환된 후 스트림에 쓰이는데, *sep* 로 구분되고 *end* 를 뒤에 붙입니다. *sep* 과 *end* 는 모두 문자열이어야 합니다; `None` 일 수도 있는데, 기본값을 사용한다는 뜻입니다. *objects* 가 주어지지 않으면 `print()` 는 *end* 만 씁니다.

file 인자는 `write(string)` 메서드를 가진 객체여야 합니다; 존재하지 않거나 `None` 이면, `sys.stdout` 이 사용됩니다. 인쇄된 인자는 텍스트 문자열로 변환되기 때문에, `print()` 는 바이너리 모드 파일 객체와 함께 사용할 수 없습니다. 이를 위해서는 대신 `file.write(...)` 를 사용합니다.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

버전 3.3에서 변경: *flush* 키워드 인자가 추가되었습니다.

<https://docs.python.org/ko/3/library/functions.html#print>에서 확인할 수 있는 내장함수 `print()` 의 설명

- 표준 라이브러리에 포함된 것
 - 기능을 기준으로 분류하면 다음과 같습니다.
 - 내장함수 - 별다른 조치 없이 바로 사용할 수 있는 함수들입니다.
 - 모듈
 - 표준 라이브러리에서 제공하는 기능들은 관련이 있는 기능 단위로 묶은 모듈 단위로 제공되며,
 - 예외는 있지만 대부분 `import` 키워드를 사용해 어떤 모듈을 사용하겠다고 선언을 해야 사용할 수 있습니다.
 - 모듈에는 함수 뿐 아니라, 변수, 클래스 및 클래스의 메서드 등도 포함되어 있습니다.
 - 표준 라이브러리에 포함된 주요한 모듈
 - `os` 모듈 - 운영 체제와 관련된 모듈
 - `sys` 모듈 - 파이썬 인터프리터와의 상호작용에 필요한 구조와 기능이 포함된 모듈
 - `datetime` 모듈 - 시각과 시간, 날짜 등을 다루는 구조와 기능이 포함된 모듈
 - `time` 모듈 - 조금 더 근원적인 형태의 시각을 다루는 기능이 포함된 모듈
 - `math` 모듈 - 수학 연산을 위한 기능을 제공하는 모듈
 - `random` 모듈 - 임의성을 활용하는 기능을 제공하는 모듈
 - 이외에도 많은 유용한 모듈이 포함되어 있습니다.
- 표준 라이브러리에 포함되지 않은 것 - 서드파티(3rd party) 모듈, 패키지, 라이브러리
 - 비록 '공식' 라이브러리에는 포함되어 있지는 않으나 특정 목적의 프로그래밍에 꼭 필요한 기능을 제공하는 제3자가 개발한 모듈, 패키지, 라이브러리 등이 있습니다.
 - 예) 데이터 분석에 주로 사용되는 패키지, 모듈, 라이브러리
 - `numpy` - 다양한 수학적 연산과 배열 처리를 위한 다양한 모듈의 집합
 - `pandas` - 데이터를 조작하고 분석하기 위한 고수준의 도구들이 모여 있는 라이브러리

- matplotlib - 데이터 시각화를 위한 기본 기능을 제공하는 라이브러리
- seaborn - matplotlib을 기반으로 동작하는 고수준의 시각화 라이브러리
- statsmodels - 통계 모델링과 데이터 분석에 사용되는 라이브러리
- SciPy - 수학, 과학, 공학 계산을 위한 고급 함수를 제공하는 라이브러리

6.2 표준 라이브러리에 포함된 모듈 사용해보기

random 모듈

- random 모듈 소개
 - 난수 생성과 관련된 다양한 기능을 제공하는 모듈
 - 표준 라이브러리에서 기능을 제공해 별도로 설치할 필요가 없습니다.
 - 완전한 난수가 아닌 의사 난수(Pseudo-random number)를 생성해 동작합니다.
 - Mersenne Twister라는 이름의 방법으로 계산된 수열에서 숫자를 선택합니다.
 - 2^{19937} - 1번의 선택 후에는 처음부터 반복되는 수준의 안전성을 보장합니다.
 - secrets 모듈을 사용하면 암호학적으로 안전한 난수를 생성합니다. 다만 random 모듈을 사용해 난수를 생성하는 속도가 훨씬 빠르므로 성능과 안전성 사이에서 적절히 타협해도 좋다면 random 모듈의 난수를 사용하는 것이 안전합니다.
 - 이러한 복잡한 난수 생성의 개요를 설명하는 이유는, 복잡한 구현 대신 라이브러리의 기능을 사용하는 것이 얼마나 효율적인지를 보여드리기 위함입니다.
- 다음과 같은 주요 함수를 random 모듈에서 사용할 수 있습니다.
 - `random()` 함수 - 0과 1 사이의 임의의 실수형 수를 반환합니다. (0이 나올 수 있지만 1은 나올 수 없습니다.)
 - `randint(a, b)` - `a`와 `b` 사이의 임의의 정수형 수를 반환합니다. (`a`, `b` 모두 나올 수 있습니다.)
 - `uniform(a, b)` - `a`와 `b` 사이의 임의의 실수형 수를 반환합니다. (`a`, `b` 모두 나올 수 있습니다.)
 - `choice(sequence)` - `sequence`에 포함된 임의의 요소 하나를 선택해 반환합니다.
 - `sequence` 인자에 리스트, 문자열, 튜플 등을 사용할 수 있습니다.
 - `choices(sequence, k)` - `sequence`에 포함된 임의의 요소 중 `k`개를 선택해 반환합니다.
 - 한번 선택한 값도 다시 선택될 수 있습니다.
 - `sample(sequence, k)` - `sequence`에 포함된 임의의 요소 중 `k`개를 선택해 반환합니다.
 - 한번 선택한 값은 다시 선택되지 않습니다.
 - `shuffle(sequence)` - `sequence`에 포함된 요소를 임의의 순서로 뒤섞습니다.
 - 원본 시퀀스가 변경됩니다. 그러므로 튜플, 문자열 등 불변형 객체는 인자로 사용할 수 없습니다.
 - `seed(k)` - 임의로 뒤섞는 기준을 지정합니다.
 - 즉 `k` 값이 동일하면 항상 같은 순서대로 같은 결과가 나옵니다.

random 모듈로 알아보는 모듈 사용법

- random 모듈을 사용하려면?

- random 모듈은 표준 라이브러리에 속한 모듈로 별도 설치가 필요 없습니다.
- `import random` : random 모듈을 사용한다는 선언할 수 있습니다.
 - 이후 모듈의 이름인 random에 점 표기법을 붙여 함수를 사용할 수 있습니다.

```
import random                                # 모듈 사용 선언
print(random.random())                      # 모듈 이름 random에 점 표기법으로 random
print(random.randint(10, 20))               # 같은 방법으로 randint() 함수 호출
```

- 쉼표로 구분해 두 개 이상의 모듈을 동시에 사용 선언 할 수 있습니다.

```
import random, math                        # random 모듈과 math 모듈 사용 선언
```

- `from random import randint` : random 모듈의 `randint()` 함수를 "마치 내가 구현한 함수처럼 사용하겠다"는 선언입니다.
 - 내가 구현한 함수처럼 사용하기 때문에 random에 점 표기법을 사용할 필요가 없습니다.
 - `import random` 의 의미를 포함하지 않습니다.

```
from random import randint                  # random 모듈의 randint() 함수 사용 선언
print(randint(10, 20))                     # 모듈을 지정해 함수 이름을 직접 선언했기 때문
                                           # random.

# 아래 두 경우 예외가 발생합니다.
# random.random() --> (import random은 하지 않았음)
# uniform(10, 20) --> random.uniform() 함수는 사용 선언을 하지 않았음
```

- 쉼표로 구분해 두 개 이상의 함수를 사용 선언할 수 있습니다.

```
from random import randint, uniform
print(randint(10, 20))
print(uniform(10, 20))
```

- 함수 이름 대신 `*` 를 사용해 모든 함수를 사용 선언할 수 있습니다.
 - 하지만 가급적 쓰지 맙시다. 복잡한 프로그램에서 실수할 가능성이 커집니다.

```
from random import *
randint(10, 20)
uniform(10, 20)
```

- `import` 대상을 `as` 키워드를 사용해 다른 이름으로 대신해 사용할 수 있습니다. (별칭이 아니라 사용 이름 변경)

```
# 모듈의 이름을 변경해 사용합니다.
import random as rd
```

```
print(rd.randint(10, 20))
# random 대신 rd를 사용한다고 선언했으므로 random.randint(10, 20)은 예외 발
```

```
# 함수의 이름을 변경해 사용합니다.
from random import randint as random_integer
print(random_integer(10, 20))
# randint 대신 random_integer를 사용한다고 선언했으므로
# randint(10, 20)은 예외 발생
```

```
# as로 이름을 바꾸어 쓸 때도 다음처럼 둘 이상을 동시에 사용 선언할 수 있습니다.
from random import randint as r_int, uniform as r_float
print(r_int(10, 20), r_float(10, 20))
```

- `import` 사용 선언은 반드시 프로그램 .py 파일의 제일 앞부분에 위치하지 않아도 되며, 사용하고자 하는 기능이 포함된 코드행 앞에서만 하면 됩니다.
 - 단 .py 파일의 제일 앞부분에 `import` 를 모아서 사용하는 것이 일반적입니다.
- 필요하지 않은 모듈을 사용 선언하면 시스템의 자원을 낭비하게 됩니다. 꼭 필요한 모듈만 사용 선언하도록 합니다.

6.3 표준 라이브러리에 포함되지 않은 모듈의 사용법

기본 사용법

- 사실 사용법은 차이가 없습니다.
- 차이는 이러한 라이브러리, 패키지, 모듈은 별도로 설치해야 한다는 점에 있습니다.

```
import pandas
# pandas는 외부 라이브러리이므로 설치하지 않고 import를 하면
# 예외가 발생합니다.
```

- 기본적으로 내려받아 설치하는 과정을 먼저 진행해야 하지만 설치할 대상에 따라 방법은 조금씩 다릅니다.
 - pip 명령을 사용해 설치 (대부분의 파이썬 서드파티 패키지)
 - 파이썬의 서드파티 패키지를 관리하는 서비스인 PyPI (Python Package Index)에서 관리하는 서드파티 패키지는 이 방법으로 설치할 수 있습니다.
 - pip 명령으로 설치하지만 내려받을 곳을 별도로 지정해야 하는 패키지
 - PyPI에서 관리하지 않는 서드파티 패키지 중 일부는 이 방법으로 설치해야 합니다.
 - PyPI에서 관리하는 경우에도 인터넷 접근성 문제, 최신 베타 버전 패키지 사용 등의 경우도 이러한 설치 방식을 사용해야 합니다.
 - 별도 설치 프로그램을 사용해야 하는 패키지
 - 이 방식은 해당 서드 파티 라이브러리, 운영체제마다 다른 방법으로 설치해야 합니다.
 - miniconda, anaconda의 패키지 관리자를 사용해 설치

- miniconda나 anaconda 환경에서는 이 방식으로 설치할 수 있습니다.
- 서드파티 패키지, 라이브러리, 모듈을 설치할 때는 가상환경 사용을 강력하게 권장합니다.

pip를 사용해 pandas 설치하기

- 일부 OS에서는 pip 대신 pip3 명령을 사용해야 합니다.
 - python 대신 python3 명령으로 파이썬 인터프리터를 실행하는 경우 대부분 여기에 해당됩니다.
- 이하 과정은 모두 파이썬 대화형 환경이나 파이썬 프로그래밍 환경이 아닌 터미널 상에서 실행해야 합니다.
- 설치 여부 및 설치된 버전 확인법 - `pip show 패키지이름`

```
(pip_test) C:\Temp\pip_test>pip show pandas
WARNING: Package(s) not found: pandas

(pip_test) C:\Temp\pip_test>pip show flask
Name: Flask
Version: 3.0.3
Summary: A simple framework for building complex web applications.
Home-page:
Author:
Author-email:
License:
Location: C:\Users\Edberg\AppData\Local\Packages\PythonSoftwareFoundation\Python312\python.exe
Requires: blinker, click, itsdangerous, Jinja2, Werkzeug
Required-by:
```

- pandas 패키지가 설치되어 있지 않으므로 설치합니다.
 - `pip install 패키지이름`
 - 설치 후 설치가 잘 되었는지 확인하는 것이 좋습니다.

```
(pip_test) C:\Temp\pip_test>pip install pandas
Defaulting to user installation because normal site-packages is not writeable
Collecting pandas
  Downloading pandas-2.2.2-cp312-cp312-win_amd64.whl.metadata (19 kB)
  (... 중간 생략...)
Successfully installed numpy-2.1.1 pandas-2.2.2 python-dateutil-2.9.0.post0 pytz-2024.2 six-1.16.0 tzdata-2024.1

(pip_test) C:\Temp\pip_test>pip show pandas
Name: pandas
Version: 2.2.2
Summary: Powerful data structures for data analysis, time series, and
```

```
d statistics
Home-page: https://pandas.pydata.org
Author:
Author-email: The Pandas Development Team <pandas-dev@python.org>
License: BSD 3-Clause License
(... 이하 생략...)
```

anaconda 패키지 관리자를 사용해 pandas 설치하기



miniconda 또는 anaconda가 설치된 환경에서만 이 방식을 사용할 수 있습니다.

- 이하 과정은 모두 파이썬 대화형 환경이나 파이썬 프로그래밍 환경이 아닌 터미널 상에서 실행해야 합니다.
- 현재 환경에 설치된 모든 패키지의 목록은 `conda list` 로 확인할 수 있습니다.
- 특정 패키지만 보려면 `conda list 패키지명` 으로 확인할 수 있습니다.

```
(conda_test) C:\Temp\pip_test>conda list
# packages in environment at C:\Users\Edberg\miniconda3\envs\conda_test:
#
# Name                               Version                               Build Channel

(conda_test) C:\Temp\pip_test>conda list pandas
# packages in environment at C:\Users\Edberg\miniconda3\envs\conda_test:
#
# Name                               Version                               Build Channel
```

- pandas가 (사실은 어떤 패키지도) 설치되어 있지 않으므로 pandas 패키지를 설치합니다.
 - `conda install 패키지명` 명령으로 설치할 수 있습니다.
 - 설치 과정에서 계속 진행할 지의 여부를 물어보기도 합니다. 그냥 엔터를 누르면 계속 설치가 진행됩니다.

```
(conda_test) C:\Temp\pip_test>conda install pandas
Channels:
- defaults
Platform: win-64
(... 중간 생략 ...)
(설치 중인 터미널이 깨끗이 지워진 후)
done
```

- pandas 설치 후에 설치된 패키지를 확인해봅시다.


```
(conda_test) C:\Temp\pip_test>conda list
# packages in environment at C:\Users\Edberg\miniconda3\envs\conda_test:
#
# Name                                Version                                Build      Channel
blas                                  1.0                                    mkl
bottleneck                           1.3.7                                py312he558020_0
bzip2                                1.0.8                                h2bbff1b_6
ca-certificates                      2024.7.2                              haa95532_0
expat                                 2.6.3                                h5da7b33_0
intel-openmp                         2023.1.0                              h59b6b97_46320
libffi                                3.4.4                                hd77b12b_1
mkl                                   2023.1.0                              h6b88ed4_46358
mkl-service                          2.4.0                                py312h2bbff1b_1
mkl_fft                              1.3.10                               py312h827c3e9_0
mkl_random                           1.2.7                                py312h0158946_0
numexpr                              2.8.7                                py312h96b7d27_0
numpy                                 1.26.4                               py312hfd52020_0
numpy-base                           1.26.4                               py312h4dde369_0
openssl                              3.0.15                               h827c3e9_0
pandas                               2.2.2                                py312h0158946_0
pip                                  24.2                                 py312haa95532_0
python                               3.12.5                               h14ffc60_1
python-dateutil                      2.9.0post0                           py312haa95532_2
python-tzdata                        2023.3                               pyhd3eb1b0_0
pytz                                  2024.1                               py312haa95532_0
setuptools                           72.1.0                               py312haa95532_0
six                                   1.16.0                               pyhd3eb1b0_1
sqlite                                3.45.3                               h2bbff1b_0
tbb                                   2021.8.0                              h59b6b97_0
tk                                    8.6.14                               h0416ee5_0
tzdata                               2024a                                h04d1e81_0
vc                                    14.40                                h2eaa2aa_1
vs2015_runtime                       14.40.33807                           h98bb1dd_1
wheel                                 0.44.0                               py312haa95532_0
xz                                    5.4.6                                h8cc25b3_1
zlib                                  1.2.13                               h8cc25b3_1
```

- pandas 패키지만 설치했음에도 불구하고 수많은 다른 패키지가 설치되었음을 확인할 수 있습니다.
- 이와 같이 어떤 패키지는 동작하는데 다른 패키지가 필요한 경우가 있습니다. 이를 “의존성이 있는 모듈”이라고 부르며, pip나 anaconda 패키지 관리자는 필요한 모든 의존성이 있는 모듈도 함께 설치합니다.

설치가 끝난 pandas 패키지 사용하기

- 표준 라이브러리에 포함된 모듈과 다를 바 없이 사용할 수 있습니다.

- 이때 설치에 사용한 가상환경이 활성화 되어 있는지 먼저 확인해야 합니다.

```
import pandas as pd
print(pd.__version__)
df = pd.DataFrame()
df.info()
```