15. 함수

최희윤 강사

함수 (function)

프로그래밍에서 함수(function)는 독립적으로 설계된 프로그램 코드의 집합
특정한 기능을 반복해서 사용 해야할 때 생성

메인코드

```
x = 함수명(인수, 인수, …)
print('결과 값: ', x)
```

함수 정의부

```
def 함수명(파라미터, 파라미터, …):
        함수 내부 코드 …
        return 반환값
```

함수가 호출되면 함수 정의부로 가서 함수 호출부의 인수 값을 함수 정의부 파라미터 값에 순서대로 대입 후
함수 정의부 내부 코드를 순차적으로 실행
해당 함수에 return값이 있는 경우, 함수 호출부를 return값으로 치환

용어 정리

✓ 파라미터 (Parameter) / 매개변수 / 인자
- 함수를 정의할 때 사용되는 변수
- 함수 내부에서 데이터를 받는 형식적인 자리표시자
- 함수 선언부에서 사용 됨
- 함수 호출 시 외부로부터 전달받은 값을 함수 내부로 전달

✓ 인수 (Argument)
- 함수를 호출할 때 실제로 전달하는 값 또는 표현식
- 함수 호출부에서 사용 됨
- 함수가 실행되면서 파라미터에 전달되어, 함수 내부에서 사용 됨

✓ 즉, 파라미터는 어떤 종류의 인자를 받을지 정의

함수 특징

✓ 함수의 특징

- 코드의 중복을 줄일 수 있어서 유지보수에 좋음
  - 코드를 목적에 맞게 사용할 수 있음
  - 중복에서 오는 실수를 줄일 수 있음
  - 재사용할 수 있음

함수(Function) vs 메서드(Method)

✓ 함수 (Function)

클래스에 포함되지 않은 채로 사용됨

함수명(인수1, 인수2, ...)

✓ 메서드 (Method)

- 클래스에 포함되어 객체를 통해 호출 됨
- 클래스를 다룰 때 더 자세히 배울 예정! ☺

객체.메서드명(인수1, 인수2, ...)

함수 정의 (define)

- ✓ 함수는 미리 정의해둔 정의절을 실행해야 호출이 가능
- ✓ 함수의 정의는 아래와 같은 형태로 작성할 수 있음

```
def 함수_이름(위치_인자, 위치_인자, *가변_위치, 키워드_인자, 키워드_인자2, **가변_키워드):
    """Doc String을 적는 곳

    작성자: 최희윤
    작성일자: 2024.11.25\
    """

    함수의 바디
    함수의 바디
```

함수 정의 (define)

✓ 함수 정의 예제

```
def my_function(a, b):
    """정수 a, b를 입력받으면 a + b를 반환하는 함수이다."""
    return a + b

print(my_function.__doc__)
print(my_function(2, 3))
>> 정수 a, b를 입력받으면 a + b를 반환하는 함수이다.
>> 5
```

함수 정의 (define)

✓ 함수 정의 예제

```
def my_function(a, b):
    # DocString은 없어도 됨
    return a + b

print(my_function.__doc__)
print(my_function(2, 3))
>> None
>> 5
```

함수정의 실습 01

✓ [문제] 자연수를 인수로 전달하면 짝수일 때 '짝수', 홀수일 때 '홀수'라는 문자열을 반환하는 함수 생성해보기

- 자연수에 대해서만 '짝수', '홀수' 반환
- 자연수가 아닌 값이 들어왔을 때 반환 값 없이 함수를 즉시 종료

```
print(odd_even().__doc__)
print(odd_even(10))
print(odd_even(9))
print(odd_even('가'))

>> number가 짝수면 '짝수', 홀수면 '홀수' 반환
>> 짝수
>> 홀수
>> None
```

함수정의 실습 02

✓ [문제] 인수로 연도 값을 입력 받습니다. 윤년을 나타내는 연도가 입력되면 '윤년'이라는 문자열을 반환하고,
그렇지 않은 수가 입력되면 평년이라는 문자열을 반환하는 함수를 만들어보세요.

- 윤년은 기본적으로 4로 나눠지는 연도입니다.
- 예외사항: 100으로 나눠지면서 400으로 나눠지지 않는 연도는 평년입니다.

➢ 실행 결과:

```
check_leap_year(2004)
반환 값: '윤년'
```

➢ 실행 결과:

```
check_leap_year(2000)
반환 값: '윤년'
```

➢ 실행 결과:

```
check_leap_year(1900)
반환 값: '평년'
```

인자와 반환 값 1

✓ 인자와 반환 값이 없는 함수

```
def my_func():
    print('Hello, World!')


print(my_func())
>> Hello, World!
   None
```

✓ 인자는 있으나 반환 값이 없는 함수

```
def my_func(name):
    print('Hello, ' + name)


print(my_func('Daisy'))
>> Hello, Daisy
   None
```

인자와 반환 값 2

✓ 인자와 반환 값이 있는 함수

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


print(my_func(2, 3))
>> 5
```

✓ 값을 여러 개 반환

```
def my_func(a, b):
    return a + b, a - b


print(my_func(2, 3))
>> (5, -1)
```

파라미터(인자)

✓ 파라미터(parameter) / 인자의 종류

- 위치-키워드(positional or keyword): 위치로 혹은 키워드로 전달될 수 있는 인자 (매개변수의 기본형태)
- 위치 전용 (positional-only): / 문자를 기준으로 좌측에 위치한 인자
- 키워드 전용 (keyword-only): *문자를 기준으로 우측에 위치한 인자
- 가변 위치 (var positional): 위치 매개변수 외에 추가적인 위치 매개변수를 개수에 상관없이 받을 수 있음
  변수명 앞에 *를 하나 더 붙여 표기
- 가변 키워드 (var keyword): 키워드 매개변수 외에 추가적인 키워드 매개변수를 개수에 상관없이 받을 수 있음
  변수명 앞에 **를 두 개 붙여 표기

파라미터(인자): 위치-키워드 인자

✓ 위치-키워드 인자 (positional-or-keyword parameters)

- 위치 인수 (positional arguments) 또는 키워드 인수 (keyword arguments)를 받을 수 있는 인자(파라미터)
- 즉, 키워드로 전달하지 않을 경우 인자의 위치에 맞게 인수를 기입해야 함
- 아래는 위치-키워드 인자가 위치 인수를 전달 받아 처리하는 예제이다.

```
# 함수 정의
def greeting(name, age):
    print(f'{name}씨, 안녕하세요. 약 {age * 365.25}일 되었습니다.')

# 함수 호출
greeting('파이썬', 32)
>> 파이썬씨, 안녕하세요. 약 11688.0일 되었습니다.
```

파라미터(인자): 위치-키워드 인자

✓ 위치-키워드 인자 (positional-or-keyword parameters)

- 기본값이 없는 인자에 위치 인수를 순서와 관계 없이 전달한 경우 -> 에러 발생

```
# 함수 정의
def greeting(name, age):
    print(f'{name}씨, 안녕하세요. 약 {age * 365.25}일 되었습니다.')

# 함수 호출
greeting(32, '파이썬')    # 위치 상관 없이 기입
>> TypeError: can't multiply sequence by non-int of type 'float'
```

파라미터(인자): 위치-키워드 인자

✓ 위치-키워드 인자 (positional-or-keyword parameters)

- 키워드 인수로 함수를 호출할 경우 -> 키워드 인수 간에 순서는 상관 없음

```
# 함수 정의
def greeting(name, age):
    print(f'{name}씨, 안녕하세요. 약 {age * 365.25}일 되었습니다.')

# 함수 호출
greeting(age=32, name='파이썬')        # 키워드 인수로 호출
>> TypeError: can't multiply sequence by non-int of type 'float'
```

파라미터(인자): 위치-키워드 인자

✓ 위치-키워드 인자 (positional-or-keyword parameters)

- 기본값이 없는 위치-키워드 인자일 경우, 인수를 받지 못하면 에러 발생

```
# 함수 정의
def greeting(name, age): # 기본 값이 없는 인자
    print(f'{name}씨, 안녕하세요. 약 {age * 365.25}일 되었습니다.')

# 함수 호출
greeting('파이썬')          # age에 해당하는 인수를 전달하지 않음
>> TypeError: greeting() missing 1 required positional argument: 'age'
```

파라미터(인자): 위치-키워드 인자

✓ 위치-키워드 인자 (positional-or-keyword parameters)

- 기본 값을 갖는 default parameter는 기본 값을 갖지 않는 non-default parameter보다 뒤에 작성해야 함

```
# 함수 정의
def greeting(name='default', age):    # 기본 값이 있는 인자
    print(f'{name}씨, 안녕하세요. 약 {age * 365.25}일 되었습니다.')

# non-default parameter가 먼저 오고 default parameter를 선언해야 함
>> SyntaxError: non-default argument follows default argument
```

파라미터(인자): 위치 인수와 키워드 인수

✓ 위치 인수와 키워드 인수

- 함수를 호출할 때, 소괄호() 안에
  위치 인수 (positional arguments)를 먼저 작성하고 -> **키워드 인수** (keyword arguments)를 나중에 작성한다.
- 위치인수는 순서에 영향을 받지만, 키워드 인수는 순서 상관없이 작성 가능하다.

```
# 함수 정의
def greeting(name, age): # 기본 값이 없는 인자
    print(f'{name}씨, 안녕하세요. 약 {age * 365.25}일 되었습니다.')

# 함수 호출
greeting(name='파이썬', 32)  # age에 해당하는 인수를 전달하지 않음
>> TypeError: positional argument follows keyword argument
```

파라미터(인자): 위치 전용 인자

✓ 위치 전용 인자 (positional-only parameters)

- 오직 위치 인수(positional arguments)의 값만 전달 가능
- 선언하는 방법: 슬래시 (/)를 인자 값으로 넣고 좌측 부분에 위치 전용 인자 선언

```
# 함수 정의
def position_only(posonly, /):
    print(posonly)

# 함수 호출
position_only('값만 입력해야 합니다.')
>> 값만 입력해야 합니다.
```

파라미터(인자): 위치 전용 인자

✓ 위치 전용 인자 (positional-only parameters)

- 위치 전용 인자에 키워드 인수 (keyword arguments)를 전달하면 TypeError 오류 발생

```
# 함수 정의
def position_only(posonly, /):
    print(posonly)

# 함수 호출
position_only(posonly='값만 입력해야 합니다.')
>> TypeError: position_only() got some positional-only arguments …
```

파라미터(인자): 키워드 전용 인자

✓ 키워드 전용 인자 (keyword-only parameters)

- 키워드 전용 인자 (keyword-only parameters)는 오직 위치 인수(keyword arguments)만 전달 가능
- 선언 방법: 애스터리스크(*)를 인자 값으로 넣고 우측에 키워드 전용 인자를 선언

```
# 함수 정의
def key_only(*, keyonly):
    print(keyonly)

# 함수 호출
key_only(keyonly='키워드로만 입력해야 합니다.')
>> 키워드로만 입력해야 합니다.
```

파라미터(인자): 키워드 전용 인자

✓ 키워드 전용 인자 (keyword-only parameters)

- 키워드 전용 인자 (keyword only parameters)도 기본 값을 가질 수 있음
- 이 때 기본값을 갖는 인자에 상응하는 인수(arguments)는 생략할 수 있음

```
# 함수 정의
def key_only(*, keyonly='default'):
    print(keyonly)

# 함수 호출
key_only()  # 기입하려는 인수가 인자의 default값과 일치한다면 생략 가능
>> default
```

파라미터(인자): 키워드 전용 인자

✓ 키워드 전용 인자 (keyword-only parameters)

- 키워드 전용 인자 (keyword only parameters)에 위치 인수(positional arguments)를 전달하면 TypeError 발생

```
# 함수 정의
def key_only(*, keyonly):
    print(keyonly)

# 함수 호출
key_only('키워드로만 입력해야 합니다.')
>> TypeError: key_only() takes 0 positional arguments but 1 was given
```

파라미터(인자): 가변-위치 인자

✓ 가변-위치 인자 (var-positional parameters)

- 명시된 인자 외에 추가적으로 위치 인수를 개수에 상관없이 유연하게 전달받을 수 있음
- 가변 변수로 사용할 변수명 앞에 *를 하나 붙여 표기

```
# 함수 정의
def var_positional(*args):
    print(type(args))
    return sum([i for i in args])

# 1, 2, 3, 4, 5의 위치 인수들을 tuple로 packing한 뒤 전달됨
print(var_positional(1, 2, 3, 4, 5))
>> <class 'tuple'>
   15
```

파라미터(인자): 가변-위치 인자

✓ 가변-위치 인자 (var-positional parameters)

- 인자에 값을 전달하는 인수가 아예 없어도 함수는 잘 동작 함
- 인수를 전달 받지 못하더라도 함수가 동작하는데 문제 없도록 함수 바디를 작성하는 것이 좋음

```
# 함수 정의
def var_positional(*args):
    print(type(args))
    return sum([i for i in args])

# 전달할 위치 인수 생략
print(var_positional())
>> <class 'tuple'>
   0
```

파라미터(인자): 가변-키워드 인자

✓ 가변-키워드 인자 (var-keyword parameters)

- 명시된 인자 외에 추가적으로 키워드 인수를 개수에 상관없이 유연하게 전달받을 수 있음
- 가변 변수로 사용할 변수명 앞에 *를 두 개 붙여 표기

```
# 함수 정의
def var_keyword(**kargs):
    print(type(kargs))
    return kargs

# 전달할 키워드 인수들을 dict로 packing한 뒤 전달
print(var_keyword(key='value', key2='value2'))
>> <class 'dict'>
   {'key': 'value', 'key2': 'value2'}
```

파라미터(인자): 가변-키워드 인자

✓ 가변-키워드 인자 (var-keyword parameters)

- 해당 매개변수에 값을 전달하는 인수가 아예 없어도 함수가 동작하는데 문제 없도록 함수 바디를 작성하는 것이

```
# 함수 정의
def var_keyword(**kargs):
    print(type(kargs))
    return kargs

# 전달할 키워드 인수 생략
print(var_keyword())
>> <class 'dict'>
   {}
```

파라미터(인자) 실습 01

✓ [문제] 각 입력받은 정수를 10씩 더해서 반환하는 함수를 만들어 보세요.

- 함수명: plus_ten

➢ 실행 결과:

```
plus_ten(70)
반환 값: 80
```

변수의 범위

✓ 전역 변수 (global variable)

- 전역 범위(global scope): 함수 바깥 영역 (들여쓰기 없이 쓴 영역)
- 전역 변수: 전역 범위에서 선언한 변수 -> 스크립트 전체에서 접근할 수 있음

```
global_variable = “This is global world” # 전역 변수

print(f“global_variable in global scope => {global_variable}”)

>> global_variable in global scope => This is global world

def my_world(name):
    a = name
    return a
```

변수의 범위 - 전역 변수

✓ 전역 변수 (global variable)

- 전역 범위(global scope): 함수 바깥 영역 (들여쓰기 없이 쓴 영역)
- 전역 변수: 전역 범위에서 선언한 변수 -> 스크립트 전체에서 접근할 수 있음

```
global_variable = “This is global world” # 전역 변수

print(f“global_variable in global scope => {global_variable}”)
>> global_variable in global scope => This is global world

def local_world(world): # 함수 내부에서 전역변수에 대한 조회 가능
     print(f“global_variable in global scope => {global_variable}”)

local_world()
>> global_variable in global scope => This is global world
```

변수의 범위 - 지역 변수

✓ 지역 변수 (local variable)

- 지역 범위(local scope): 함수 바디 영역 (함수 내부 영역)
- 지역 변수: 지역 범위에서 (함수 바디에서)선언한 변수 -> 함수 안에서만 접근할 수 있음

```
def local_world(world):
    my_variable = “This is my little world.”

print(my_variable) # 지역 변수를 전역 범위에서 선언
>> NameError: name “my_variable” is not defined # 오류 발생
```

변수의 범위

✓ 지역 범위에서 전역 변수 할당하기

- 전역 변수의 값을 지역 범위에서 변경해보자
- 변수를 호출 했을 때 어떤 값이 출력되는지 확인해보기

```
important_is_an_unbroken_heart = '중요한 것은 꺾이지 않는 마음'

def trials_and_tribulations():
    important_is_an_unbroken_heart = '흔들흔들' # 전역 변수의 값 변경 시도

print(important_is_an_unbroken_heart)
>> 중요한 것은 꺾이지 않는 마음 # 값이 변경되지 않음을 알 수 있다.
```

변수의 범위

✓ 지역 범위에서 전역 변수 할당하기

- global 키워드를 활용하여 지역 범위에서 전역 변수 변경 가능

```
global_variable = '변경이 될까?'

def heartbreaker():
    global global_variable # global 키워드 활용하여 지역 범위에서 전역변수 사용 설정
    global_variable = 'global 키워드를 활용해서 변경 완료'

heartbreaker() # 함수를 선언해줘야 값 변경이 실행이 됨
print(global_variable)
>> global 키워드를 활용해서 변경 완료
```

중첩 함수

✓ 함수 바디에 함수를 중첩해서 만들기

- 함수 바디 안에 def를 써서 다시 함수를 만들 수 있음

```
def level_1():
    message = “This is level 1”
    def level_2():
        print(message) # message 변수에 접근할 수 있는 범위
    level_2() # level_1() 범위에서 level_2()를 호출해줘야됨

level_1()
>> This is level 1
```

중첩 함수

✓ 안쪽 함수에서 바깥쪽 함수의 변수 변경해보기

```
def level_1():
    message = “This is level 1”
    def level_2():
        message = “Level 2 is better than level 1” # message 값 변경
    level_2()
    print(message)

level_1()
>> This is level 1 # message 값 변경 안 됨
```

중첩 함수

✓ 안쪽 함수에서 바깥쪽 함수의 변수 변경해보기

```
def level_1():
    message = “This is level 1”
    def level_2():
        message = “Level 2 is better than level 1” # message 값 변경
    level_2()
    print(message) # 즉, 여기서 호출된 message는 level_1의 message임

level_1()
>> This is level 1
```

새로운 범위의 지역변수 message를 만든 것
-> 바깥쪽 변수 값 변경 실패

중첩 함수

✓ 안쪽 함수에서 바깥쪽 함수의 변수 변경해보기

```
def level_1():
    message = “This is level 1”
    def level_2():
        nonlocal message # nonlocal 키워드를 통해 바깥 영역의 변수 사용 설정
        message = “Level 2 is better than level 1” # message 값 변경
    level_2()
    print(message)

level_1()
>> Level 2 is better than level 1 # message 값 변경 성공
```

중첩 함수

✓ global, nonlocal 특징과 권고 사항

- global 키워드는 함수의 중첩된 정도와 상관없이 전역 범위의 변수를 매칭
- 중첩된 함수마다 같은 이름의 변수가 있다면 nonlocal 키워드는 제일 가까운 바깥 변수 매칭
- 가급적이면 함수마다 이름이 같은 변수를 사용하기 보단, 다른 변수명을 사용하는 게 좋음

✓ 변수 범위 정리

- 함수 안에서 선언한 변수는 함수를 호출해서 실행되는 동안만 사용 가능
- 범위마다 같은 이름의 변수를 사용해도 각각 독립적으로 동작
- 지역 변수(local variable)를 저장하는 이름 공간을 지역 영역(local scope)라고 함
- 전역 변수(global variable)를 저장하는 이름 공간을 전역 영역(global scope)라고 함
- 파이썬 자체에서 정의한 이름 공간을 내장 영역(built-in scope)이라고 함
- 함수에서 변수를 호출하면 지역 영역 -> 전역 영역 -> 내장 영역 순으로 해당하는 변수를 확인

16. 람다 (Lambda)

최희윤 강사

람다 (Lambda)

- ✓ 호출될 때 값이 구해지는 하나의 표현식
- ✓ 이름이 없는 인라인 함수

```
lambda parameters : expression
```

매개변수 지정 — 반환 값으로 사용할 식

→ Lambda 표현식

```
def func_name(parameter):
    return expression
```

반환 값

→ 일반 함수 정의절

람다 (Lambda)

✓ 람다 표현식을 바로 호출하는 방법

- 람다 표현식 전체를 소괄호()로 감싸고 뒤에 함수 호출하듯이 소괄호를 붙임
- 그리고 뒷쪽에 작성한 소괄호 안에 인수를 넣으면 람다 표현식이 바로 호출 됨

```
(lambda x: x + 10)(10)
>> 20
```

(lambda parameters : expression)(arguments)

람다 (Lambda)

✓ 람다 표현식을 변수를 통해 호출하기

- 람다는 기본적으로 이름이 없는 함수임
- 람다로 만든 익명 함수를 호출하려면, 변수에 할당해서 사용할 수 있음

```
twice = lambda x: x * 2
>> print(twice(10))
```

variable = lambda parameters : expression

variable(argument)

람다 (Lambda)

✓ 람다는 한 줄로 표현하는 함수

- 람다의 expression 부분은 변수 없이 식 한 줄로 표현 가능해야 함
- 따라서, 표현식 안에 새 변수를 만들 수 없음
- 변수가 필요한 경우 def를 써서 함수를 정의해서 사용해야 함

```
(lambda x: x + y)(1)
>> SyntaxError: invalid syntax
```

18. 클래스 (Class)

최희윤 강사

객체지향(Object Oriented)프로그래밍

✓ 객체지향 프로그래밍

- 복잡한 문제를 잘게 나누어 객체로 만들고, 객체를 조합해서 문제를 해결
- 즉, 프로그램을 객체(object)단위로 나누고, 이 객체들이 서로 상호작용하며 동작하도록 설계하는 프로그래밍
- 현실 세계의 복잡한 문제를 처리하는데 유용
- 기능을 개선하고 발전시킬 때도 해당 클래스만 수정하면 되므로 큰 프로젝트의 유지보수에도 매우 효율적
- 객체가 가진 데이터를 클래스의 속성(Attribute)이라 부르고 객체가 갖는 기능을 메서드(Method)라고 함

객체(object)와 클래스(Class)

객체(인스턴스) : 속성(변수)과 행동(함수)으로 구성된 대상

클래스 : 객체(인스턴스)를 만들기 위한
도구/문법/템플릿/청사진/설계도

클래스 자체는 독립적으로 동작하지 않음. 객체가 생성되어야만 사용 가능

```
class ClassName:      # 클래스명으로 주로 PascalCase(UpperCamelCase)를 씁니다.
    def method_name(self):
        method_body
    class_body
    ...
```

소스코드

18-1 객체와 클래스.ipynb

클래스(Class)

✓ 클래스 (Class)

- 지금까지 사용해온 int, list, dict 등도 클래스임
- 우리는 이러한 클래스로부터 인스턴스를 생성하고 메서드를 사용해왔음

```
number = int(10.0)
print(type(number))
>> <class ‘int’>

num_list = list(range(10))
print(type(num_list))
>> <class ‘list’>
```

인스턴스(Instance)

✓ 인스턴스란?

- 클래스(Class)를 기반으로 생성된 구체적인 객체를 의미
- 즉, 인스턴스 = 클래스이름()에서 만들어진 객체

instance_variable = ClassName()

인스턴스

인스턴스(Instance)

✓ 인스턴스와 객체

- 객체는 굳이 어떤 클래스를 통해 생성된 객체인지는 언급하지 않음
- 인스턴스는 특정 클래스를 통해 만들어진 객체를 언급할 때 쓰임
- ex) dog = Dog()일 때, dog는 객체이고 dog는 Dog의 인스턴스라고 표현할 수 있음

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

dog1 = Dog("Buddy", 3) # Dog 클래스에 대한 인스턴스 생성
dog2 = Dog("Max", 5)
```

메서드(Method)

✓ 메서드란?

- 클래스 바디 안에서 정의되는 함수
- 클래스의 인스턴스의 속성(attribute)으로 호출되면, 그 메서드는 첫 번째 인자로 인스턴스 객체를 받음
- 첫 번째 인자를 설정 안 하면 에러 발생
- 이 첫 번째 인자를 'self'라고 씀

```
class ClassName:
    def method_name(self):
        method_body
    class_body
    ...
```

메서드(Method)

✓ 메서드란?

- 클래스의 인스턴스 속성(attribute)으로 호출되면, 그 메서드는 첫 번째 인자로 인스턴스 객체를 받음

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

dog = Dog(“Buddy”, 3) # 인스턴스 생성
print(dog.name) # 인스턴스 속성(name)으로 호출 -> 해당 메서드는 __init__ 함수
```

클래스 속성(Attribute)

✓ 클래스 속성(Attribute)이란?

- 모든 객체(인스턴스)가 공유하는 속성
- 클래스 메소드에서 객체 없이 클래스명으로 접근 가능

```
class Dog:
    dog_count = 0 # 클래스 속성
    def __init__(self, name):
        self.name = name # 인스턴스 속성
        Dog.dog_count += 1 # 클래스 속성 접근

dog = Dog(“Buddy”, 3)
print(dog.name)
>> Buddy
```

생성자 함수

✓ 생성자 함수 __init__

- 생성자 함수 __init__ 메서드는 james = Person() 처럼 클래스에 소괄호()를 붙여서 **인스턴스를 만들 때 호출되는 특별한 메서드 임**
- __init__은 initialize의 줄임말로 인스턴스(객체)를 초기화(메모리에 공간을 할당하고 값을 부여) 함.
- 언더바 두 개 (__, double under, 던더)가 양 옆으로 붙어있는 메서드는 파이썬이 자동으로 호출하는 메서드
  - 스페셜 메서드(special method) 혹은 매직 메서드 (magic method), 던더 메서드로 불림
- 파이썬의 여러 기능을 사용할 때 이 던더 메서드를 구현하는 식으로 사용하게 됨

```
class Person:
    def __init__(self):
        self.hello = '안녕하세요'
```

인스턴스 속성

✓ 인스턴스 속성

- 인스턴스 속성은 __init__ 메서드에서 만듦
- self에 마침표를 붙여 속성명을 붙이고 값을 할당
- 클래스 바디에서도 속성을 접근할 때 **self.속성**과 같이 self에 마침표를 찍고 사용하면 됨

```
class Person:
    def __init__(self):
        self.hello = '안녕하세요'

    def greeting(self):
        print(self.hello)

james = Person()
james.greeting()
>> 안녕하세요
```

self

✓ slef의 의미

- 메서드가 호출된 **현재 객체(인스턴스)**를 참조
- 클래스 내부에서 객체의 속성이나 메서드에 접근할 때 사용됨
- 반드시 첫 번째 매개변수로 사용하며, 관례적으로 self를 사용

self

✓ slef의 의미

- dog1.bark()를 호출하면 self는 dog1을 참조
- dog2.bark()를 호출하면 self는 dog2를 참조
- 이처럼 여러 인스턴스를 만들 때 구분을 위한 역할로 사용

```
class Dog:
      def __init__(self, name):
            self.name = name # 인스턴스 속성
      def bark(self):
            print(f"{self.name}가 짖습니다!")
# 객체 생성
dog1 = Dog("바둑이")
dog1 = Dog("흰둥이")

dog1.bark() # 출력: 바둑이
dog2.bark() # 출력: 흰둥이
```

self와 속성

✓ self를 사용하지 않으면 name과 age는 일반 로컬 변수 취급을 받는다 (로컬변수: 함수 내부에서만 사용가능)

```
class Dog:
    def __init__(self, name, age):
        self.name = name # 인스턴스 속성
        self.age = age

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

# 객체 생성
dog1 = Dog("바둑이", 5)
dog1 = Dog("흰둥이", 3)

print(dog1.name) # 출력: 바둑이
print(dog2.age) # 출력: 3
```

```
class Dog2:
    def __init__(self, name, age):
        name = name # self를 생략한 잘못된 코드
        age = age

# 객체 생성
dog = Dog2("바둑이", 5)
print(dog.name)

>> AttributeError: 'Dog2' object has no attribute 'name'
```

self와 속성

- ✓ 인스턴스를 생성할 때 속성 값을 할당하려면 다음 예제와 같이 __init__메서드에서 self 다음에 받을 값을 매개변수로 지정해야 한다.
- ✓ 그리고 매개변수를 **self.속성**에 할당한다.
- ✓ 그 후 인스턴스를 만들 때 추가된 매개변수만큼 인자를 넘겨줘야 한다.

```
class ClassName:
    def __init__(self, param1, param2):
        self.attr1 = param1 # 인스턴스 속성
        self.attr2 = param2
```

self와 속성

✓ 예제를 통해 self와 속성의 관계성 이해하기

- 인스턴스를 만들 때 이름, 나이, 주소를 받는다
- 그 후 인스턴스의 속성에 접근해서 출력

```
class Person:
    def __init__(self, name, age, address):
        self.hello = "안녕하세요."
        self.name = name
        self.age = age
        self.address = address

    def greeting(self):
        print(f"{self.hello} 제 이름은 {self.name}입니다.")
```

```
maria = Person("마리아", 20, "서울시 서초구 반포동")
maria.greeting() # 안녕하세요. 제 이름은 마리아입니다.


print("이름: ", maria.name) # 마리아
print("나이: ", maria.age) # 20
print("주소: ", maria.address) # 서울시 서초구 반포동
```

비공개 속성 사용하기

✓ 비공개 속성이란?

- **'인스턴스.속성 = 값'**을 통해 새롭게 인스턴스의 속성 값을 변경하지 못하도록 막고 싶을 때는 비공개 속
- 사용법: 속성 앞에 던더(__)를 붙이면 비공개 속성이 됨

```
class ClassName:
    def __init__(self, param1, param2):
        self.attr1 = param1
        self.__attr2 = param2
```

비공개 속성

비공개 속성 사용하기

✓ Person클래스에 '__wallet'이라는 이름으로 속성을 추가해보자

```
class Person:
    def __init__(self, name, age, address, wallet):
        self.name = name
        self.age = age
        self.address = address
        self.__wallet = wallet # 변수 앞에 __를 붙여서 비공개 속성으로 만듦

maria = Person('마리아', 20, '서울시 서초구 반포동', 10000)
maria.__wallet -= 10000 # 클래스 바깥에서 비공개 속성에 접근하면 에러 발생
>> AttributeError: 'Person' object has no attribute '__wallet'
```

비공개 속성 사용하기

✓ Person클래스에 Pay 메서드를 정의해서 사용해보자

```
class Person:
    def __init__(self, name, age, address, wallet):
        self.name = name
        self.age = age
        self.address = address
        self.__wallet = wallet
    def pay(self, amount):
        if self.__wallet < amount:
            print('돈이 모자랍니다')
        self.__wallet -= amount
        print('이제 {0}원 남았네요.'.format(self.__wallet))
```

```
maria = Person('마리아', 20, '서울시 서초구 반포동', 10000)
maria.pay(3000)
>> 이제 7000원 남았네요
```

클래스 복습

```
class Dog:

    def __init__(self, name, color):
        self.hungry = 0
        self.name = name
        self.color = color

    def eat(self):
        self.hungry -= 10
        print('밥 먹음', self.hungry)

    def walk(self):
        self.hungry += 10
        print('산책', self.hungry)
```

0.0s

객체
생성자함수

```
choco = Dog('choco', 'black')
```

클래스 복습

객체 생성자함수

Self : 객체 (인스턴스) 자기 자신

```
class Dog:

    def __init__(self, name, color):
        self.hungry = 0
        self.name = name
        self.color = color

    def eat(self):
        self.hungry -= 10
        print('밥 먹음', self.hungry)

    def walk(self):
        self.hungry += 10
        print('산책', self.hungry)
```

0.0s

클래스 복습

객체 생성자함수

Self : 객체 (인스턴스) 자기 자신

인스턴스 기본 속성

```
class Dog:

    def __init__(self, name, color):
        self.hungry = 0
        self.name = name
        self.color = color

    def eat(self):
        self.hungry -= 10
        print('밥 먹음', self.hungry)

    def walk(self):
        self.hungry += 10
        print('산책', self.hungry)
```

0.0s

```
choco = Dog('choco', 'black')
choco.hungry
```

✓ 0.0s

```
0
```

클래스 복습

객체 생성자함수

인스턴스 메소드

Self : 객체 (인스턴스) 자기 자신

인스턴스 기본 속성

```
class Dog:

    def __init__(self, name, color):
        self.hungry = 0
        self.name = name
        self.color = color

    def eat(self):
        self.hungry -= 10
        print('밥 먹음', self.hungry)

    def walk(self):
        self.hungry += 10
        print('산책', self.hungry)
```

0.0s

```
choco = Dog('choco', 'black')
choco.hungry
choco.eat()
choco.eat()
```

✓ 0.0s

```
밥 먹음 -10
밥 먹음 -20
```

클래스 복습

```
class Dog:

    def __init__(self, name, color):
        self.hungry = 0
        self.name = name
        self.color = color

    def eat(self):
        self.hungry -= 10
        print('밥 먹음', self.hungry)

    def walk(self):
        self.hungry += 10
        print('산책', self.hungry)
```

0.0s

Self : 객체 (인스턴스) 자기 자신

객체 생성자함수

인스턴스 기본 속성

인스턴스 메소드

```
choco = Dog('choco', 'black') # 클래스Dog의 객체 생성
jjong = Dog('jjong', 'white')


choco.eat()
choco.eat()
jjong.walk()

print(choco.hungry)
print(jjong.hungry)
```

✓ 0.0s

```
밥 먹음 -10
밥 먹음 -20
산책 10
-20
10
```

생성자 함수를 통해 인스턴스 생성

메소드를 실행하고 난 후의 인스턴스의 속성 확인

클래스 복습

✓ 비공개 속성 사용하기

```python
class Dog:

    def __init__(self, name, color):
        self.name = name
        self.color = color
        self.__hungry = 0

    def eat(self):
        if self.__hungry <= 0:
            print('배가 너무 불러요!')
        else:
            self.__hungry -= 10
            print('밥 먹음', self.__hungry)

    def walk(self):
        self.__hungry += 10
        print('산책', self.__hungry)

    def condition(self):
        print(f'{self.name} 배고픔 : {self.__hungry}')
```

외부에서 속성에 접근하지 못하게 차단

속성명 앞에 (언더바 두 개)추가

```python
mery = Dog('mery', 'black')
mery.eat()
mery.walk()
mery.walk()
mery.condition()
```

✓ 0.0s

```
배가 너무 불러요!
산책 10
산책 20
mery 배고픔 : 20
```

클래스 복습

✓ 비공개 속성 사용하기

```python
class Dog:

    def __init__(self, name, color):
        self.name = name
        self.color = color
        self.__hungry = 0

    def eat(self):
        if self.__hungry <= 0:
            print('배가 너무 불러요!')
        else:
            self.__hungry -= 10
            print('밥 먹음', self.__hungry)

    def walk(self):
        self.__hungry += 10
        print('산책', self.__hungry)

    def condition(self):
        print(f'{self.name} 배고픔 : {self.__hungry}')
```

외부에서 속성에 접근하지 못하게 차단

속성명 앞에 __ (언더바 두 개)추가

```
mery.__hungry += 100
⊗ 0.0s
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[169], line 1
----> 1 mery.__hungry += 100

AttributeError: 'Dog' object has no attribute '__hungry'
```

__hungry에 직접 접근할 경우 에러발생

클래스 복습

✓ 클래스 속성: 모든 객체가 공유, 클래스 내에서는 객체 없이 클래스명으로 접근 가능

```
class Dog:

    dog__count = 0   # 클래스 속성

    def __init__(self, name, color):
        self.name = name   # 인스턴스 속성
        self.color = color
        Dog.dog__count += 1   # 클래스 속성 접근

    def dog_counting(self):
        print('총 강아지는: ', Dog.dog__count)
```

객체가 생성될 때 마다 +1 됨

```
hello = Dog('hello', 'black')
hello.dogCount()

happy = Dog('happy', 'white')
happy.dogCount()
```

✓ 0.0s

```
총 강아지는 : 1
총 강아지는 : 2
```

총 몇 개의 객체가 생성됐는지 알 수 있음

클래스 실습 01

✓ [문제]

- Person 클래스로 maria, james라는 인스턴스를 생성합니다
- maria와 james 두 사람의 나이, 이름, 주소를 받아와 객체에 저장하고 화면에 출력해보세요

➢ **실행 결과:**

```
이름은 무엇인가요?: 마리아
나이는 무엇인가요?: 20
주소는 무엇인가요?: 서울시 강남구
이름은 무엇인가요?: 제임스
나이는 무엇인가요?: 21
주소는 무엇인가요?: 서울시 구로구
첫 번째 이름: 마리아
첫 번째 나이: 20
첫 번째 주소: 서울시 강남구
두 번째 이름: 제임스
두 번째 나이: 21
두 번째 주소: 서울시 구로구
```

클래스 실습 02

✓ [문제]

- 사용자로부터 체력, 마나, AP를 입력받아 옵니다.
- 주어진 코드에서 애니(Annie)클래스를 작성하고 티버(tibbers)스킬의 피해량이 출력되게 만들어보세요.
- 티버 피해량은 AP * 0.65 + 400 이며, AP(Abbility Power, 주문력)는 마법 능력치를 뜻합니다.

➢ **실행 결과:**

```
체력, 마나, AP를 입력하세요: 511.68 334.0 298
티버: 피해량 593.7
```

➢ **실행 결과:**

```
체력, 마나, AP를 입력하세요: 1803.68 1184.0 645
티버: 피해량 819.25
```

데코레이터

✓ 데코레이터란?

- 함수나 메서드에 추가기능을 쉽게 추가할 수 있도록 도와주는 함수
- 다른 함수를 감싸는 함수
- 입력으로 함수를 받고, 수정된 함수를 반환
- 주로 함수나 메서드의 동작을 확장하거나 수정할 때 사용

데코레이터 정의

```
def decorator(func): # 다른 함수를 인자로 받음
    def wrapper(): # 감싸는 함수 (실제로 동작을 정의)
        print('함수 실행 전 작업')
        func() # 원래 함수 호출
        print('함수 실행 후 작업')
    return wrapper # 수정된 함수 반환
```

데코레이터 적용

```
@decorator # test_function()에 데코레이터 적용
def test_function():
    print('원래 함수 실행')

test_function()
```

데코레이터

✓ 데코레이터의 주요 특징

- 기존 코드를 수정하지 않고 기능 확장 가능: 함수에 새로운 기능 추가할 때 유용
- @문법 사용으로 간결함

✓ 데코레이터 사용 예제2

데코레이터 정의

```
def logger(func): # 데코레이터 함수
    def wrapper(*args, **kwargs): # 감싸는 함수 (실제로 동작을 정의)
        print('함수 실행 중인 함수: {func.__name__}')
        return func(*args, **kwargs) # 원래 함수 실행
    return wrapper # 수정된 함수 반환
```

데코레이터 적용

```
@logger
def say_hello(name):
    print(f'안녕하세요, {name}님!')

say_hello('철수')
>> 실행 중인 함수: say_hello
   안녕하세요, 철수님!
```

데코레이터 - 정적 메서드

✓ 정적 메서드 (static method)

- **@staticmethod** 데코레이터를 사용하여 정의
- 클래스나 인스턴스와 독립적으로 작동하는 메소드
- 클래스 내부에서 정의되지만, **클래스의 속성이나 인스턴스 속성을 사용하지 않는 메소드**
- 객체 지향 프로그래밍에서 클래스 내에 논리적으로 묶이지만, 인스턴스와 관련이 없는 기능을 구현할 때 사용
- 특정 데이터나 속성에 의존하지 않고, 클래스와 관련된 일반적인 기능을 제공할 때 유용
- 정적 메소드는 인스턴스를 통해서도 호출이 가능하다.
- 예) 날짜 계산, 문자열 처리 등의 작업
- 즉, 특정 기능이 클래스와 관련이 있는 경우(수학 계산과 같은 ..), 정적 메소드로 구현해 클래스를 유틸리티로 활용

* 유틸리티: 도움이 되는 기능이나 편리한 도구를 제공하는 것을 의미

데코레이터 - 정적 메서드

✓ 정적 메서드 예제

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def multiply(a, b):
        return a * b
```

```
# 정적 메소드는 클래스 이름으로 호출
print(Math.add(5, 3)) # 출력: 8
print(Math.multiply(5, 3)) # 출력: 15

# 정적 메소드는 인스턴스를 통해서도 호출 가능
math_instance = Math()
print(math_instance.add(10, 20)) # 출력: 30
```

데코레이터

✓ 클래스 메서드 (class method)

- **클래스 자체를 대상으로 동작**하는 메서드
- 클래스의 상태(속성)를 다루거나 클래스와 관련된 동작을 정의할 때 사용
- 즉, 클래스의 속성의 값을 읽거나 수정할 때 유용
- **@classmethod** 데코레이터를 사용하여 정의
- 첫 번째 매개변수로 'cls'를 받음
- 클래스 자체(cls)를 대상으로 동작하기 대문에 인스턴스와 독립적으로 작동
  - 인스턴스 없이 호출 가능 (인스턴스 호출 없이 동작 가능)
  - cls는 클래스 자체를 나타냄
  - 인스턴스 속성에는 접근 불가

```
class MyClass:
    @classmethod
    def add(cls, a, b):
        print(a + b)

MyClass.add(30, 40)
```

데코레이터

✓ 클래스 메서드 예제

```
class MyClass:
    class_variable = 0 # 클래스 속성

    @classmethod
    def increment_class_variable(cla):
        cls.class_variable += 1
        print(f"클래스 변수 값: {cls.class_variable}")

# 클래스 메서드 호출
MyClass.increment_class_variable() # 출력: 클래스 변수 값: 1
MyClass.increment_class_variable() # 출력: 클래스 변수 값: 2
```

정적 메서드 vs 클래스 메서드

특징	클래스 메서드(@classmethod)	정적 메서드(@staticmethod)
첫 번째 매개변수의 유무	cls (클래스를 나타냄)	없음
클래스 속성 접근 가능 여부	가능	불가능
주로 사용되는 목적	클래스 상태 변경, 대체 생성자	독립적인 유틸리티 기능 제공

클래스 실습 03

✓ [문제] Car 클래스를 만드세요

- 객체 생성 시 **차이름**, **배기량**, **생산년도**를 입력받고 **인스턴스 속성**으로 만들어 주세요.
- 차이름, 배기량, 생산년도는 직접 변경하지 못합니다.
- 차이름을 확인하는 함수와 변경하는 **함수**를 생성해보세요.
- 배기량에 따라 1000cc 보다 작으면 소형

  1000cc 이상 2000cc 이하 중형

  2000cc 보다 크면 대형을 출력하는 **인스턴스 함수**를 만드세요.
- 객체 생성 마다 등록된 차량 갯수를 기록하는 **클래스 속성**을 만들어 주세요.
- 총 등록된 차량 개수를 출력하는 **클래스 함수**를 만드세요.

상속

공통되는 내용 = 부모 클래스

클래스(자식)는 공통되는 내용을 부모클래스로부터 상속

class 부모클래스:

코드

class 자식클래스(부모클래스명):

코드

상속

✓ 부모 클래스

```
# 부모 클래스
class Animal:

    def __init__(self):
        self.hungry = 0

    def eat(self):
        self.hungry -= 10
        print('밥먹음', self.hungry)

    def walk(self):
        self.hungry += 10
        print('산책', self.hungry)
```

✓ 자식 클래스

```
# 자식 클래스
class Dog(Animal):

    def __init__(self):
        super().__init__()

    def sound(self):
        print('멍멍')


class Cat(Animal):

    def __init__(self):
        super().__init__()

    def sound(self):
        print('야옹')
```

✓ 클래스 및 메서드 호출

```
# 개
print('개----------')
dog = Dog()
dog.sound()

dog.walk()
dog.walk()

# 고양이
print('고양이----------')
cat = Cat()
cat.sound()

cat.walk()
```

상속 받은 기능 사용

상속 받은 기능 사용

✓ 0.0s

```
개----------
멍멍
산책 10
산책 20
고양이----------
야옹
산책 10
```

상속

✓ 부모 클래스

```
# 부모 클래스
class Animal:

    def __init__(self):
        self.hungry = 0

    def eat(self):
        self.hungry -= 10
        print('밥먹음', self.hungry)

    def walk(self):
        self.hungry += 10
        print('산책', self.hungry)
```

✓ 자식 클래스

```
# 자식 클래스

class Dog(Animal):

    def __init__(self):
        super().__init__()

    def sound(self):
        print('멍멍')

    def eat(self):
        super().eat()
        print('왈왈') # 추가
```

부모 클래스의 기능을 가져다 씀

✓ 클래스 및 메서드 호출

```
dog = Dog()
dog.eat()
```

✓ 0.0s

```
밥먹음 -10
왈왈
```

상속을 통한 정적메소드, 클래스 메소드 비교

```
class Animal:

    type = “동물"

    @staticmethod
    def getType1():
        return Animal.type

    @classmethod
    def getType2(cls):
        return cls.type

    def __init__(self):
        self.hungry = 0
```

```
class Dog(Animal):

    type = “강아지”

    def __init__(self):
        super().__init__()

    def sound(self):
        print(“멍멍”)
```

```
Dog.getType1()
>> 동물

Dog.getType2()
>> 강아지
```

상속을 통한 정적메소드, 클래스 메소드 비교

```
class Animal:

    type = “동물"

    @staticmethod
    def getType1():
        return Animal.type

    @classmethod
    def getType2(cls):
        return cls.type

    def __init__(self):
        self.hungry = 0
```

```
class Dog(Animal):

    type = “강아지"

    # 부모 클래스 상속
    def __init__(self):
        super().__init__()

    def sound(self):
        print(“멍멍”)
```

자식클래스가 부모 메서드 사용

```
Dog.getType1()
>> 동물

Dog.getType2()
>> 강아지
```

고정메서드 : 자식 클래스가 부모 클래스의 변수를 다룸

클래스메서드 : 클래스 레벨로 작업을 수행

cls매개변수를 통해 자식 클래스의 변수에 영향을 받음

클래스 실습 04

✓ [문제 1] Character 클래스를 만들어 주세요.

- Character 클래스의 Health 속성에 200을 할당해주세요.
- Character 클래스에 Move() 메서드를 추가하고 메서드 사용시 Health 가 -10 이 됩니다.
- Character 클래스에 Rest() 메서드를 추가하고 메서드 사용시 Health 가 + 10 됩니다.
- 현재 Health를 알수있는 checkHealth() 메서드를 추가해주세요

✓ [문제 2] Knight와 Healer 클래스를 만들어 주세요.

- Knight 와 Healer 클래스는 Charcter 클래스를 상속합니다.
- Knight 클래스는 Move() 사용시 Health 가 -5 더 소모됩니다.
- Knight 클래스는 Attack() 추가하고 실행시 '공격합니다'를 출력해주세요
- Healer 클래스는 Mana속성을 추가해주세요 (생성시 100)
- Healer 클래스는 heal(character) 메서드를 추가하고 메서드는 character 들을 매개변수로 받습니다.
- Healer 클래스는 heal(character) 메소드 실행시 Mana가 -10되고 전달받은 character 객체의 rest() 메소드를 실행합니다.
- Healer 클래스는 현재 마나속성을 확인할수있는 checkMana() 메서드를 추가해주세요

19. 예외처리

최희윤 강사

예외처리

✓ 예외처리란?

- 프로그램 처리 중 발생하는 '오류(예외)'를 처리하여 프로그램이 갑자기 중단되지 않고, 예상 가능한 방식으로 동작하도록 돕는 방법
- 발생하는 오류 예시)
  - TypeError: 잘못된 타입 연산
  - FileNotFoundError: 파일을 찾을 수 없을 때
- 예외처리 **기본 구조**: try-except 구문 (아래의 예제 코드는 모든 예외를 처리하는 코드임)

```
try:
    print(10/0) # 오류가 발생할 가능성이 있는 코드 작성
except:
    print('예외 오류 발생') # 오류가 발생 시 실행할 코드
```

특정 예외만 처리

✓ 특정 예외만 처리할 경우

- 각 예외를 명시적으로 처리 -> 가독성과 유지보수성이 좋아짐

```
try:
        x = int(input('숫자를 입력하세요: '))
        print(10/x)
except ZeroDivisionError:
        print('0으로 나눌 수 없습니다.')
except ValueError:
        print('유효한 숫자를 입력하세요.')
```

as 키워드 활용

✓ as 키워드 활용하여 별칭을 만들고 그 별칭을 통해 메시지를 출력하는 것을 살펴볼 수 있음

```
try:
        x = int(input('숫자를 입력하세요: '))
        print(10/x)
except ZeroDivisionError as e:
        print(e)  # 출력: division by zero
```

예외처리 else와 finally

- ✓ else: 예외가 발생하지 않았을 때 실행
- ✓ finally: 예외 발생 여부와 상관없이 항상 실행

```
try:
        x = int(input('숫자를 입력하세요: '))
        print(10/x)
except ZeroDivisionError:
        print('0으로 나눌 수 없습니다.')
else:
        print('성공적으로 실행되었습니다.')
finally:
        print('프로그램이 종료되었습니다.')
```

예외처리 else와 finally

✓ finally: 예외 발생 여부와 상관없이 항상 실행

- 특정 동작에 대해 반드시 뒤따라 오는 부분, 꼭 해야하는 구문을 작성할 때 try와 finally를
  활용하면 좋음
- 이는 우리가 데이터베이스에 Connection을 얻고 사용 뒤 반환해야 하는 것과 비슷
- 즉, finally 부분에 connection.close()하는 코드를 작성하면 중간에 어떤 네트워크 에러 및 로직상의 에러가 발생하더라도 connection을 반납하도록 코드를 작성할 수 있음

Exception을 활용한 모든 예외 잡기

✓ Exception 클래스: 모든 내자 예외의 기본 클래스

- 일반적으로 예외를 구체적으로 처리하는 게 좋지만, 모든 예외를 처리해야 할 경우 사용
- 예외 객체 e를 출력하기(예외 정보 출력) 때문에, 어떤 오류가 발생했는지 정확히 알 수 있음

```
try:
    x = int(input('숫자를 입력하세요: '))
    print(10/x)
except Exception as e:
    print(f"오류가 발생했습니다: {e}")
```

예외처리 실습 01

✓ [문제] 사용자로 부터 숫자 2개를 입력받아 split() 메서드를 활용하여 x, y 변수로 unpacking 합니다.

- 만약 사용자가 잘못된 값을 입력하더라도 프로그램이 중단되지 않고 '값을 잘못 입력하셨습니다.' 라는 메시지를 출력할 수 있게 예외처리 부분을 추가해서 완성해보세요.

```
x, y = input('숫자 두 개를
입력하세요.').split()
```

예외처리 실습 02

✓ [문제]

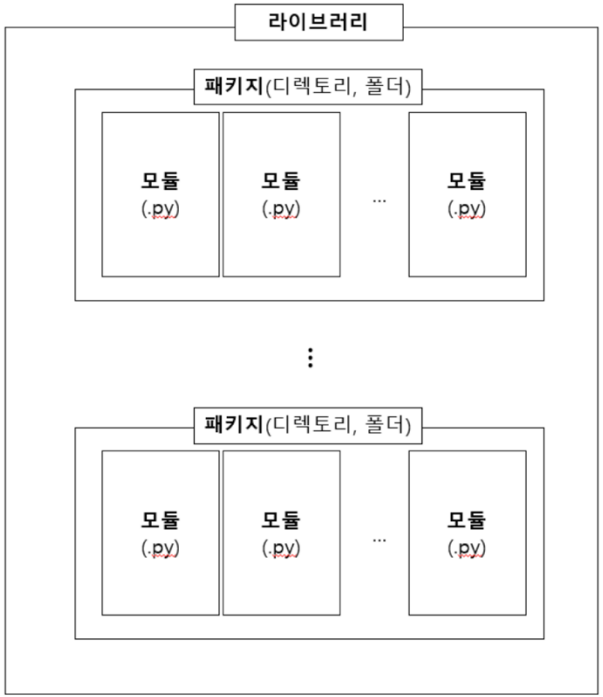
- 사용자로 부터 1회 값을 입력 받습니다.
- 이 때 정수 2개를 띄어쓰기로 구분하여 값을 받고, 이를 split() 메서드로 값을 나눈 뒤
  map() 함수를 활용하여 각각 int로 형변환 합니다.
- 형변환한 두 값은 unpacking해서 x, y라는 변수에 차례로 담습니다.
- 그리고 result라는 변수에 x / y의 연산 결과를 담습니다.
- 이 때 예외처리는 아래와 같이 작성합니다.
  - ValueError가 발생될 때는 '값을 잘못 입력하셨습니다.'를 출력
  - ZeroDivisionError(0으로 나눈 경우)는 '0으로 나눌 수 없습니다.'를 출력
  - Error가 발생되지 않은 경우 result의 값을 출력

20. 모듈과 패키지

최희윤 강사

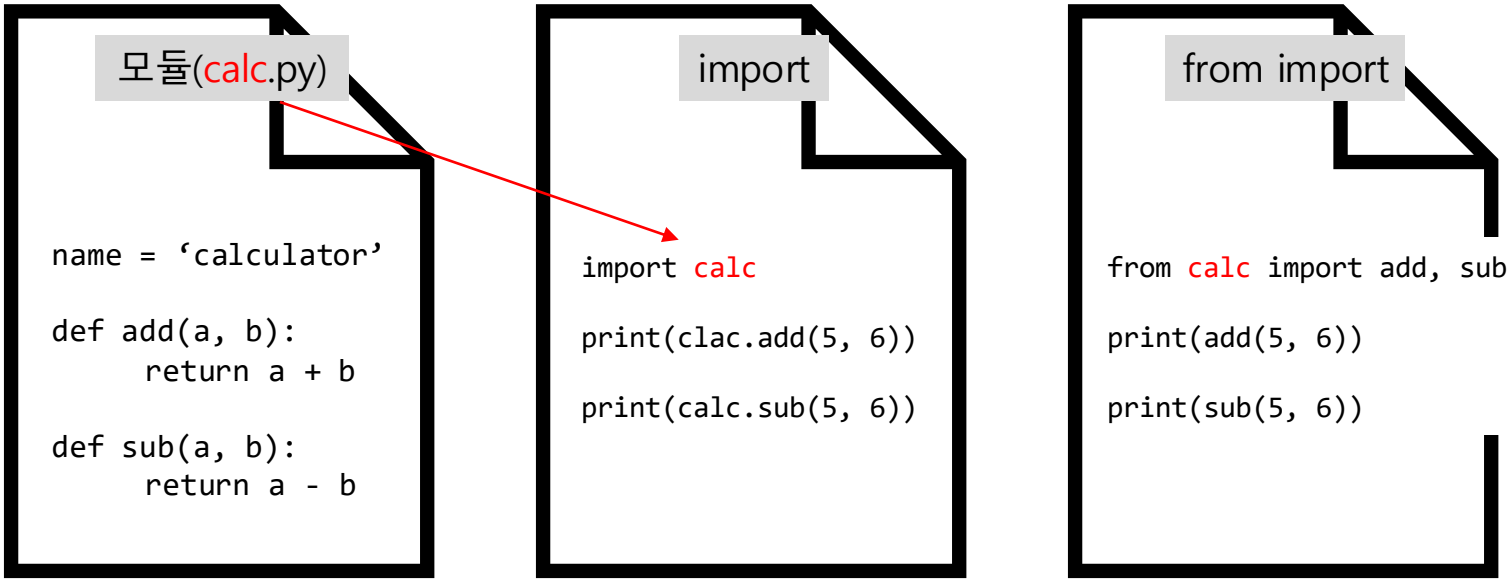
라이브러리, 패키지, 모듈의 관계

✓ 모듈이 모이면 패키지가 되고, 패키지가 모이면 라이브러리가 됨



모듈

✓ 변수, 함수, 클래스 등을 모아놓은 스크립트 파일



__name__ 특수변수

✓ __name__ (던더네임): 모듈의 이름을 저장해놓은 변수

- name의 값은 파이썬에서 알아서 정해줌
  - 만약 파이썬 파일을 직접 실행한다면, 그 파일의 name은 main으로 설정 됨
  - 파일을 다른곳에서 import해서 사용하면 name은 원래 모듈 이름으로 설정 됨
  - 즉, my_module.py 라는 파일을 직접 실행하면, name은 main으로 설정되고, run.py 라는 파일에서 my_module.py를 import해서 사용하면 name은 my_module이 됨
- python 파일이 직접 실행될 때와 다른 파일에서 임포트되어 사용될 대 동작 구분을 하기 위해 **if __name__ == '__main__':** 구문을 통해 특정 **코드가 테스트용 코드나 메인 스크립트 실행 코드일 때**, 파일이 임포트 되었을 경우 실행되지 않도록 함

__name__ 특수변수 실습

- ✓ calc.py 모듈을 작성 후 ipynb 파일에서 import 해보자
- ✓ area.py 모듈을 작성 후 ipynb 파일에서 import 해보자

패키지

✓ 패키지란?

- 여러가지 모듈을 모아놓은 것

✓ 패키지 생성해보기

- shapes 패키지 생성 및 구조 확인

shapes/

__init__.py

area.py

volume.py

18_패키지.pys

실습 파일 : shapes/18_패키지.ipynb, area2.py, volume.py

패키지 생성 및 import 실습

1. shapes/area.py 파일을 생성해보자
2. shapes/volume.py 파일을 생성해보자
3. 18_패키지.ipynb 파일에서 두 개의 스크립트 파일을 import 해보자

__init__ 파일 활용

✓ __init__ 파일이란?

- '이 폴더는 파이썬 패키지다' 라고 말해주는 파일
- 디렉토리 안에 init파일이 없으면 디렉토리가 패키지로 인식이 안 돼서 import할 수 없음
- 패키지를 초기화 할 때 사용되는 파일

- 처음으로 패키지나, 패키지 안에 있는 모듈 및 함수를 import하면, 가장 먼저 init파일에 있는 코드가 실행 됨

✓ shapes 디렉토리 안에 __init__파일 생성 전후의 shapes 패키지 import 결과를 비교 해보자

실습 파일 : shapes/18_패키지.ipynb, area2.py, volume.py

모듈과 패키지 찾는 경로

✓ 모듈과 패키지의 위치 알 수 있는 방법

```
import sys

print(sys.path)
```

✓ 또는 site-packages 폴더에 pip로 설치한 패키지가 들어감