함수

작성일시	@2022년 7월 20일 오전 10:50
내용	문법 파이썬 함수
주차	1주차
자료	

함수 기초

- Decomposition(분해)
  - 기능을 분해하고 재사용 가능하게 만들고
- Abstraction(추상화)
  - 복잡한 내용을 모르더라도 사용할 수 있도록
  - 재사용성과 가독성, 생산성

함수의 종류

- 내장 함수
  - 파이썬에 기본적으로 포함된 함수
- 외장 함수
  - import 문을 통해 사용하며, 외부 라이브러리에서 제공하는 함수
- 사용자 정의함수
  - 사용자가 직접 만드는 함수

함수 기본 구조

- 선언과 호출(Define & Call)
  - def <키워드>를 활용함
  - 들여쓰기를 통해 Function body(실행될 코드 블록)을 작성함
    - Docstring은 함수 body 앞에 선택적으로 작성 가능
      - 작성 시에는 반드시 첫 번째 문장에 문자열 ‘’’’’

    - 함수는 parameter를 넘겨줄 수 있음
    - 함수는 동작 후에 return을 통해 결괏값을 전달함
- 입력(Input)
- 문서화(Docstirng)
- 범위(Scope)
- 결과값(Output)

```
def function_name(parameter):
    # code block
    return returning_value
```

함수의 결과값(Output)

값에 따른 함수의 종류

- Void Function
    - 명시적인 return 값이 없는 경우, None을 반환하고 종료
- Value returning function
    - 함수 실행 후, return문을 통해 값 반환
    - return을 하게 되면, 값 반환 후 함수가 바로 종료
- **print() vs return()**
    - print를 사용하면 호출될 때마다 값이 출력됨(주로 테스트를 위해 사용)
        - 반환되는 값은 없음!
    - 데이터 처리를 위해서는 return 사용

두 개 이상의 값 반환

- return은 항상 하나의 값만을 반환

```
# wrong 코드

def minus_and_product(x, y):
    return x - y
    return x * y
```

```
y = minus_and_product(4, 5)
print(y) # -1
```

- 반환 값으로 튜플 사용(리스트와 같은 컨테이너도 가능!)

```
def minus_and_product(x, y):
    return x - y, x * y

y = minus_and_product(4, 5)
print(y) # (-1, 20)
print(type(y)) # <class 'tuple'>
```

함수의 입력(Input)

Parameter와 Argument

- Parameter: 함수를 정의할 때, 함수 내부에서 사용되는 변수
- Argument: 함수를 호출할 때, 넣어주는 값

```
def function(ham): # parameter: ham
    return ham

function('spam') # argument: spam
    # spam
```

Argument

- 함수 호출 시 함수의 parameter를 통해 전달되는 값
- 소괄호 안에 할당 func_name(argument)
  - 필수 argument: 반드시 전달되어야 하는 argument
  - 선택 argument: 값을 전달하지 않아도 되는 경우는 기본 값이 전달
- Positional Arguments
  - 위치에 따라 전달
- Keyword Arguments
  - 직접 변수의 이름으로 특정 Argument 전달 가능

    - Keyword Argument 다음에 Positional Argument **활용 불가능!**
- Default Argument Values
    - 기본 값을 지정하여 함수 호출 시 argument 값을 설정하지 않아도 되도록 함
- 가변인자(*args)
    - 여러 개의 Positional Argument를 하나의 필수 parameter로 받아서 사용
    - 몇 개의 positional argument를 받을지 모르는 함수를 정의할 때 유용
    - 패킹 / 언패킹
        - 패킹
            - 여러 개의 데이터를 묶어서 변수에 할당

```
numbers = (1, 2, 3, 4, 5) # 패킹
print(numbers) # (1, 2, 3, 4, 5)
```

        - 언패킹
            - 시퀀스 속의 요소들을 여러 개의 변수에 나누어 할당

```
numbers = (1, 2, 3, 4, 5)
a, b, c, d, e = numbers # 언패킹
print(a, b, c, d, e) # 1 2 3 4 5
```

            - 언패킹 시 변수의 개수와 할당하고자 하는 요소의 갯수가 동일해야 함

```
numbers = (1, 2, 3, 4, 5) # 패킹
a, b, c, d, e, f = numbers # 언패킹
# ValueError: not enough values to unpack (expected 6, got 5)
```

            - 언패킹 시 왼쪽의 변수에 asterisk(*)를 붙이면, 할당하고 남은 요소들을 리스트에 담을 수 있음

```
numbers(1, 2, 3, 4, 5) # 패킹

a, b, *rest = numbers # 1, 2를 제외한 나머지를 rest에 대입
print(a, b, rest) # 1 2 [3, 4, 5]

a, *rest, e = numbers # 1, 5를 제외한 나머지를 rest에 대입
print(rest) # [2, 3, 4]
```

- Asterisk (*)와 가변 인자
  - *는 시퀀스 언패킹 연산자라고도 불리며, 말 그대로 시퀀스를 풀어 헤치는 연산자
    - 주로 **튜플**이나 **리스트** 언패킹
    - *를 활용하여 가변 인자를 만들 수 있음

```
def func(*args)
      print(args)
      print(type(args))

func(1, 2, 3, 'a', 'b')
'''
(1, 2, 3, 'a', 'b')
<class 'tuple'>
'''
```

- 가변 키워드 인자(**kwargs)
  - 몇 개의 키워드 인자를 받을지 모르는 함수 정의할 때 유용
  - **kwargs는 **딕셔너리**로 묶여 처리 되며, parameter에 **를 붙여 표현

```
def family(**kwargs):
    for key, value in kwargs.items():
        print(key, ':', value)

family(father = '아부지', mother = '어무니', baby = '아기') # 변수처럼 작성하기(문자열 x)
'''
father : 아부지
mother : 어무니
baby : 아기
'''
```