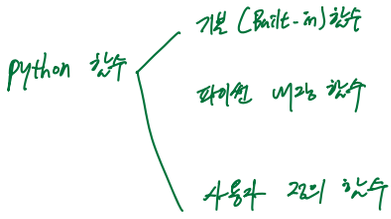
# 9. 함수

## 9.1 함수의 개념 배우기

### 9.1.1 함수란?

- 하나의 특정 작업을 수행하는 코드들의 집합이다.
- 작업에 필요한 매개변수(입력값)을 전달받아 작업 결과를 반환한다.

### 9.1.2 Python 함수의 분류




#### 기본(Built-in) 함수

- 파이썬에서 인터프리터에서 기본으로 내장된 함수
- 아무 설정없이 바로 사용 가능

#### 패키지 내장 함수

- 특정 패키지나 모듈에 포함된 함수
- import 명령어를 사용해 해당 패키지를 불러온 후 사용 가능
- math 모듈에 포함된 함수

#### 사용자 정의 함수

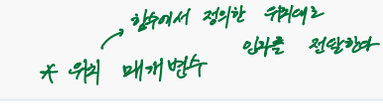
- 사용자가 필요에 따라 특정 기능의 함수를 직접 작성

### 9.1.3 함수를 정의하는 방법

#### 위치 매개변수(일반 매개변수)

- 가장 기본적으로 인자값을 전달하는 방식. 함수에서 정의한 위치대로 인자값을 할당한다.

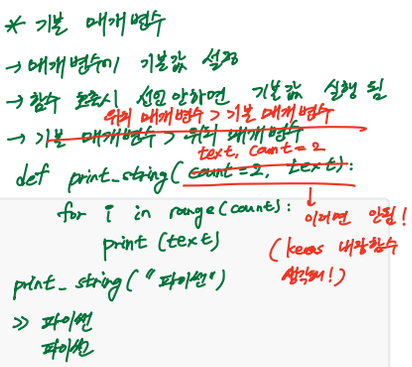
```
def print_string(text, count):
    for i in range(count):
        print(text)
print_string("파이썬입니다", 3)
```




```
def print_string(count, text):
    for i in range(count):
        print(text)
print_string(3, "Hello")
>> Hello
   Hello
   Hello
```

## 기본 매개변수 (위치 > 기본)

- 함수를 정의할 때 매개변수에 default값을 설정하는 것.
- 기본값이 설정된 매개변수를 기본 매개변수라고 한다.
- 함수 호출시, 값을 입력하지 않으면 기본값이 실행된다.
- 항상 위치 매개변수 이후에 기본 매개변수가 와야 한다.




```
def print_string(text, count = 3):
    for i in range(count):
        print(text)
print_string("파이썬입니다.")
```

```
# 잘못된 예시, 기본 매개변수를 위치 매개변수보다 앞에 넣었기 때문에
def print_string(count = 3, text):
    for i in range(count):
        print(text)
print_string
```

## 가변 매개변수 (위치 > 가변)

- 인자값이 몇 개인지 모르는 상황에서 사용
- 변수 이름 앞에 *를 붙이면 튜플, **를 붙이면 딕셔너리로 받는다.
- 위치 매개변수 뒤에 와야 함.
- 함수당 1개의 가변 매개변수 사용가능.

```
#예제1 튜플을 이용한 가변매개변수
def print_string(count, *text):
    for i in range(count):
        for t in text:
            print(t)
print_string(1, "파이썬","수업","입니다.")

'''
파이썬
수업
입니다.
'''
```

```
# 예제2 딕셔너리를 이용한 가변매개변수
def person_info(**person):
    for key in person.keys():
        print("{}:{}".format(key, person[key]))

person_info(name = "홍길동",age = 30, address = "인천광역시 미추홀구")

'''
name:홍길동
age:30
address:인천광역시 미추홀구
'''
```

```
# 잘못된 예시, 위치매개변수 뒤에 가변매개변수가 와야한다.
def print_string(*text, count):
    for i in range(count):
        for t in text:
            print(t)
print_string("파이썬","수업","입니다.",3 )

'''
TypeError: print_string() missing 1 required keyword-only argument: 'count'
'''
```

## 키워드 매개변수

- 함수 호출 시, 직접 매개변수를 지정해서 찾아가는 방식.

### 기본매개변수와 가변매개변수를 같이 사용하는 경우

1. **가변 매개변수가 기본 매개변수보다 먼저 나오는 경우** (가변 > 기본)

```
#변수 선언 (count=1)
def print_string(*text, count=2):
    for i in range(count):
        for t in text:
            print(t)
print_string("파이썬","프로그래밍","수업입니다.", count=1)

'''
파이썬
프로그래밍
수업입니다.
'''

---
#변수 선언 안함 (1)
def print_string(*text, count=2):
    for i in range(count):
        for t in text:
            print(t)
```

```
print_string("파이썬","프로그래밍","수업입니다.", 1)

'''
파이썬
프로그래밍
수업입니다.
1
파이썬
프로그래밍
수업입니다.
1
'''
```

이렇게 count라는 키워드를 지정해주지 않으면 튜플로 인식해버린다.

2. **기본 매개변수가 가변 매개변수보다 먼저 나오는 경우** (기본 > 가변)

```
# 기본 매개변수값 입력없이 함수 호출한 경우
def print_string(count=2, *text):
    for i in range(count):
        for t in text:
            print(t)
print_string("파이썬","프로그래밍","수업입니다.")

'''
TypeError: 'str' object cannot be interpreted as an integer
'''
```

기본 매개변수가 먼저 나온 경우에는 함수 호출시 키워드매개변수로 선언해주어야 한다.

```
def print_string(count=2, *text):
    for i in range(count):
        for t in text:
            print(t)
print_string(count=1, "파이썬","프로그래밍","수업입니다.")

'''
SyntaxError: positional argument follows keyword argument
'''

def print_string(count=2, *text):
    for i in range(count):
        for t in text:
            print(t)
print_string(1, "파이썬","프로그래밍","수업입니다.")

'''
파이썬
프로그래밍
수업입니다.
'''
```

위의 사례는 기본 매개변수를 선언해주었지만, 변수명을 썼느냐 안썼느냐로 결과가 다르게 나옴.

이런 경우 변수명을 써주지 말아야 한다.

#### 기본 매개변수 중 필요한 값만 입력하기, 순서 상관없이 입력하기

```
def sum(a,b = 10, c = 100):
    return a+2*b+3*c

print(sum(1,20,100)) #(1,20,100), 341
print(sum(1,c=20,b=100)) #(1,100,20), 261
print(sum(1,20)) #(1,20,100), 341
print(sum(1,c=20)) #(1,10,20), 81
```

### 9.1.4 함수의 필요성

1. 프로그램이 크고 복잡할 때 함수를 만들어 단순한 부분으로 분해할 수 있음
2. 한번 작성된 함수를 여러번 호출하여 사용할 수 있음
3. 한번 작성된 함수를 다른 프로그래미에서 사용하기도 쉬워짐
4. 코드의 안정성이 좋아지고 오류를 수정하기 쉬워진다.

## 9.2 함수 응용하기

### 9.2.1 둘 이상의 값 반환하기

- 2개 이상의 값을 콤마로 구분해서 지정해 반환한다.

#### 둘 이상의 값을 튜플로 반환

```
def add_sub(a,b):
    return a+b, a-b
x,y = add_sub(10,20)
print("10과 20의 합은 {}이고, 차는 {}입니다.".format(x,y))
print(type(add_sub(10,20))) # tuple
```

#### 둘 이상의 값을 리스트로 직접 반환

```
def twice_list(objlist):
    result = [2*i for i in objlist]
    return result
list1 = [1,2,3,4,5]
list2 = twice_list(list1)
print(list2) #[2,4,6,8,10]
print(type(twice_list)) # <class 'list'>
```

## 9.2.2 재귀함수란?

- 함수 안에서 함수 자기 자신을 호출하는 방식
- 언제까지 자신을 호출할지 종료조건을 만들어줘야함
- 재귀함수는 같은 항이 여러번 중복되어서 계산되므로 비효율적이다.

### 팩토리얼 using 반복문, 재귀함수

#### 반복문으로 팩토리얼 구하기

```
def factorial(n):
    result = 1
    for i in range(1,n+1):
        result *= i
    return result
print("2!:{}".format(factorial(2))) # 2
print("3!:{}".format(factorial(3))) # 6
```

#### 재귀함수로 팩토리얼 구하기

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
print("2!:{}".format(factorial(2))) # 2
print("3!:{}".format(factorial(3))) # 6
```

### 피보나치 수열

재귀함수로 피보나치 수열 구하기

```
def fibonacci(n):
    if n==1:
        return 1
    elif n==2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print("fibonacci(1):{}".format(fibonacci(1))) #1
print("fibonacci(2):{}".format(fibonacci(2))) #1
print("fibonacci(3):{}".format(fibonacci(3))) #2
print("fibonacci(4):{}".format(fibonacci(4))) #3
print("fibonacci(5):{}".format(fibonacci(5))) #5
print("fibonacci(6):{}".format(fibonacci(6))) #8
```

#### 딕셔너리로 피보나치 수열 구하기

한번 계산한 값은 딕셔너리에 저장해 다시 계산하지 않음

```
dict1 = {
    1:1,
    2:1
}
def fibonacci(n):
    if n in dict1:
        return dict1[n]
    else:
        result = fibonacci(n-1) + fibonacci(n-2)
        dict1[n] = result
        return result

print("fibonacci(40) : {}".format(fibonacci(40))) #102334155
```