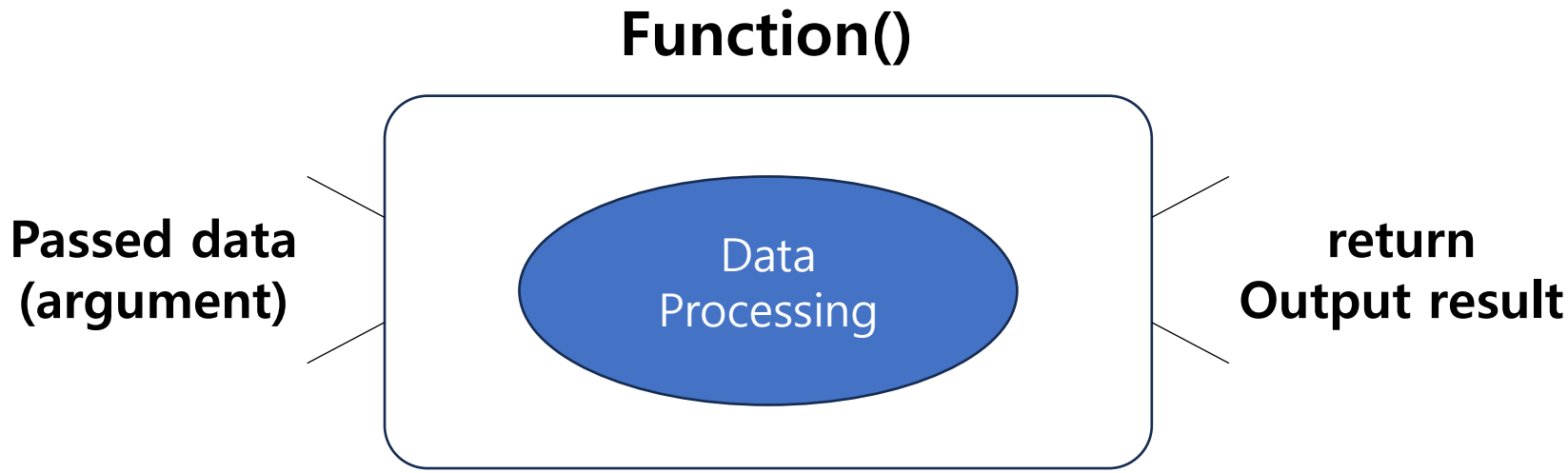
1. 함수의 기본

- 함수란
- 함수의 장점과 단점
- 한수의 사용법
- 매개 변수와 인수
- 함수 결과 반환_return

함수란

함수(Function)

- 특정한 작업을 수행하는 코드의 집합
- 데이터(인수)를 전달받아 처리하고, 그 결과를 반환 함



함수의 장점과 단점

➢ 함수의 장점

- 코드의 중복을 최소화할 수 있다.
- 코드를 재사용할 수 있다.
- 전체 프로그램을 모듈로 나눌 수 있어 개발 과정이 체계적이며 유지보수가 쉬워진다.

  * 모듈(module): 프로그램을 구성하는 구성 요소, 관련된 데이터와 함수를 하나로 묶은 단위를 의미함

➢ 함수의 단점

- 함수 호출할 때 마다 운영체제(OS)가 함수에 사용되는 지역 변수들의 준비와 인수(argument)전달 등을 처리해야 하므로 부담이 발생한다.
- 따라서 너무 작은 단위의 함수를 구성하여 자주 호출하는 경우 성능에 문제가 발생할 수 있다.

함수의 사용법

- 함수의 생성은 **def**라는 키워드를 입력하고 함수의 이름을 작성한 후 **():**으로 선언
- 함수의 시작과 끝을 들여쓰기만으로 구분하며 시작과 끝을 명시해 주지 않음

함수 선언

```
def 함수명( ):
    코드
```

예: 함수 선언

```
def hello( ):
    print('Hello world!' )
```

함수 호출

```
함수명( )
```

예: 함수 호출

```
hello( )
```

예: 실행 결과 'Hello world!'

매개 변수와 인수(1)

- 매개 변수(parameter) : 함수에 입력으로 전달된 값을 받는 변수
- 인수(argument): 함수를 호출할 때 전달하는 입력 값을 의미

함수 선언

```
def add(x, y):
    sum=x + y

    return sum
```

매개변수: x, y

함수 호출

```
print(add(3, 4))
```

인수: 3, 4

주의점

- 매개변수와 인수가 여러 개면 쉼표로 구분
- 매개변수와 인수의 개수는 같아야 함(다르면 에러 발생)

매개 변수와 인수(2)

- 함수의 호출과 인수 전달

함수 선언

```
def add(x, y):
    print(x + y)
```

함수 호출

```
add(1, 2)
add(1.0, 2.0)
add('abc ' , 'def')
add([1,2,3], [4,5,6])
add((1,2,3), (4,5,6))
```

실행 결과

```
3
3.0
abcdef
[1, 2, 3, 4, 5, 6]
(1, 2, 3, 4, 5, 6)
```

함수 결과 반환_return(1)

➢ **return 없는 함수**

```
def hi():
    print('안녕')
```

```
hi()
```

안녕

```
x = hi()
print(x)
```

안녕
None

X = hi() → hi함수가 실행되어 ' 안녕'이 출력되지만, return이 없기 때문에 x에는 None 반환됨

➢ **return 반환 값**

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

```
x = add(3,4)
print(x)
```

7

X = add(3,4) → add함수가 실행되어 3과 4가 더해진 값 7이 X에 반환 됨

함수 결과 반환_return(2)

➢ **return 반환값1, 반환값2....**

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

```
x= add_sub(5,3)
print(x)
```

```
(8, 2)
```

함수의 결과값이 튜플로 반환

```
x, y = add_sub(5,3)
print('x=', x, 'y=',y)
```

```
x= 8 y= 2
```

x, y 각 변수에 8, 2가 각각 반환 (unpacking 됨)

➢ **Return 반환값 생략**

return: 값 반환 or 함수 실행 종료

```
def chek_negative(num):
    if num < 0:
        return
    return num
```

```
chek_negative(3)
```

```
3
```

```
chek_negative(-3)
```

chek_negative(3)은 3을 반환하지만,
chek_negative(-3)은 num이 0보다 작으므로 return이 실행되어 아무것도 반환하지 않고 함수를 빠져나옴

2. 매개 변수와 인수

- 매개변수 정의와 인수 전달
- 위치 인수
- 기본값 위치 인수
- 가변적 위치 인수
- 키워드 인수
- 가변적 키워드 인수
- 주의: 매개변수 혼합 - 초기값 매개변수의 위치
- 주의: 매개변수 혼합 - 위치 인수와 가변적 위치 인수 혼합
- 주의: 매개변수 혼합 – 위치 인수와 가변적 키워드 인수 혼합
- 주의: 매개변수 혼합 – 가변적 위치인수와 가변적 키워드 인수의 혼합

매개변수(parameter)와 인수

➢ **매개변수 형식 정의와 인수 전달**

구분	형식	설명
함수 선언 (매개변수형식 정의)	def func(arg)	고정 매개변수(위치매개변수) 선언
	def func(arg=value)	고정 매개변수(위치매개변수)에 기본값 설정
	def func(*varargs)	가변적 위치 매개변수 선언 (가변적 매개변수: 매개변수 개수가 정해지지 않음)
	def func(**kwargs)	가변적 키워드 매개변수 선언
	def func(arg,*other)	고정 매개변수와 가변적 위치 매개변수 사용 선언 (arg 먼저 지정하고 *other이 선언, 순서 중요)
	def func(*arg,**kwargs)	가변적 위치 매개변수와 가변적 키워드 매개변수 사용 (*arg가 먼저 오고 **kwargs가 나중에 옴, 순서 중요)
	def func(*, name=value)	* 표시 이후에는 카워드 매개변수만 사용하도록 선언하며, 기본값 설정
함수 호출 (인수 정의)	func(arg1, arg2...)	고정 위치 인수 설정, 위치 인수로 전달
	func(arg=value)	키워드 인수, 이름으로 매개변수에 연계
	func(*iterable)	iterable(예: 문자,리스트, 튜플 등) 객체로 인수를 전달, interable 객체의 각 개체들이 위치인수로 연계됨
	func(**dict)	키워드:값 으로 표현된 항목들을 dict 자료형 인수로 전달되면, 인수에 포함된 키워드를 사용하여 키워드 매개변수에 전달 됨

매개변수(parameter)와 인수 (1) :위치 인수

➢ **위치 인수(Positional Argument): 함수에 인수를 순서대로 넣는 방식**

```
def pos_args(x,y,z):
    return 3*x + 2*y +z
```

```
w=pos_args(1,2,3)
print(w)
```

```
10
```

pos_args(1,2,3) 함수 호출로 인해 위치 순서대로 x=1, y=2, z=3이 각각 지정 되고

w변수에는 3*1 + 2*2 +3 의 수식 결과 10이 반환됨

매개변수(parameter)와 인수 (2) :위치 인수

- **func(*iterable) : iterable(dP:문자, 리스트, 튜플 등) 객체로 인수를 전달i, iterable 객체의 각 개체들이 위치 인수로 연계됨**

```
def pos_arg(x,y,z):
    return 3*x + 2+y +z
```

```
data=[1,2,3]
pos_arg(*data)
```

```
10
```

post_arg(*data)로 함수를 호출할 때 data=[1,2,3]이라는 리스트 요소들이 함수에

개별 인자로 풀어져서(언팩킹) 전달 됨. 즉, x=1, y=2, z=3이 전달 됨

매개변수(parameter)와 인수 (3) : 기본값 위치 인수

➢ **기본값(default value)을 지정하는 위치 인수**

- 기본값을 가지며, 함수 호출시 생략 가능함

```
def volume(width=1, length=1, height=1):
    return width * length * height
```

```
print(volume())
print(volume(10))
print(volume(10,20))
print(volume(10,20,30))
```

인수로 값을 전달하지 않았으므로 매개변수는 모두 기본값(width=1, length=1, height=1)

width에만 10을 전달, 나머지는 기본값(width=10, length=1, height=1)

width에는 10, length에는 20 전달, height는 기본값 (width=10, length=20, height=1)

width에는 10, length에는 20, height=30 전달(width=10, length=20, height=30)

```
1
10
200
6000
```

매개변수(parameter)와 인수 (4) : 가변적 위치 인수

➢ **가변적 위치 인수(variable positional argument)**

- 인수의 개수가 정해지지 않은 가변적 인수일 때, 매개변수 이름 앞에 *를 표시
- 함수 내부에서 튜플 형태로 처리됨

```
def var_postargs(*args):
    print(type(args),args)
```

```
var_postargs(1)
var_postargs(1,2)
var_postargs(1,2,3)
```

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

매개변수(parameter)와 인수 (5) : 키워드 인수

➢ **키워드 인수(Keyword argument)**

- 함수 호출에서 전달되는 인수의 이름(키워드)을 명시
- 매개변수의 순서를 맞추지 않아도 키워드에 해당값을 전달 가능

```
def BMI(weight, height):
    print('몸무게:', weight)
    print('키:', height)
    print('BMI:',weight /((height/100)**2))
```

✓ 위치인수: weight, height 값을 순서대로 입력

```
BMI(50, 160)
```

```
몸무게: 50
키: 160
BMI: 19.531249999999996
```

✓ 키워드 인수: weight, height 값을 순서 상관 없음

```
BMI(weight=50, height=160)
```

```
몸무게: 50
키: 160
BMI: 19.531249999999996
```

```
BMI(height=160,weight=50)
```

```
몸무게: 50
키: 160
BMI: 19.531249999999996
```

매개변수(parameter)와 인수 (6) : 키워드 인수

➢ **키워드 인수: 딕셔너리 자료형으로 키워드 인수를 전달**

- 딕셔너리 앞에 **를 붙여서 매개변수에 전달 (딕셔너리 언패킹 사용)

```
def dict_args(x,y,z):
    return 3*x + 2*y +z
```

```
w=dict_args(**{'x':1, 'y':2, 'z':3})
print(w)
```

```
10
```

```
data={
    'x':1,
    'y':2,
    'z':3
}
w=dict_args(**data)
print(w)
```

```
10
```

매개변수(parameter)와 인수 (7) : 키워드 인수

➢ **키워드 인수: * 표시 다음에 키워드 인수만을 사용하는 함수**

- 함수 매개변수 형식 정의에서 *표시가 있으면, 그 다음은 키워드 인수만 사용해야 함

```
def keyword_only_args(x,*, y,z):
    return 3*x + 2*y +z
```

← ***표시 뒤의 y, z는 키워드 인수를 사용해야 함**

✓ **y, z를 키워드 인수로 전달**

```
w= keyword_only_args(1,y=2, z=3)
print(w)
```

```
10
```

```
w= keyword_only_args(1,**{'y':2, 'z':3})
print(w)
```

```
10
```

✓ **y, z를 위치인수로 전달하면 오류가 남**

```
w= keyword_only_args(1,2,3)
print(w)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[108], line 1
----> 1 w= keyword_only_args(1,2,3)
      2 print(w)

TypeError: keyword_only_args() takes 1 positional argument but 3 were given
```

매개변수(parameter)와 인수 (8) : 가변적 키워드 인수

➢ **가변적 키워드 인수:**

- 인수의 개수가 정해지지 않은 가변적 키워드 인수일 때, 매개변수 이름 앞에 **를 표시
- 함수 내에서는 딕셔너리 자료형으로 처리됨

```
def varkw_args(**kwargs):
    print(type(kwargs))
    print(kwargs)
```

**kwargs: 가변적 키워드 인수 지정

```
varkw_args(name='홍길동')
```

```
<class 'dict'>
{'name': '홍길동'}
```

```
varkw_args(name='홍길동', age=20,  add='서울')
```

```
<class 'dict'>
{'name': '홍길동', 'age': 20, 'add': '서울'}
```

```
data={
    'name':'홍길동',
    'age':20,
    'add':'서울'
}
varkw_args(**data)
```

**data: 딕셔너리 자료형으로 인수 전달

```
<class 'dict'>
{'name': '홍길동', 'age': 20, 'add': '서울'}
```

주의: 매개변수 혼합 - 초기값 매개변수의 위치

➢ 초기값 매개변수의 위치

- 초기값이 지정된 매개변수 다음에는 초기값이 없는 매개 변수가 올 수 없음

```
def p_info(name, add='비공개', age):
    print('이름:',name)
    print('나이:',age)
    print('주소:',add)
```

```
  Cell In[136], line 1
    def p_info(name, add='비공개', age):
                                   ^
SyntaxError: non-default argument follows default argument
```

p_info(name, add='비공개',age) → 초기값이 지정된 매개변수 뒤에 초기값 없는 매개 변수 올 수 없음 (오류)

```
def p_info(name, age, add='비공개'):
    print('이름:',name)
    print('나이:',age)
    print('주소:',add)
```

```
p_info('홍길동',30)
```

```
이름: 홍길동
나이: 30
주소: 비공개
```

p_info(name, age, add='비공개') → name, age 매개변수는 반드시 값을 전달해야하고, add는 값이 제공되지 않으면 기본값 '비공개'가 사용됨

주의: 매개변수 혼합 - 위치 인수와 가변적 위치 인수 혼합

➢ 위치 인수와 가변적 위치 인수 함께 사용

- 위치 인수 개수 만큼 매핑시킨 후, 나머지는 튜플로 전달
- 가변 매개 변수 뒤에는 일반 매개 변수 사용할 수 없다. (*args가 고정매개변수 a보다 먼저 오면 안됨)
- 가변 매개 변수는 하나만 사용할 수 있음

```
def var_postargs(a, *args):
    print('a=',a,'args=',args)
```

```
var_postargs(1)
var_postargs(1,2)
var_postargs(1,2,3,4,5)
```

```
a= 1 args= ()
a= 1 args= (2,)
a= 1 args= (2, 3, 4, 5)
```

주의: 매개변수 혼합 - 위치 인수와 키워드 인수 혼합

➢ 위치 인수와 키워드 인수 함께 사용

- 위치 인수, 키워드 인수 순서대로 배치되어야 함

```
def pos_kw(name, **kwargs):
    print('이름:', name)
    print('kwargs:', kwargs)
```

Name: 위치 인수로, 필수 값 제공
****kwargs: 키워드 인수로 여러 개의 키-쌍으로 값 제공**

```
pos_kw('홍길동')
```

```
이름: 홍길동
kwargs: {}
```

```
pos_kw('홍길동', age=20, addr='서울')
```

```
이름: 홍길동
kwargs: {'age': 20, 'addr': '서울'}
```

```
data={
    'age':20,
    'addr':'서울'
}
pos_kw('홍길동', **data)
```

****data: 딕셔너리 자료로 인수 전달**

```
이름: 홍길동
kwargs: {'age': 20, 'addr': '서울'}
```

주의: 매개변수 혼합 - 가변적 위치인수와 가변적 키워드 인수의 혼합

➢ **가변적 위치 인수(*args)와 가변적 키워드 인수(**kwargs) 함께 사용**

- 가변적 위치 인수(*args) 먼저 지정한 다음에 가변적 키워드 인수(**kwargs)가 지정되어야 함

```
def var_pos_key(*args, **kwargs):
    print('args=', args)
    print('kwargs=', kwargs)
```

```
var_pos_key(1,x=3)
```

```
args= (1,)
kwargs= {'x': 3}
```

```
var_pos_key(1,2,3,x=3,y=4)
```

```
args= (1, 2, 3)
kwargs= {'x': 3, 'y': 4}
```

3. 함수 활용

- 네임스페이스와 유효범위(Scope)
- 파이썬 내장함수
- 람다함수 (lambda function)
- 1급 함수(Fist class function)
- 데코레이터 함수 (Function Decorator)

네임스페이스와 유효범위(Scope) (1)

➢ **네임스페이스(namespace):**이름(변수, 함수, 클래스 등)이 저장되는 공간을 의미

Name Space	설명
지역(local)	• 함수 또는 클래스 메서드 내에서 정의된 이름. • 함수나 메서드가 호출될 때 생성되고, 호출이 종료되면 소멸. • locals()함수로 확인 가능
전역(global)	• 프로그램의 전체 범위에서 접근할 수 있는 변수 • 함수 외부에서 정의되며, 프로그램의 어느 곳에서나 접근 가능 • globals()함수로 확인 가능
내장(built-in)	• 파이썬 인터프리터에 의해 미리 정의된 변수. • dir(__builtins__)로 확인 가능 • 예: print(), abs(), len(), max() 등의 함수나 True, False, None 등

파이썬에서는 이름을 찾을 때, **지역 네임스페이스(Local) → 전역 네임스페이스(Global) → 내장 네임스페이스(Built-in)** 순서로 탐색함

네임스페이스와 유효범위(Scope) (2)

```
def func1():
    a=10
    print(f'func1()-> a={a}')

def func2():
    print(f'func2()-> a={a}')

a=20
func1()
func2()
```

지역변수

전역변수

func1함수 호출

func2함수 호출

```
func1()-> a=10
func2()-> a=20
```

- func1( ) → func1함수 내 지역변수 a를 찾아 출력함 (a=10)
- func2( ) → func2함수 내 지역변수 a를 먼저 찾는데 없으니 전역변수 a를 찾아 출력함 (a=20)

네임스페이스와 유효범위(Scope) (3)

Name Scole 지정선언	의미
Global	함수 내부에서 전역변수를 수정하고자 할 때, 해당 변수명 앞에 global 키워드를 사용
nonlocal	중첩된 함수 내부에서 부모 함수의 변수를 수정하고자 할 때, 해당 변수명 앞에 nonlocal 키워드를 사용

```
x=1
y=3
z=5
def func():
    x=10
    global y
    y=30
    z=50
    print(f'func() -> x={x},y={y},z={z}')
    def sub_func():
        nonlocal z
        z=100
        print(f'sub_func() -> x={x},y={y},z={z}')
    sub_func()
    print(f'func() -> x={x},y={y},z={z}')
func()
print(f'main -> x={x},y={y},z={z}')
```

x, y, z는전역변수

X 지역 변수

y전역 변수

z 지역 변수

z 부모함수 func()의 지역 변수

```
func() -> x=10,y=30,z=50
sub_func() -> x=10,y=30,z=100
func() -> x=10,y=30,z=100
main -> x=1,y=30,z=5
```

1 2 func( ) 함수 호출

X → 함수 내 x값 할당 (지역변수 **x=10**)
Y → global y 인해 전역변수 y값이 새로 할당 됨 (전역변수 **y=30**)
z → 함수 내 z값 할당 (지역변수 **z=50**)

3 4 sub_func( ) 함수 호출

X → 함수 내 x가 없어 부모함수에서 찾음 (지역변수 **x=10**)
Y → 함수 내 y가 없어 부모함수에서 찾음(전역변수 **y=30**)
Z → nonlocal z 로 인해 부모함수의 z값을 새로 할당 (부모지역변수 **z=100**)

5 X → (func)함수 내 x값 (지역변수 **x=10**)
Y → 함수 내 y 값(전역변수 **y=30**)
Z → sub_func( )실행으로 Z=100으로 수정 (부모지역변수 **z=100**)

6 X → 전역변수 **X=1**
Y → 전역변수 **Y=30** (FUNC()실행으로 Y=30수정 됨
Z → 전역변수 **Z=5**

파이썬 내장함수 (1)

분류	함수	설명
자료형 변수 생성/변환	int()	문자열 또는 숫자를 정수로 변환합니다.
	float()	문자열 또는 숫자를 부동 소수점 수로 변환합니다.
	str()	주어진 객체를 문자열로 변환합니다.
	bool()	주어진 객체의 불린(논리적) 값을 반환합니다.
	list()	반복 가능한 객체로부터 리스트를 생성합니다.
	tuple()	반복 가능한 객체로부터 튜플을 생성합니다.
	set()	반복 가능한 객체로부터 세트(집합)를 생성합니다.
	dict()	키-값 쌍으로 딕셔너리를 생성합니다.
수치 데이터 연산	abs()	주어진 숫자의 절대값을 반환합니다.
	sum()	반복 가능한 객체 내 모든 요소의 합을 반환합니다.
	min()	반복 가능한 객체 내 최소값을 반환합니다.
	max()	반복 가능한 객체 내 최대값을 반환합니다.
	round()	숫자를 주어진 자릿수까지 반올림합니다.

파이썬 내장함수(2)

분류	함수	설명
문자/문자열 연산	len()	객체의 길이(요소의 개수)를 반환합니다.
	chr()	주어진 유니코드 코드 포인트에 해당하는 문자를 반환합니다.
	ord()	주어진 문자의 유니코드 코드 포인트를 반환합니다.
반복 가능 자료형 연산	range()	지정된 범위의 숫자 시퀀스를 생성합니다.
	enumerate()	반복 가능한 객체의 인덱스와 값을 쌍으로 반환합니다.
	zip()	여러 반복 가능한 객체들을 짝지어 튜플의 시퀀스로 반환합니다.
	filter()	함수와 반복 가능한 객체를 받아, 함수 조건에 맞는 요소만 걸러내어 반환합니다.
	map()	함수와 반복 가능한 객체를 받아, 각 요소에 함수를 적용한 결과를 반환합니다.
	sorted()	반복 가능한 객체를 정렬하여 리스트로 반환
객체	type()	객체의 타입을 반환합니다.
	id()	객체의 고유 식별자를 반환합니다.
	isinstance()	객체가 주어진 클래스의 인스턴스인지 여부를 반환합니다.

파이썬 내장함수(3)

분류	함수	설명
파일/디렉토리 관리	open()	파일을 열고 파일 객체를 반환합니다.
	os 모듈	파일 시스템을 다루는 여러 함수를 제공합니다 (os 모듈 자체는 내장 함수는 아니지만, 파일 및 디렉토리 관리를 위해 중요합니다).
실행 가능 문자열	eval()	문자열로 표현된 파이썬 식을 평가하고 결과를 반환합니다.
	exec()	문자열로 표현된 파이썬 코드를 실행합니다.
입력/출력	print()	주어진 객체를 표준 출력에 출력합니다.
	input()	사용자 입력을 문자열로 받습니다.
함수	help()	내장 도움말 시스템을 사용하여 함수나 모듈, 클래스 등에 대한 정보를 제공합니다.
	lambda()	이름이 없는 익명 함수 생성

파이썬 내장함수 – map()

➢ map( ): 함수와 반복 가능한 객체를 받아, 각 요소에 함수를 적용한 결과를 반환

기본구조 map(함수, 반복가능한 객체)

```
a=['1','2','3','4']
int_a=list(map(int, a))
print(int_a)
```

리스트(a)에 int함수를 각각 적용

```
[1, 2, 3, 4]
```

```
numbers = [1, 2, 3, 4, 5]
squared=list(map(lambda x:x*x, numbers))
print(squared)
```

```
[1, 4, 9, 16, 25]
```

리스트(numbers)에
익명함수(lambda)에 적용

```
def square(x):
    return x * x
numbers = [1, 2, 3, 4, 5]
squared = map(square, numbers)
print(list(squared))
```

리스트(numbers)에
square함수를 각각 적용

```
[1, 4, 9, 16, 25]
```

파이썬 내장함수 – filter()

➢ filter( ): 함수와 반복 가능한 객체를 받아, 함수 조건에 맞는 요소만 걸러내어 반환

(함수의 반환 값이 True 일 때만 해당 요소를 가져옴)

기본구조 filter(함수, 반복가능한 객체)

```
def func(x):
    return x>0 and x<5

a=[0,1,2,3,4,5,6,7,8,9,10]
result=list(filter(func,a))
print(result)
```

```
[1, 2, 3, 4]
```

✔ list(filter(func,a) → 리스트(a)에 func함수를 각각 적용해 결과가 True인 것만 해당요소 가져옴

```
a=[0,1,2,3,4,5,6,7,8,9,10]
result=list(filter(lambda x: x>0 and x<5,a))
print(result)
```

```
[1, 2, 3, 4]
```

✔ list(filter(lambda x:x>0 and x<5,a) → 리스트(a)에 lambda함수를 각각 적용해 결과가 True인 것만 해당요소 가져옴

파이썬 내장함수 – zip(), zip(*)

- zip( ): 함수는 여러 반복 가능한(iterable) 객체들의 요소를 튜플로 묶는 데 사용
- zip(*반복가능한 객체) : zip으로 결합된 객체나 이터레이터 앞에 *붙이면 분리 가능(언팩킹)

기본구조 **zip(반복가능한 개체 2개이상)**

```
x=[1,2,3]
y=['a','b','c']
z=[100,200,300]

zip_xyz=list(zip(x,y,z))
print('zip_xyz=',zip_xyz)
```

```
zip_xyz= [(1, 'a', 100), (2, 'b', 200), (3, 'c', 300)]
```

```
d1,d2,d3=zip(*zip_xyz)
print('d1=',d1)
print('d2=',d2)
print('d3=',d3)
```

```
d1= (1, 2, 3)
d2= ('a', 'b', 'c')
d3= (100, 200, 300)
```

list(zip(x, y, z)): x, y, z의 각 요소를 순서대로 묶어서 새로운 튜플 리스트 zip_xyz를 생성

d1, d2, d3 = zip(*zip_xyz)는 zip_xyz의 각 튜플을 언패킹하여 zip() 함수에 다시 전달한 후 원래의 세 리스트 로 분리되어, d1,d2,d3에 각각 할당 됨

파이썬 내장함수 – sorted() (1)

➢ sorted( ): 반복 가능한(iterable) 객체의 모든 요소를 정렬하여 새로운 리스트로 반환
원본 데이터를 변경하지 않고, 정렬된 새로운 리스트를 생성

기본구조 sorted(반복가능한 개체, key=None, reverse=False)

```
L=[5,7,2,1,8]
print('sorted(L):',sorted(L))
print('sorted(L,reverse=True):',sorted(L,reverse=True))
```

← L 오름차순 정렬

← L 내림차순 정렬 (reverse=True)

```
sorted(L): [1, 2, 5, 7, 8]
sorted(L,reverse=True): [8, 7, 5, 2, 1]
```

```
TL=[('park',47,60.5),('lee',23,80.2),('kim',30,70.2)]
print('sorted(TL):',sorted(TL))
print('sorted(TL, key=lambda x:x[1]):',sorted(TL, key=lambda x:x[1]))
print('sorted(TL, key=lambda x:x[2]):',sorted(TL, key=lambda x:x[2]))
```

← TL의 첫번째 요소로 정렬

← TL의 두번째 요소로 정렬 (key=lambda x:x[1])

← TL의 두번째 요소로 정렬 (key=lambda x:x[2])

```
sorted(TL): [('kim', 30, 70.2), ('lee', 23, 80.2), ('park', 47, 60.5)]
sorted(TL, key=lambda x:x[1]): [('lee', 23, 80.2), ('kim', 30, 70.2), ('park', 47, 60.5)]
sorted(TL, key=lambda x:x[2]): [('park', 47, 60.5), ('kim', 30, 70.2), ('lee', 23, 80.2)]
```

파이썬 내장함수 – sorted() (2)

➢ sorted()함수를 사용하여 딕셔너리의 키(key)와 값(value)을 다양한 기준으로 정렬하는 하기

```
my_dict={
    'k1':7,
    'k3':4,
    'k2':5
}
print('sorted(my_dict):',sorted(my_dict))❶
print('my_dict.items():',my_dict.items())❷
print('sorted(my_dict.items()):',sorted(my_dict.items()))❸
print('sorted(my_dict.items(),key=lambda x:x[1]):',sorted(my_dict.items(),key=lambda x:x[1]))❹
```

```
sorted(my_dict): ['k1', 'k2', 'k3']
my_dict.items(): dict_items([('k1', 7), ('k3', 4), ('k2', 5)])
sorted(my_dict.items()): [('k1', 7), ('k2', 5), ('k3', 4)]
sorted(my_dict.items(),key=lambda x:x[1]): [('k3', 4), ('k2', 5), ('k1', 7)]
```

❶ my_dict의 key기준으로 오름차순 정렬
❷ my_dict.items()는 딕셔너리의 키-값 쌍을 튜플로 가지는 리스트로 반환
❸ sorted(my_dict.items())는 딕셔너리 아이템(키-값 쌍)을 키를 기준으로 오름차순으로 정렬
❹ sorted(dict.items(), key=lambda x: x[1])는 딕셔너리 값의 오름차순으로 정렬
lambda x: x[1]는 키-값 쌍 튜플에서 값을 기준으로 정렬 기준을 설정

람다 함수(lambda function)

- lambda 함수: 이름이 없는 함수로, 한 줄짜리 코드로 작성
  일회성으로 사용되거나 다른 함수의 인자로 전달될 때 주로 사용 (ex: map(), filter())

기본구조 lambda 매개변수들: 식

```
double= lambda x: x*2
triple= lambda x: x*3

print('dobule(5):', double(5))
print('triple(5):', triple(5))
print('(lambda x:x*4)(5):',(lambda x:x*4)(5))
```

```
dobule(5): 10
triple(5): 15
(lambda x:x*4)(5): 20
```

```
files=['1.txt','2.jpg','4.docx','5.jpg']
jpg_files=list(filter(lambda x: '.jpg' in x,files))
print(jpg_files)
```

```
['2.jpg', '5.jpg']
```

1급 함수(first class function) (1)

- **1급 함수:** 함수가 다른 객체와 동일하게 취급되며 다음 특징이 있음
  - 변수 할당: 함수를 변수에 할당
  - 함수 인자로 전달: 함수를 다른 함수의 인자로 전달
  - 함수에서 반환: 함수가 다른 함수를 반환
  - 데이터 구조에 저장: 함수를 리스트, 딕셔너리 등 데이터 구조에 저장 가능

✓ **함수를 변수에 할당**

```
def great(name):
    return 'Hello, '+name
sayHello=great
sayHello('kim')
```

```
'Hello, kim'
```

1급 함수(first class function) (2)

✓ 함수를 다른 함수의 인자로 전달

```
def square(n):
    return n*n

def cubic(n):
    return n*n*n

def apply_func(L,func):
    L_result=[]
    for i in L:
        L_result.append(func(i))
    return L_result

L=[1,2,3,4,5]
print('L=',L)
print('apply_func(L,square)=',apply_func(L,square))
print('apply_func(L,cubic)=',apply_func(L,cubic))
```

```
L= [1, 2, 3, 4, 5]
apply_func(L,square)= [1, 4, 9, 16, 25]
apply_func(L,cubic)= [1, 8, 27, 64, 125]
```

✓ 함수를 리스트에 저장

```
def square(n):
    return n*n

def cubic(n):
    return n*n*n

func_L=[square, cubic]
L=[1,2,3,4,5]
for f in func_L:
    L_result=[]
    for i in L:
        L_result.append(f(i))
    print(L_result)
```

```
[1, 4, 9, 16, 25]
[1, 8, 27, 64, 125]
```

데코레이터 함수(Function Decorator) (1)

➢ **데코레이터 함수(Function Decorator):**

- 기존 함수의 동작을 변경하거나 확장하는데 시용
- 기존함수를 변경하지 않고도 추가적인 기능을 적용할 수 있음
- 함수 수식자는 로깅, 인증 체크, 성능 측정 등 다양한 상황에서 유용하게 사용

기본구조

```
def decorator(func):
    def wrapper():
        # 여기서 어떤 작업을 수행
        return func()
    return wrapper

@decorator
def my_function():
    print("Hello, World!")
```

decorator : 데코레이터 함수, 다른 함수(func)를 인자로 받음

wrapper: 내부함수로, 다른함수(func)의 호출을 감싸는 역할, 이 함수에서 추가적인 작업을 수행할 수 있음

@decorator: @기호와 함께 특정 함수를 수식함

my_function()함수를 호출하면 실제로는 'decorator(my_function)이 호출되어 'wrapper'함수가 실행됨

데코레이터 함수(Function Decorator) (2)

```python
import time

def timer_decorator(func):
    def wrapper():
        start = time.time()
        result = func()
        end = time.time()
        print(f"Function took {end - start} seconds to run.")
        return result
    return wrapper

@timer_decorator
def my_function():
    print("Function is running...")

my_function()
```

```
Function is running...
Function took 0.0 seconds to run.
```

my_function()은 timer_decorator에 의해 수식되어, 함수 실행 시간을 계산하고 출력하는 기능이 추가 됨

4. 재귀함수와 제너레이터

- 재귀 함수
- 제너레이터

재귀 함수 (1)

➢ 재귀함수

- 자기 자신을 호출하는 함수
- 재귀함수에 종료 조건이 없으면, 함수는 무한히 자기 자신을 호출하여 "최대 재귀 깊이 초과(maximum recursion depth exceeded)" 오류를 발생시킴

```
def hello():
    print('Hello word!')
    hello()

hello()
```

종료 조건이 없어 오류가 남

```
Hello word!
Hello word!
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[56], line 5
                    중간 생략
--> 506     return os.getpid() == self._master_pid

RecursionError: maximum recursion depth exceeded while calling a Python object
```

```
def hello(n):
    if n==0:                    ← 종료 조건 만듦
        return
    print('Hello word!')
    n -= 1                      ← n을 1감소
    hello(n)                    ← hello함수 호출

hello(5)
```

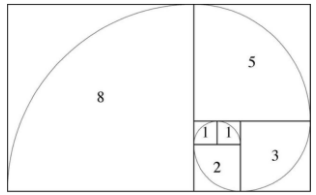
```
Hello word!
Hello word!
Hello word!
Hello word!
Hello word!
```

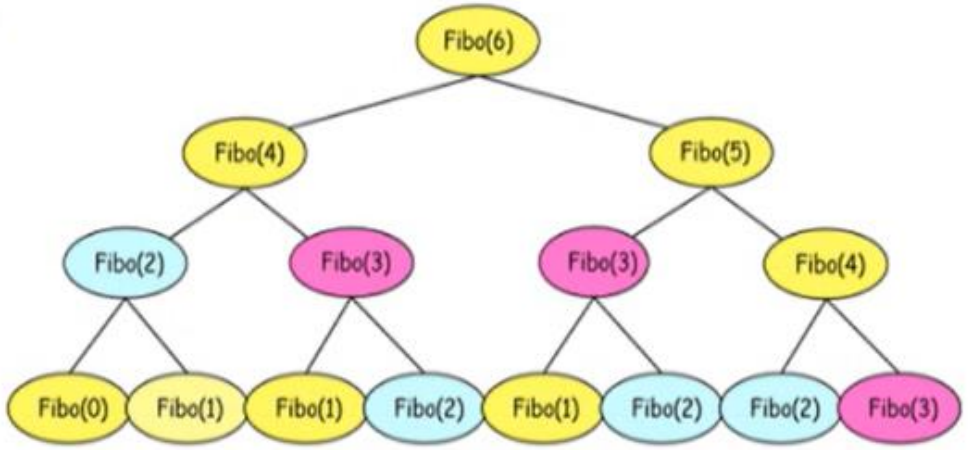
재귀 함수 (2)

➢ 재귀함수 예: 피보나치 수열 계산

- 피보나치 수열: 수열의 시작 두 숫자는 일반적으로 0과 1, 그 다음부터는 앞선 두 숫자의 합으로 구성
  예:0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- 재귀함수 피보나치 함수는 n이 커질수록 매우 비효율적이고 같은 계산을 반복적으로 수행
  → 동적 프로그래밍을 사용하면 이미 계산된 값을 저장하여 재사용함으로 속도를 향상 시킬 수 있음

- F(0) = 0, F(1) = 1
- F(n) = F(n-1) + F(n-2)






```
import time
start=time.time()
def f(n):
    if n <=0:
        return 0
    elif n==1:
        return 1
    else:
        return f(n-1) + f(n-2)
print('f(40):', f(40))
print('실행시간:', time.time()-start)
```

피보나치 재귀함수는 n이 40만 되어도
30초이상의 실행시간이 걸림
(일반적으로 재귀함수는 35 넘기 힘듬)

```
f(40): 102334155
실행시간: 30.449120044708252
```

재귀 함수 (3)

➢ 재귀함수와 동적프로그래밍(메모이제이션)사용한 피보나치 수열

동적 프로그래밍(Dynamic Programming, DP):

- 복잡한 문제를 여러 개의 간단한 하위 문제로 나누어 푸는 알고리즘
- 각 하위 문제를 한 번만 계산하고, 저장하여 재사용하는 방식으로 속도 향상

```
import time
start = time.time()

memo={}
def dynFibo(n):
    if n in memo:
        return memo[n]
    elif n<=1:
        memo[n]=n
        return n
    else:
        fibo_n=dynFibo(n-1)+dynFibo(n-2)
        memo[n]=fibo_n
        return fibo_n

print(dynFibo(100))
print('실행시간: ', time.time()-start)
```

```
354224848179261915075
실행시간:  0.0
```

제너레이터 (1)

➢ **제너레이터** (Generator)

- 제너레이터는 데이터를 순차적으로 생성하여 전달
- 제너레이터는 전체 항목을 한 번에 메모리에 적재하지 않고, 반복 가능한 상황에서 그 순간에 필요한 값을 하나씩 생성('yield') 함, 대규모 데이터셋을 다룰 때 메모리를 효율적으로 사용할 수 있음
- 제너레이터 함수는 return 대신 yield를 사용하여 (중간)결과값을 전달

```
def my_range(n):
    i=0
    while i<n:
        yield i
        i+=1
itr=my_range(3)
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
```

yield I → 현재의 i값을 전달하고, 함수를 일시 중지 함. next함를 통해 다음 값이 요청될 때 여기서부터 실행 재개 됨

제터레이터 객체 생성 (0~2까지 숫자 생성 제너레이터)

next(itr) 함수를 호출할 때마다 다음 숫자를 순차적으로 생성 (0,1,2)

next(itr) 함수를 호출할 때 생성할 값이 더 이상 없을 때 StopIteration 예외 발생 함

```
0
1
2
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[84], line 10
```

제너레이터 (2)

- **for와 제너레이터**

  for문은 반복할 때마다 __next__를 호출하므로 yield에서 발생시킨 값을 가져오고
  함수 끝까지 도달하여 StopIteration 예외가 발생하면 반복을 끝냄

```
def my_range(n):
    i=0
    while i<n:
        yield i
        i+=1
for i in my_range(3):
    print(i)
```

```
0
1
2
```

제너레이터 (3)

- **코루틴** (Coroutine)
  - 코루틴은 제너레이터의 일반화된 형태로, 두 개의 함수나 루틴 간에 데이터를 주고받을 수 있는 방법을 제공
  - 실행 중지 및 재개: yield 표현식을 사용하여 실행을 중지하고, 다시 그 지점부터 재개
  - 양방향 통신: 코루틴은 send() 메소드를 통해 외부에서 값을 받고, yield를 통해 값을 반환

- **코루틴의 주요 메소드**
  - send(value): 코루틴에 값을 보냄. 코루틴이 yield 표현식에서 멈춰 있는 경우, send()로 보낸 value가 yield에 의해 반환되고 실행이 재개
  - close(): 코루틴을 종료. 이후 코루틴에 값을 보내려고 하면 StopIteration 예외가 발생
  - throw(type[, value[, traceback]]): 코루틴 내부에 예외를 던짐

  - 제너레이터: next함수를 반복 호출하여 값을 얻어내는 방식
  - 코루틴: next함수를 한 번만 호출한 뒤 send로 값을 주고 받는 방식

제너레이터 (4)

➢ 코루틴 사용 예

```
def simple_coroutine():
    while True:
        x = yield
        print(f'Coroutine received: {x}')

# 코루틴 생성 및 시작
coro = simple_coroutine()
next(coro) # 코루틴을 시작하기 위해 첫 번째 `yield`까지 실행합니다.

# 코루틴에 데이터 보내기
coro.send(10)
coro.send(20)

# 코루틴 종료
coro.close()
```

```
Coroutine received: 10
Coroutine received: 20
```

코루틴은 시작하기 전에 반드시 next(coro) 또는 coro.send(None)으로 초기화해야 함, 코루틴 yield에서 중지

코루틴에 10을 보냄, 현재 코루틴 x=yield에서 오른쪽 yield에서 중지 됐음으로 왼쪽 x에서부터 재시작. 즉 x에 10 대입되고 다음 명령 계속 수행하다 다시 yield에서 멈춤

제너레이터 (5)

```
def coro_sum():
    try:
        total=0
        while True:
            x=yield
            total +=x
    except RuntimeError as e:
        print(e)
        yield total

co=coro_sum()           #코루틴 객체생성
next(co)                #코루틴을 시작하기 위해 첫 번째 'yield'까지 실행

for i in range(1,11):   #코루틴에 1~10까지 값을 전달
    co.send(i)

print(co.throw(RuntimeError, '예외로 코루틴 끝내기')) #co.Throw(예외이름,에러메시지): 코루틴 안에 예외 발생
```

```
예외로 코루틴 끝내기
55
```