5-1장. 함수

파이썬

Contents

❖ 목차

0. 학습목표

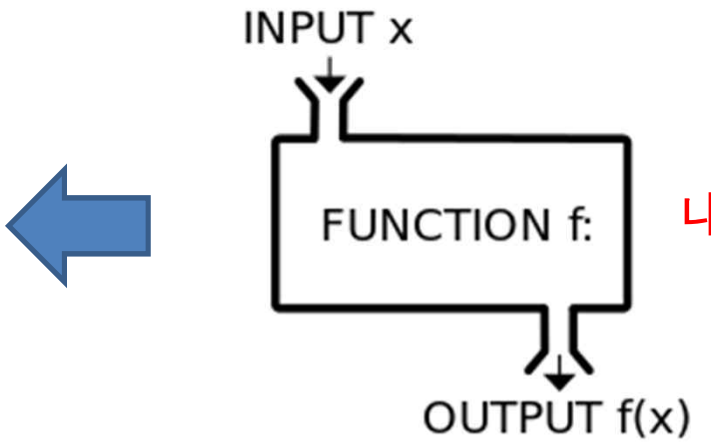
- 함수를 사용할 수 있다.
- 함수를 사용하는 방법을 설명할 수 있다.
- 튜플을 정의하고, 활용할 수 있다.

1. 함수 용어 정리

❖ 기본 용어들

- **함수 호출**
  - 함수를 사용하는 것
- **인수(argument) ->매개변수**
  - 함수 호출 시 괄호 내부에 넣는 여러 자료(전달받는 변수를 매개변수라함)
- **리턴값**
  - 함수 호출하여 최종적으로 나오는 결과

print()
len()
str()
int()
range()
.format()



내부 기능을 설계하는 일

1. 함수 용어 정리

❖ 내장 함수(Built-in Function)와 외장 함수(Standard Library)

❖ 내장 함수(Built-in Function)

- 일반적으로 많이 사용하는 함수, 파이썬에서 바로 사용

❖ 외장 함수(Standard Library)

- 파이썬에서 함께 제공되는 함수, import 후 사용

```
>>> help(함수명)          #내장함수 사용법에 대한 도움말 함수

>>> help(__builtins__)    #내장함수 종류 보기
```

1. 함수 용어 정리

❖ 내장 함수(Built-in Function)와 외장 함수(Standard Library)

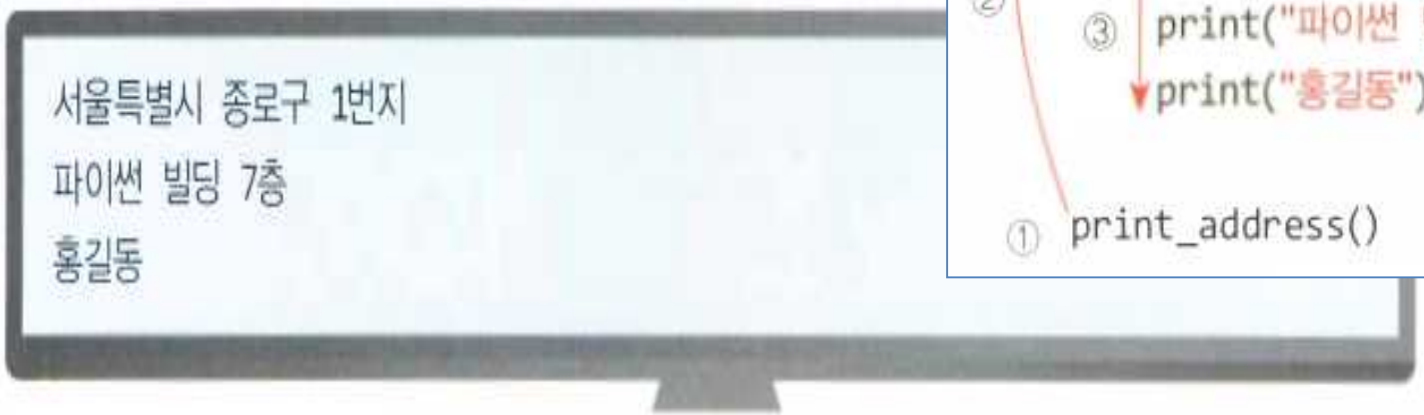
내장함수 목록				
abs()	delattr()	hash()	memoryview()	*set()*
all()	*dict()*	*help()*	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	*input()*	oct()	staticmethod()
bool()	*eval()*	*int()*	open()	*str()*
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	*float()*	iter()	*print()*	*tuple()*
callable()	format()	*len()*	property()	*type()*
chr()	frozenset()	*list()*	*range()*	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	*map()*	reversed()	__import__()
complex()	hasattr()	max()	*round()*	

1. 함수 용어 정리

- 정의
  - 정의(Definition)란, 어떤 이름을 가진 코드가 구체적으로 어떻게 동작하는지를 "구체적으로 기술"하는 것(특별한 기능을 수행하는 코드의 묶음)
- 파이썬에서는 함수나 메소드를 정의할 때 definition(정의)를 줄인 키워드인 **def**를 사용

- 실습 1 (def 키워드를 이용한 함수 정의)

```
def <함수 이름>(<매개변수>, <매개변수>, …):
    <문장>
```




```
def print_address():
    print("서울특별시 종로구 1번지")
    print("파이썬 빌딩 7층")
    print("홍길동")

print_address()
```

1. 함수 용어 정리

- 정의
  - 정의(Definition)란, 어떤 이름을 가진 코드가 구체적으로 어떻게 동작하는지를 "구체적으로 기술"하는 것
- 파이썬에서는 함수나 메소드를 정의할 때 definition(정의)를 줄인 키워드인 **def**를 사용
- 실습 1 (def 키워드를 이용한 함수 정의)

```
>>> def say_hello():
        print("어서오세요. 파이썬 세계에 오신 것을 환영합니다.")
        print("지금은 5장, 함수 World 입니다.")
>>> say_hello():
어서오세요. 파이썬 세계에 오신 것을 환영합니다.
지금은 5장, 함수 World 입니다.
```

>>> 함수명()

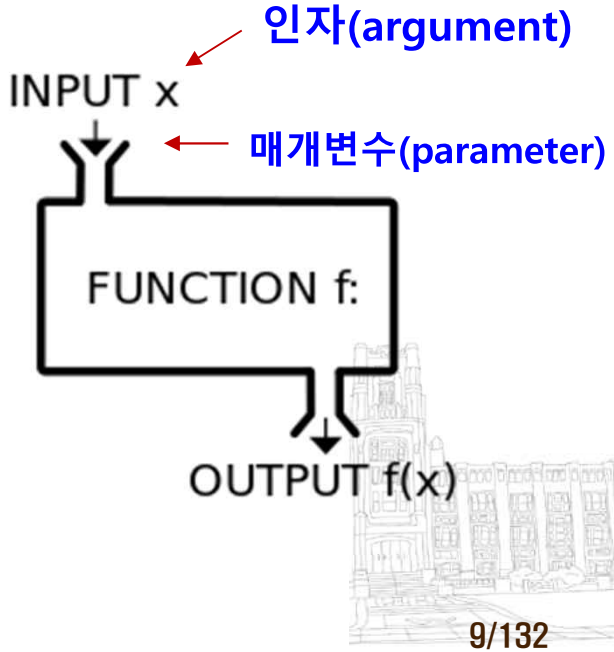
함수에 저장된 작업 수행

1. 함수 용어 정리

❖ 매개변수를 사용하여 입력값을 함수안으로 전달하기

```
>>> def triple(x):      # x를 매개변수라고 한다. 3의 값이 x변수에 대입된다.
        print(x * x * x)
>>> triple(3)          # 3을 인자값이라고 한다.
27
```

```
>>> def multiple(x, y):   # 인자값 2개 – 매개변수 2개
        print(x * y)
>>> multiple(7, 9):
63
```



1. 함수 용어 정리

❖ 함수의 결과 값을 변수에 넣으려고 할 때 발생하는 일

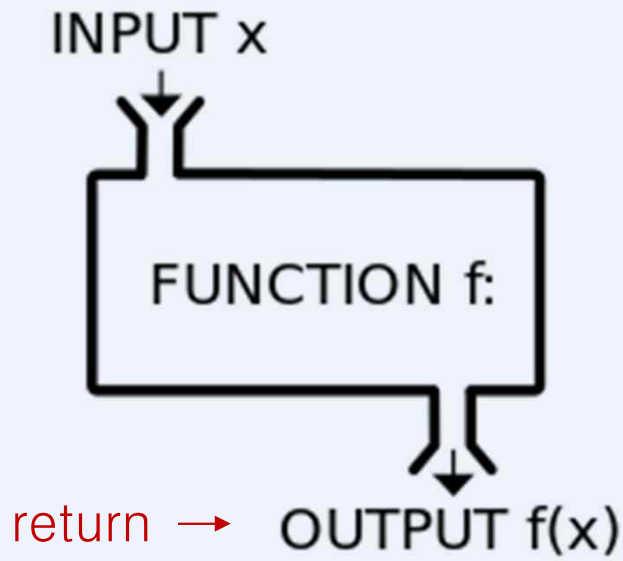
```
>>> def multiply(a, b, c):
        print(a*b*c)
>>> d = multiply(3,4,5)
60
>>> d
>>>
```

❖ print로 출력하였으나 정작 d에는 아무것도 입력되지 않음

1. 함수 용어 정리

❖ return을 이용하여 변수에 함수 결과 값 입력하기

```
>>> def multiply(a,b,c):
        print(a*b*c)
        return a*b*c
>>> d = multiply(3,4,5)
60
>>> d
60
>>>
```



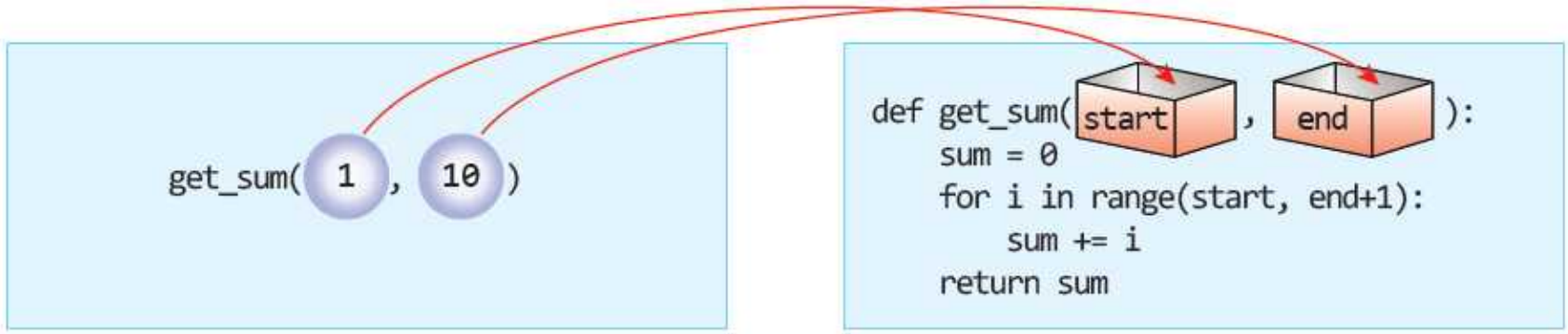
- return은 함수가 처리한 내용을 다시 함수 바깥으로 꺼내줌

1. width 변수에 인자가 두 개인 함수를 실행하여 두 숫자의 곱을 입력하여라.
2. width 변수에 (a+b)*c를 입력하는 함수 trepezium(a,b,c)를 만들고 실행하여라.

1. 함수 용어 정리

❖ 함수로 여러 개의 값을 전달하기

```
>>> def get_sum(start, end):
        sum = 0
        for i in range(start, end+1):
            sum += i
        return sum


>>> print(get_sum(1, 10))
```



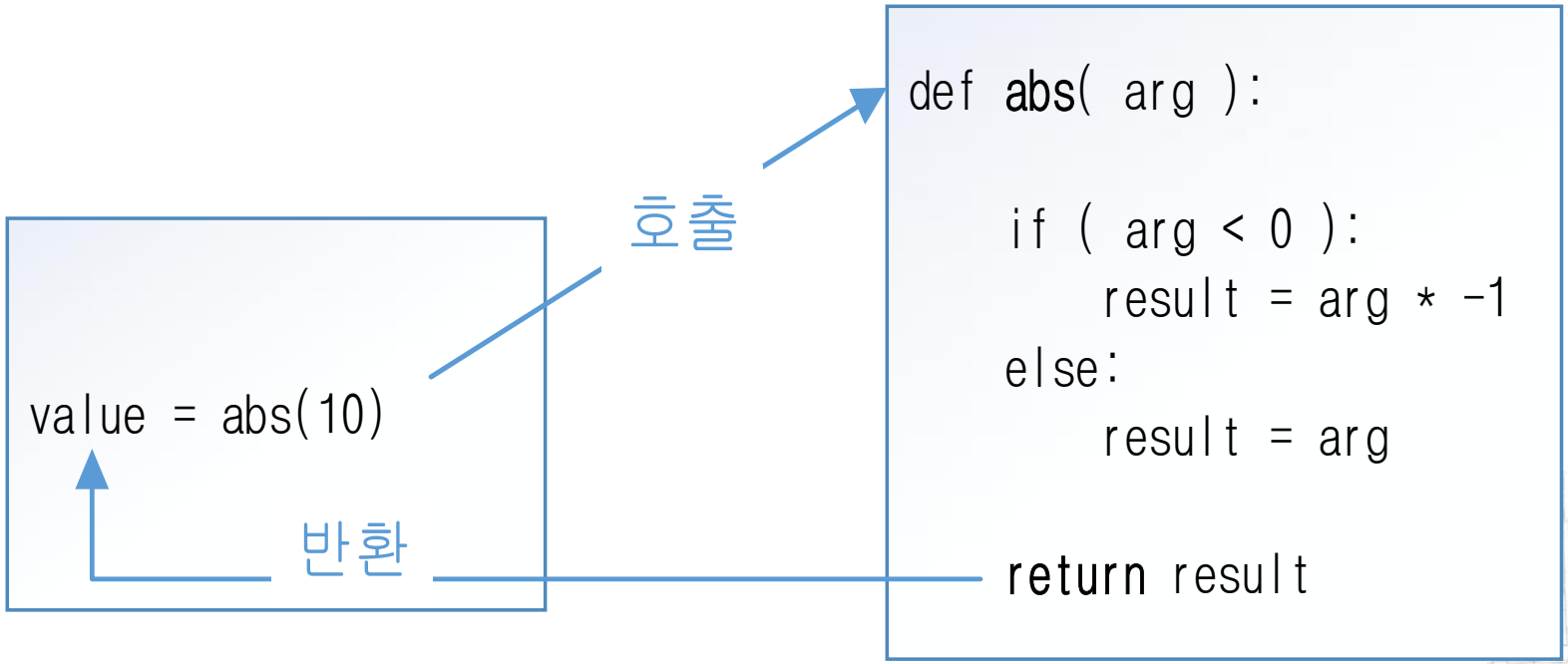
1. 함수 용어 정리

❖ **호출(Call)**

- 모든 함수는 이름을 갖고 있으며, 이 이름을 불러주면 파이썬은 그 이름 아래 정의되어 있는 코드를 실행.

❖ **반환(Return)**

- 함수가 자신의 코드를 실행하고 나면 결과가 나오는데, 그 결과를 자신의 이름을 부른 코드에게 돌려줌.



2. 함수 정의하기

- 함수는 **def** 키워드를 이용해서 코드블록에 이름을 붙인 형태

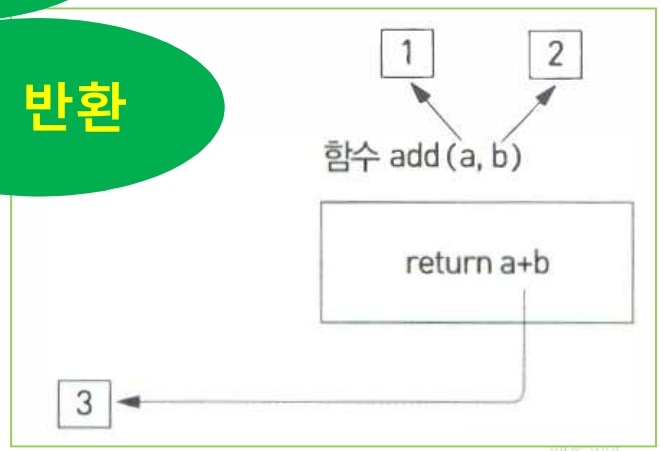
```
def 함수이름( 매개변수 목록 ):
    # 코드블록
    return <값>
```

- 실습 1 (함수 정의)

함수정의

값 반환

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

>>> add
<function add at 0x03A71228>
>>>
>>> add(5, 3)
8
```



2. 함수 정의하기

❖ 연습문제

- 아래의 결과가 출력되도록 프로그램을 작성하시오

이름은 무엇입니까? 홍길동
좋아하는 동물은 무엇입니까? 사자
홍길동님의 성격 분석을 시작하겠습니다.
홍길동님의 성격은 따뜻하면서 이성적인 성격입니다. 남들에게 이성적으로 보이면서도 내면 깊은 곳에는 따뜻함이 자리잡고 있습니다. 따라서 남들에겐 강해보일지 몰라도 말하지 못하는 유약함도 가졌습니다. 하지만 이러한 성격이 남들에게는 장점으로 비추어져 홍길동님을 의지하고 믿게 하는 기반이 됩니다. 사자라는 동물을 선택한 이유도 그와 같습니다. 사자처럼 때로는 강하고, 때로는 부드럽고, 때로는 유쾌한 모습을 지니고 있는 거죠. 이 장점으로 홍길동님은 아주 인기있는 사람임을 알 수 있습니다.

2. 함수 정의하기

- 함수는 **def** 키워드를 이용해서 코드블록에 이름을 붙인 형태

```
def 함수이름( 매개변수 목록 ):
    # 코드블록
    return <값>
```

- 실습 2 (함수는 다른 함수를 호출할 수 있다.
  - 함수는 계층 구조로, 입출력이나 기초적인 기능을 하는 함수들위에 고급 기능을 하는 함수를 작성하고, 그 위에 또 다른 고급 함수를 정의한다.

```
>>> def addabs(a, b):
        c = add(a, b)
        return abs(c)

>>> addabs(-5, -7)
12
```

2. 함수 기본

❖ 함수 기본

- 생성 기본형

```
def <함수 이름>():
    <문장>
```

- 코드 집합

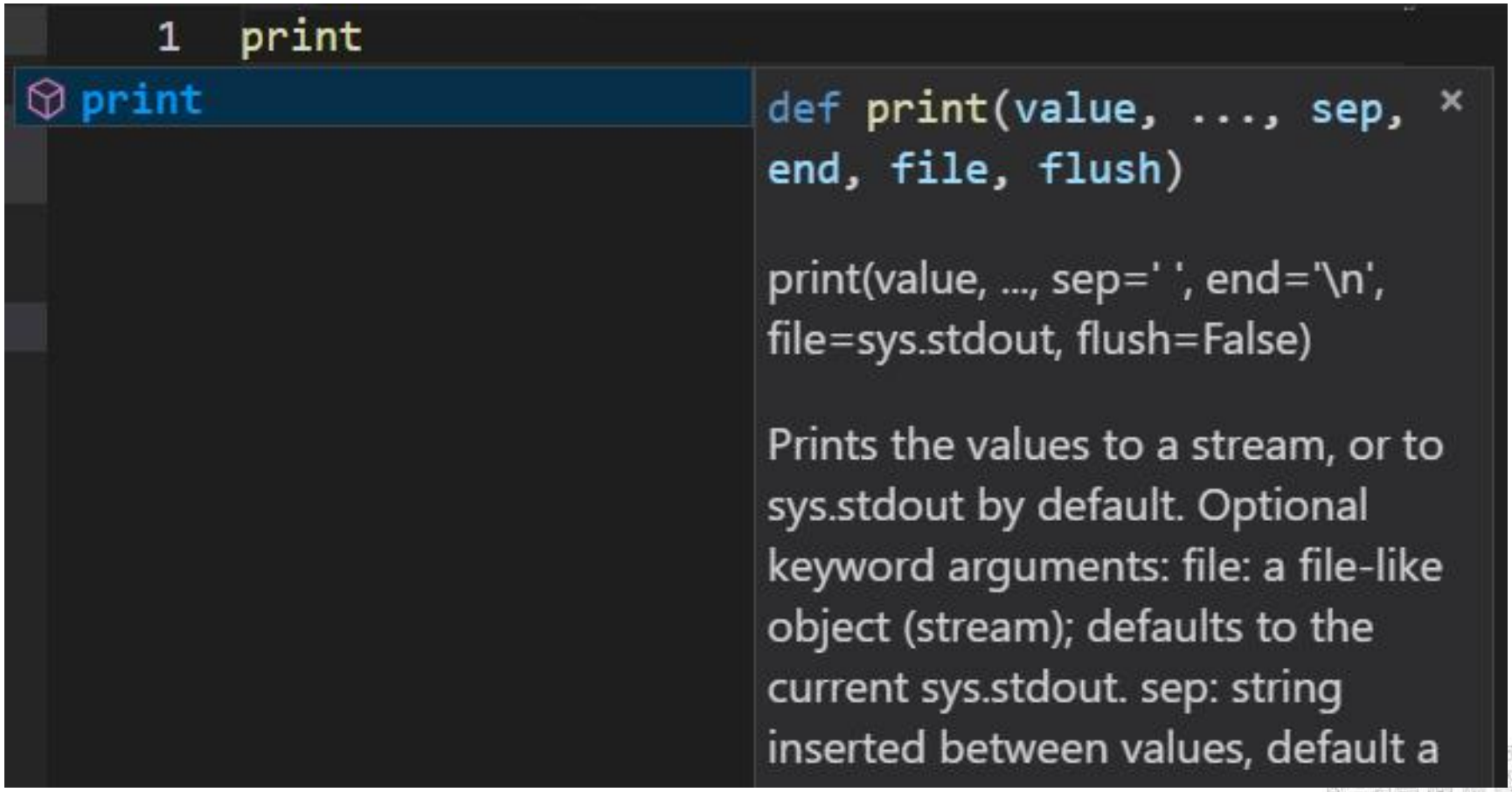
```
def print_3_times():
    print("안녕하세요")
    print("안녕하세요")
    print("안녕하세요")

print_3_times()
```

```
안녕하세요
안녕하세요
안녕하세요
```

2. 함수 기본

❖ 함수 기본

- 매개 변수 (기본 매개 변수, 가변 매개 변수, 키워드 매개 변수)



3. 매개변수를 입력 받는 여러 가지 방법

- 매개(媒介)는 중간에서 둘 사이의 관계를 맺어주는 것을 뜻하는 말
- 매개변수는 호출자와 함수 사이의 관계를 맺어주는 변수를 뜻함

```
def my_abs( arg ):
    if ( arg < 0 ):
        result = arg * -1
    else:
        result = arg

    return result
```

매개변수

- 실습 1 (잘못된 매개변수를 이용하여 함수를 호출하는 경우)

```
>>> my_abs()
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    my_abs()
TypeError: my_abs() missing 1 required positional argument: 'arg'
```

3. 매개변수를 입력 받는 여러 가지 방법

- 매개변수의 이름은 보통의 변수처럼 문자와 숫자, 그리고 _ 로 만들어짐
  - 숫자로 매개변수의 이름을 시작할 수는 없음.
  - 변수의 역할과 의미가 잘 나타나는 이름을 붙일 것.

```
def print_name1( 123abc ) :
    ...

def print_name2( aaa, bbb ) :
    ...

def print_name3( first_name, last_name ) :
    ...
```

사용불가. 123abc는 숫자가 앞에 오므로 사용할 수 없는 이름입니다.

나쁨. aaa, bbb를 봐서는 변수의 역할을 유추할 수 없습니다.

좋음. 의미를 분명하게 전달하는 이름입니다.

3. 매개변수를 입력 받는 여러 가지 방법

❖ 실습 2 (입력 받은 매개변수에 따라 문자열을 반복 출력)

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


>>> print_string("안녕하세요", 3)
안녕하세요
안녕하세요
안녕하세요
```

3. 매개변수를 입력 받는 여러 가지 방법

- 기본값 매개변수와 키워드 매개변수

❖ 기본값 매개변수(Default Argument Value)

- "이 매개변수를 입력할지 말지는 호출자 당신의 자유야. 단, 입력하지 않으면 내가 갖고 있는 기본값으로 할당할 거야."

❖ 실습 1 (키워드 매개변수 정의와 사용)

```
>>> def print_string(text, count=1) :
        for i in range(count) :
            print(text)


>>> print_string("안녕하세요")
안녕하세요
>>>
>>>
>>> print_string("안녕하세요", 2)
안녕하세요
안녕하세요
```

매개변수를 정의할 때
값을 할당해놓으면
기본값 매개변수가 된다.

호출할 때
두 번째 매개변수를 생략하면
기본값 1이 사용된다

3. 매개변수를 입력 받는 여러 가지 방법

- 기본값 매개변수와 키워드 매개변수

❖ 기본값 매개변수(Default Argument Value)

- 함수 작성 시 기본값이 있는 것과 기본값이 없는 것이 함께 있을 때는 기본값이 없는 것이 앞으로 오도록 함

❖ 실습 2 (키워드 매개변수 정의와 사용)

```
>>> def sayhello(name, place="파이썬월드"):
        print("%s님 안녕하세요. %s에 오신 것을 환영합니다." %(name, place))
        return name

>>> user = sayhello("john")
john님 안녕하세요. 파이썬월드에 오신 것을 환영합니다.

>>> user = sayhello("john", "프로그래밍 월드")
john님 안녕하세요. 프로그래밍 월드에 오신 것을 환영합니다.
>>>
```

3. 매개변수를 입력 받는 여러 가지 방법

- 기본값 매개변수와 키워드 매개변수

- 키워드 매개변수(Keyword Argument)
  - 매개변수가 많은 경우에는 호출자가 매개변수의 이름을 일일이 지정하여 데이터를 입력
- 실습 3 (키워드 매개변수 정의와 사용)

```
>>> def print_personnel(name, position='staff', nationality='Korea') :
        print('name = {0}'.format(name))
        print('position = {0}'.format(position))
        print('nationality = {0}'.format(nationality))


>>> print_personnel(name='홍길동')
name = 홍길동
position = staff
nationality = Korea
```

position과 nationality는 기본값이 사용됩니다.

3. 매개변수를 입력 받는 여러 가지 방법

- 기본값 매개변수와 키워드 매개변수

- 키워드 매개변수(Keyword Argument)
  - 매개변수가 많은 경우에는 호출자가 매개변수의 이름을 일일이 지정하여 데이터를 입력
- 실습 4 (키워드 매개변수 정의와 사용)

```
>>> def print_personnel(name, position='staff', nationality='Korea') :
        print('name = {0}'.format(name))
        print('position = {0}'.format(position))
        print('nationality = {0}'.format(nationality))


>>> print_personnel(name='홍길동', nationality='ROK')
name = 홍길동
position = staff
nationality = ROK
```

position만이 기본값을 사용합니다.

3. 매개변수를 입력 받는 여러 가지 방법

- 기본값 매개변수와 키워드 매개변수

❖ 키워드 매개변수(Keyword Argument)
- 매개변수가 많은 경우에는 호출자가 매개변수의 이름을 일일이 지정하여 데이터를 입력

❖ 실습 5 (키워드 매개변수 정의와 사용)

이름, 키, 몸무게를 인자로 입력받아 BMI를 출력하는 함수를 작성
(단 이름을 매개변수로 넣지 않으면 고객 이름은 "익명"으로 출력한다.)

```
>>> def bmi(height, weight, name="익명"):
        b = weight / ((height/100) * (height/100))
        print("%s님의 BMI는 %d입니다."%(name, b))

>>> bmi(176, 71)
익명님의 BMI는 22입니다.
```

3. 매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

- 가변 매개변수(Arbitrary Argument List)
  - 입력 개수가 달라질 수 있는 매개변수
  - *를 이용하여 정의된 가변 매개변수는 튜플

```
def 함수이름(*매개변수):
    코드블록
```

매개변수 앞에 *를 붙이면
해당매개변수는 가변으로 지정된다.

- 실습 6 (가변 매개변수)

```
>>> def merge_string(*text_list) :
        result = ''
        for s in text_list :
            result = result + s
        return result

>>> merge_string('아버지가', '방에', '들어가신다')
'아버지가방에들어가신다'
```

3. 매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

- 가변 매개변수(Arbitrary Argument List)
  - 입력 개수가 달라질 수 있는 매개변수
  - *를 이용하여 정의된 가변 매개변수는 튜플

def 함수이름(*매개변수):
**   코드블록**

매개변수 앞에 *를 붙이면
해당매개변수는 가변으로 지정된다.

- 실습 7 (가변 매개변수)

```
>>> def apark(*calloff):
        print("오늘", calloff, "은(는) 운행하지 않습니다.")
>>> apark("청룡열차")
오늘 ('청룡열차',) 은(는) 운행하지 않습니다.
>>> apark("청룡열차", "프롬라이드")
오늘 ('청룡열차', '프롬라이드') 은(는) 운행하지 않습니다.
>>>
```

3. 매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

❖ 실습 8 (딕셔너리 형식 가변 매개변수)

```
>>> def print_team(**players):
        for k in players.keys():
            print('{0} = {1}'.format(k, players[k]))

>>> print_team(이천웅='중견수', 김용의='1루수', 이형종='우익수', ₩
김현수='좌익수', 유강남='포수', 페게로='지명타자', 김민성='3루수', ₩
정주현='2루수', 오지환='유격수', 차우찬='투수', 류중일='감독')

이천웅 = 중견수
김용의 = 1루수
이형종 = 우익수
김현수 = 좌익수
유강남 = 포수
페게로 = 지명타자
김민성 = 3루수
정주현 = 2루수
오지환 = 유격수
차우찬 = 투수
류중일 = 감독
```

매개변수 앞에 **를 붙이면
딕셔너리 가변 매개변수가 됩니다.

3. 매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

❖ 실습 9 (일반 매개변수와 함께 사용하는 가변매개변수)

```
>>> def print_args(argc, *argv):
        for i in range(argc):
            print(argv[i])

>>> print_args(3, "argv1", "argv2", "argv3")
argv1
argv2
argv3
>>>
>>> print_args(argc=3, "argv1", "argv2", "argv3")
SyntaxError: positional argument follows keyword argument
```

가변 매개변수 앞에 정의된
일반 매개변수는 키워드 매개변수로 호출할 수 없습니다.

3. 매개변수를 입력 받는 여러 가지 방법 - 가변매개변수

❖ 실습 10 (가변 매개변수와 함께 사용하는 일반 매개변수)

```
>>> def print_args(*argv, argc):
        for i in range(argc):
            print(argv[i])


>>> print_args("argv1", "argv2", "argv3", argc=3)
argv1
argv2
argv3
>>>
>>> print_args("argv1", "argv2", "argv3", 3)
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    print_args("argv1", "argv2", "argv3", 3)
TypeError: print_args() missing 1 required keyword-only argument: 'argc'
```

가변 매개변수 뒤에 정의된 일반 매개변수는
반드시 키워드 매개변수로 호출해야 합니다.

3. 매개변수

❖ 매개변수 기본

```
def <함수 이름>(<매개변수>, <매개변수>, …):
    <문장>
```

- Ex) 매개변수로 value와 n 값을 받아 반복문을 n 값만큼 실행하여 value를 출력한다.

```
def print_n_times(value, n):
    for i in range(n):
        print(value)

# 함수를 호출합니다.
print_n_times("안녕하세요", 10)
```

```
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
```

3. 매개변수

❖ 매개변수 오류

▪ 매개변수 넣지 않은 경우

- print_n_times() 함수 매개변수 없음 오류

```
def print_n_times(value, n):
    for i in range(n):
        print(value)

print_n_times("안녕하세요")
```

```
Traceback (most recent call last):
  File "D:/5-3.py", line 5, in <module>
    print_n_times("안녕하세요")
TypeError: print_n_times() missing 1 required positional argument: 'n'
```

3. 매개변수

- 매개변수 더 많이 넣은 경우
  - print_n_times() 함수 매개변수 개수 오류

```
def print_n_times(value, n):
    for i in range(n):
        print(value)

# 함수를 호출합니다.
print_n_times("안녕하세요", 10, 20)
```

```
Traceback (most recent call last):
  File "D:/5-4.py", line 6, in <module>
    print_n_times("안녕하세요", 10, 20)
TypeError: print_n_times() takes 2 positional arguments but 3 were given
```

3. 매개변수

❖ 가변 매개변수 함수

- 매개변수를 원하는 만큼 받을 수 있는 함수(*을 붙여 계속 값을 대입)
  - 가변 매개변수 뒤에는 일반 매개변수 올 수 없음
  - 가변 매개변수는 하나만 사용할 수 있음

```
def <함수 이름>(<매개변수>, <매개변수>, …, *<가변 매개변수>):
    <문장>
```

```
def print_n_times(n, *values):
    # n번 반복합니다.
    for i in range(n):
        # values는 리스트처럼 활용합니다.
        for value in values:
            print(value)
        # 단순한 줄바꿈
        print()

# 함수를 호출합니다.
print_n_times(3, "안녕하세요", "즐거운", "파이썬 프로그래밍")
```

```
안녕하세요
즐거운
파이썬 프로그래밍

안녕하세요
즐거운
파이썬 프로그래밍

안녕하세요
즐거운
파이썬 프로그래밍
```

3. 매개변수

❖ 기본 매개변수

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

- <매개변수>=<값>
  - 매개변수를 입력하지 않을 경우 들어가는 기본 값
  - 기본 매개변수 뒤에는 일반 매개변수 올 수 없음

```
def print_n_times(value, n=2):
    # n번 반복합니다.
    for i in range(n):
        print(value)

# 함수를 호출합니다.
print_n_times("안녕하세요")
```

안녕하세요
안녕하세요

3. 매개변수

❖ 키워드 매개변수

- 가변 매개변수와 기본 매개변수를 함께 쓸 수 있는가?

- 기본 매개변수가 가변 매개변수보다 앞에 올 때

```
def print_n_times(n=2, *values):
    # n번 반복합니다.
    for i in range(n):
        # values는 리스트처럼 활용합니다.
        for value in values:
            print(value)
        # 단순한 줄바꿈
        print()

# 함수를 호출합니다.
print_n_times("안녕하세요", "즐거운", "파이썬 프로그래밍")
```

  - n에는 무엇이 들어가는가?

3. 매개변수

- range() 함수 매개변수에는 숫자만 들어갈 수 있음
  - 오류 발생

```
Traceback (most recent call last):
  File "D:/5-7.py", line 11, in <module>
    print_n_times("안녕하세요", "즐거운", "파이썬 프로그래밍")
  File "D:/5-7.py", line 3, in print_n_times
    for i in range(n):
TypeError: 'str' object cannot be interpreted as an integer
```

  - 기본 매개변수는 가변 매개변수 앞에 써도 의미가 없음

3. 매개변수

- 가변 매개변수가 기본 매개변수보다 앞에 올 때

```
def print_n_times(*values, n=2):
    # n번 반복합니다.
    for i in range(n):
        # values는 리스트처럼 활용합니다.
        for value in values:
            print(value)
        # 단순한 줄바꿈
        print()

# 함수를 호출합니다.
print_n_times("안녕하세요", "즐거운", "파이썬 프로그래밍", 3)
```

- 실행 결과가 어떻게 될 것인가?
  - ["안녕하세요", "즐거운", "파이썬 프로그래밍"] 3회 출력
  - ["안녕하세요", "즐거운", "파이썬 프로그래밍"3] 2회 출력

3. 매개변수

- 두 번째 형태로 실행

```
안녕하세요
즐거운
파이썬 프로그래밍
3

안녕하세요
즐거운
파이썬 프로그래밍
3
```

  - 가변 매개변수 우선시

3. 매개변수

- 키워드 매개변수
  - 두 가지를 함께 사용하는 방법은 없는가?
  - print() 함수 활용

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

```
def print_n_times(*values, n=2):
    # n번 반복합니다.
    for i in range(n):
        # values는 리스트처럼 활용합니다.
        for value in values:
            print(value)
        # 단순한 줄바꿈
        print()

# 함수를 호출합니다.
print_n_times("안녕하세요", "즐거운", "파이썬 프로그래밍", n=3)
```

```
안녕하세요
즐거운
파이썬 프로그래밍

안녕하세요
즐거운
파이썬 프로그래밍

안녕하세요
즐거운
파이썬 프로그래밍
```

– 매개변수 이름 지정하여 입력

3. 매개변수

- 기본 매개변수 중 필요한 값만 입력하기

```
def test(a, b=10, c=100) :
    print(a + b + c)


# 1) 기본 형태
test(10, 20, 30)

# 2) 키워드 매개변수로 모든 매개변수를 지정한 형태
test(a=10, b=100, c=200)

# 3) 키워드 매개변수로 모든 매개변수를 마구잡이로 지정한 형태
test(c=10, a=100, b=200)

# 4) 키워드 매개변수로 일부 매개변수만 지정한 형태
test(10, c=200)
```

```
60
310
310
220
```

1) a는 일반 매개변수이므로 반드시 해당 위치에

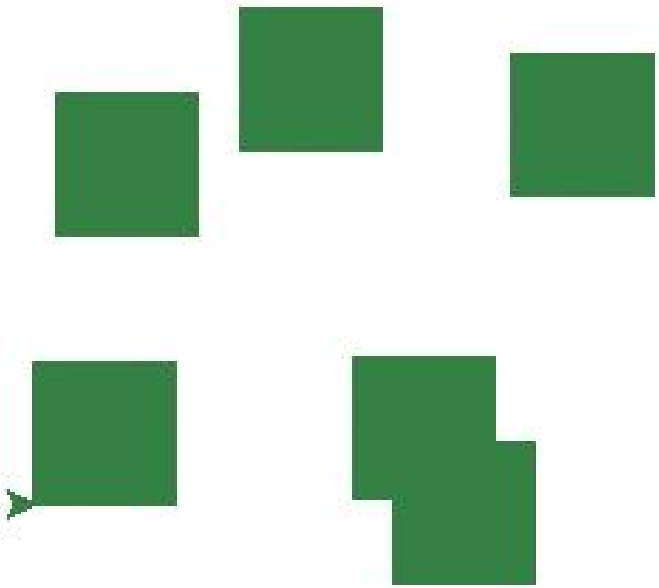
3) 키워드 지정하는 경우 매개변수 순서 무작위 변경 가능

4) b 생략: 필요한 매개변수에만 값 전달

3. 매개변수

❖ 마우스로 클릭하는 곳에 사각형 그리기

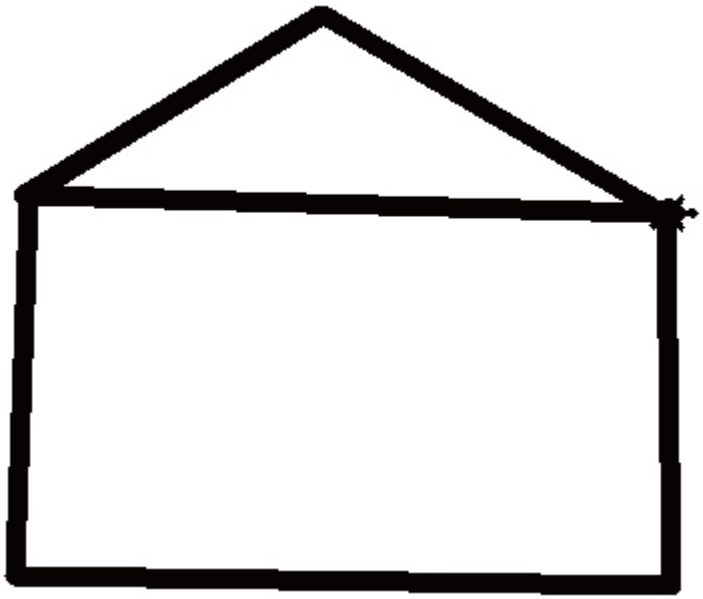
사용자가 화면에서 마우스 버튼을 클릭한 경우,
클릭 된 위치에 사각형을 그리는 프로그램을 작성한다.



3. 매개변수

❖ 마우스로 그림 그리기 : 콜백 함수

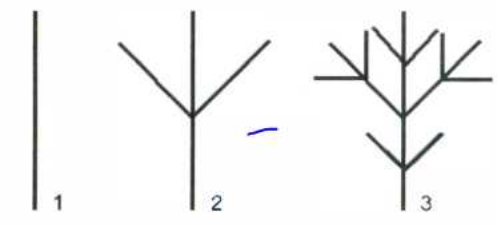
이벤트가 발생시, 이벤트를 처리하는 함수를 콜백(callback function) 라고 부른다.
콜백 함수를 이용하여 마우스로 그림을 그리는 프로그램을 작성한다.

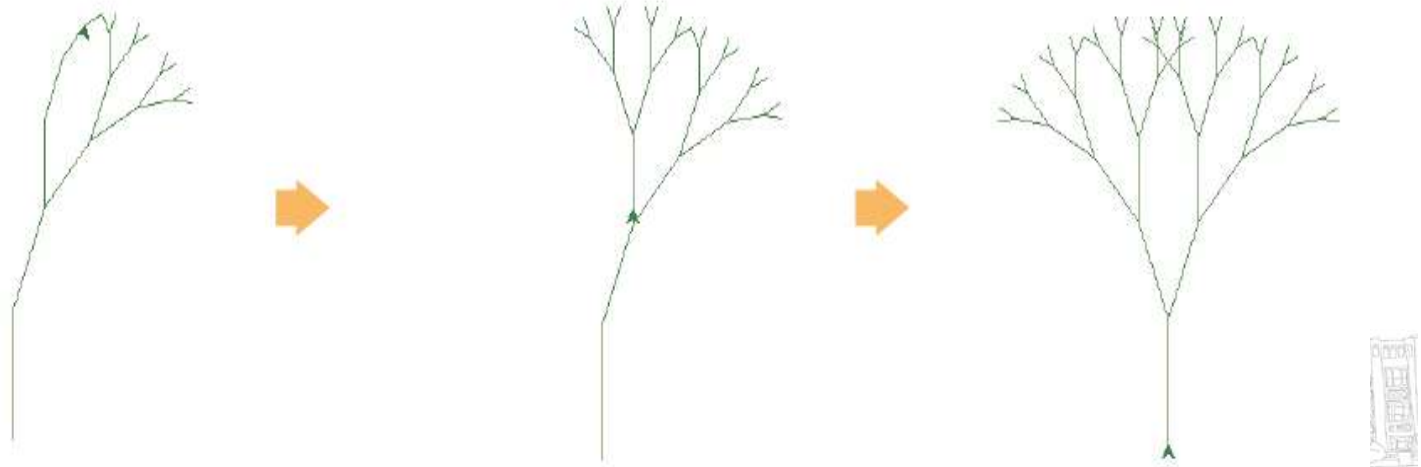


3. 매개변수

❖ 나무 그리기

순환적으로 나무를 그리는 프랙털(fractal) 프로그램을 작성한다.

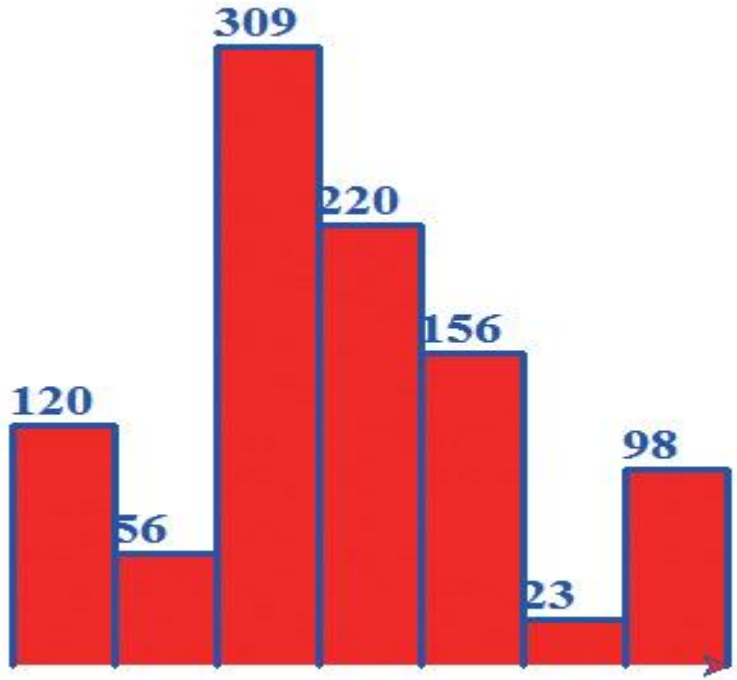




3. 매개변수

❖ 막대그래프 그리기

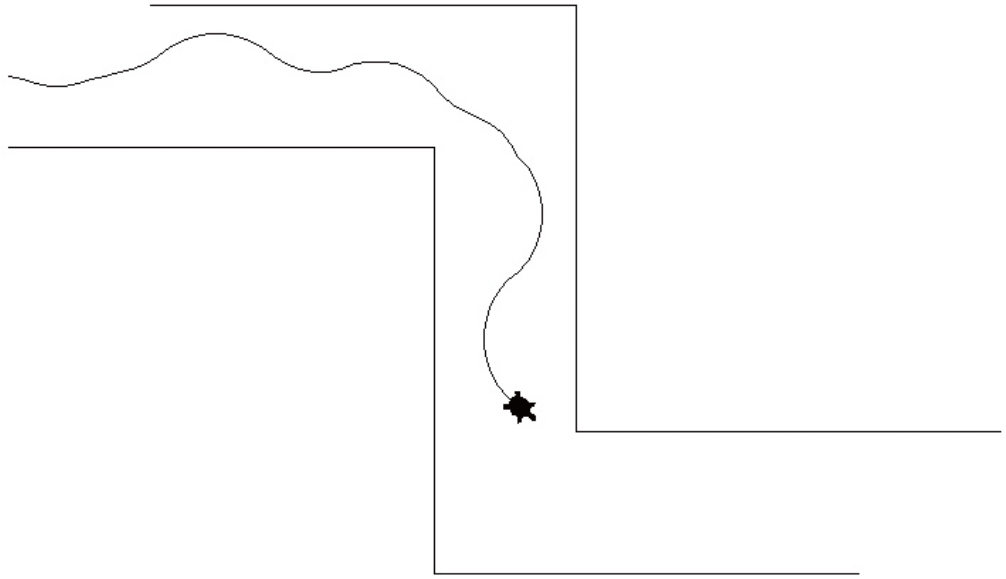
파이썬의 터틀 그래픽을 이용해서 막대 그래프를 그리는 프로그램을 작성한다.



3. 매개변수

❖ 막대그래프 그리기

화면에 미로를 만들고 거북이가 화살표를 이용하여 미로에 닫지 않게 진행하는 프로그램을 작성한다.

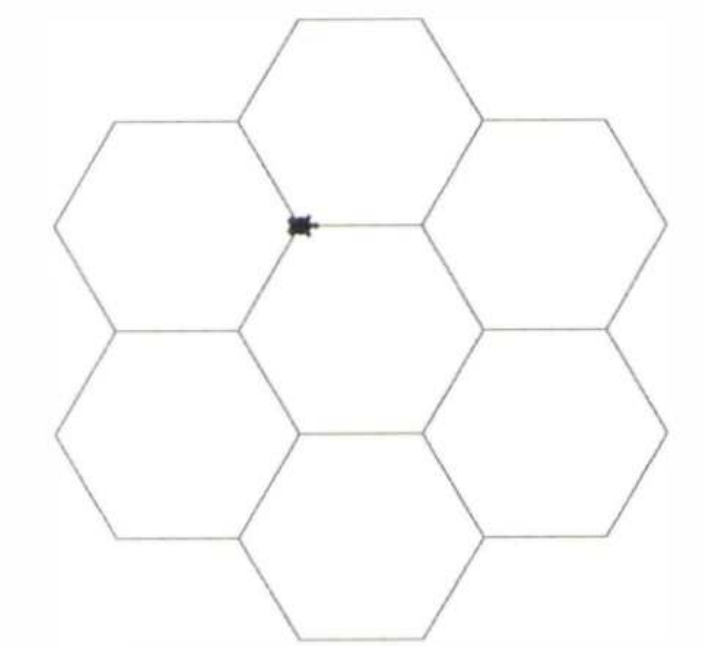


3. 매개변수

❖ 벌집 그리기

6각형을 그리는 draw_hexa() 함수를 작성하고 이 함수를 호출하여 다음과 같이 벌집 그림을 그리는 프로그램을 작성한다.

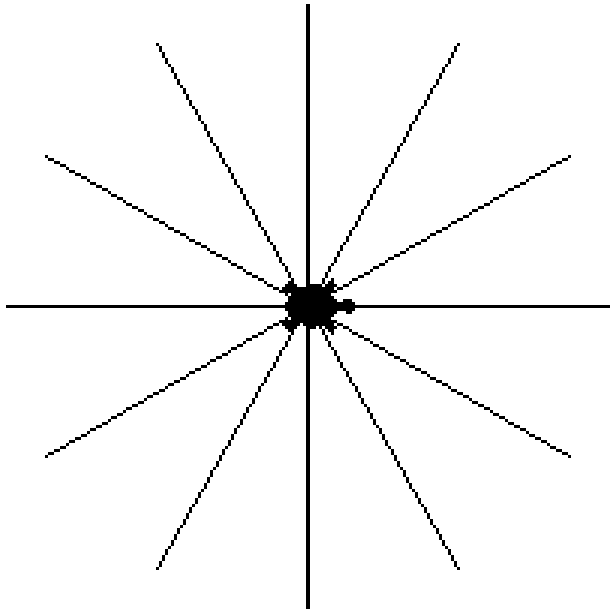
Hint> 6각형 -> forward(100) + left(60) * 6



3. 매개변수

❖ 벌집 그리기

터틀 그래픽에서 거북이를 움직이지 않고 선을 긋는 함수 draw_line()을 정의하고 이것을 이용하여 다음과 같은 거미줄과 같은 모양을 그리는 프로그램을 작성한다.



4. 리턴 : 호출자에게 반환하기

❖ **리턴값** (Return Value)

- input() 함수 실행 뒤 함수 결과 받아 사용

```
# input() 함수의 리턴값을 변수에 저장합니다.
value = input("> ")

# 출력합니다.
print(value)
```

❖ 기본

- return 키워드
  - 함수를 실행했던 위치로 돌아가기

```
# 함수를 정의합니다.
def return_test() :
    print("A 위치입니다")
    return      # 리턴합니다.
    print("B 위치입니다")


# 함수를 호출합니다.
return_test()
```

A 위치입니다

4. 리턴 : 호출자에게 반환하기

❖ 자료와 함께 리턴

- 리턴 뒤에 자료 입력하면 이를 가지고 돌아감

```
# 함수를 정의합니다.
def return_test() :
    return 100

# 함수를 호출합니다.
value = return_test()
print(value)
```

100

```
# 함수를 정의합니다.
def return_test() :
    return

# 함수를 호출합니다.
value = return_test()
print(value)
```

None

4. 리턴 : 호출자에게 반환하기

- 경고

```
# 함수를 호출합니다.
value = return_test()
[pylint] E1128:Assigning to function call which only returns No
ne
[pylint] C0103:Invalid constant name "value"
value
value = return_test()
```

– 아무 것도 리턴하지 않는 함수 값 할당

4. 리턴 : 호출자에게 반환하기

- 인수 없이 return 문만을 사용하면 함수를 호출한 측에 아무 값도 전달하지 않는다.

```
>>> def nothing() :
        return

>>> nothing()
>>>
>>> print(nothing())
None
```

```
>>> def simple() :
        pass

>>> simple()
>>>
>>> print(simple())
None
```

```
>>> def swap(a, b) :
        return b, a

>>> a = 10; b = 20
>>> swap(a, b)
(20, 10)
```

4. 리턴 : 호출자에게 반환하기

- 함수가 호출자에게 값을 반환할 때에는 return문을 이용
- return문을 이용하는 세 가지 방법
  - return문에 결과 데이터를 담아 실행하기
    ➔ 함수가 즉시 종료되고 호출자에게 결과가 전달됨.
  - return문에 아무 결과도 넣지 않고 실행하기 ➔ 함수가 즉시 종료됨.
  - return문 생략하기 ➔ 함수의 모든 코드가 실행되면 종료됨.

- 실습 1 : **리턴문에 결과 데이터를 담아 실행하기**

```
>>> def multiply(a, b):
        return a*b

>>> result = multiply(2, 3)

>>> result
6
```

return문은 함수의 실행을 종료시키고 자신에게 넘겨진 데이터를 호출자에게 전달합니다.

4. 리턴 : 호출자에게 반환하기

❖ 실습 2 (여러 개의 return)

```
>>> def my_abs(arg):
        if arg < 0:
            return arg * -1
        else:
            return arg

>>> result = my_abs(-1)

>>> result
1
>>>
>>> result = my_abs(1)
>>> result
1
```

4. 리턴 : 호출자에게 반환하기

❖ 실습 3 (None을 반환하는 경우)

```
>>> def my_abs(arg):
        if arg < 0:
            return arg * -1
        elif arg > 0:
            return arg

>>> result = my_abs(-1)
>>> result
1
```

```
>>> result = my_abs(1)
>>> result
1
>>> result = my_abs(0)
>>> result
>>>
>>> result == None
True
>>>
>>> type(result)
<class 'NoneType'>
```

return을 실행하지 못하고 함수가 종료되면
함수는 호출자에게 None을 반환합니다.

4. 리턴 : 호출자에게 반환하기

❖ 실습 4 (결과 없는 return)

```
>>> def ogamdo(num):
        for i in range(1, num+1):
            print('제 {0}의 아해'.format(i))
            if i == 5:
                return
```

반환할 데이터 없이 실행하는 return문은 "반환"의 의미보다는 "함수 종료"의 의미로 사용됩니다

4. 리턴 : 호출자에게 반환하기

❖ 실습 4 (결과 없는 return)

```
>>> ogamdo(3)
제 1의 아해
제 2의 아해
제 3의 아해
>>> ogamdo(5)
제 1의 아해
제 2의 아해
제 3의 아해
제 4의 아해
제 5의 아해
>>> ogamdo(8)
제 1의 아해
제 2의 아해
제 3의 아해
제 4의 아해
제 5의 아해
```

8을 입력하면 for 반복문은 8번 반복을 수행하려고 준비하겠지만 실행되는 return문 때문에 다섯 번 수행하면 함수가 종료되고 맙니다.

4. 리턴 : 호출자에게 반환하기

❖ 실습 5 (return없는 함수)

```
>>> def print_something(*args):
        for s in args:
            print(s)

>>> print_something(1, 2, 3, 4, 5)

1
2
3
4
5
```

반환할 결과도 없고 함수를 중간에 종료시킬 일도 없다면 return문은 생략해도 됩니다.

5. 기본적인 함수

- 함수 활용의 방법들
- 범위 내부의 정수 모두 더하는 함수

```
# 함수를 선언합니다.
def sum_all(start, end):
    # 변수를 선언합니다.
    output = 0
    # 반복문을 돌려 숫자를 더합니다.
    for i in range(start, end + 1):
        output += i
    # 리턴합니다.
    return output

# 함수를 호출합니다.
print("0 to 100:", sum_all(0, 100))
print("0 to 1000:", sum_all(0, 1000))
print("50 to 100:", sum_all(50, 100))
print("500 to 1000:", sum_all(500, 1000))
```

5. 기본적인 함수

- 기본 매개변수 사용하여 조금 더 편리하게

```
# 함수를 선언합니다.
def sum_all(start=0, end=100, step=1):
    # 변수를 선언합니다.
    output = 0
    # 반복문을 돌려 숫자를 더합니다.
    for i in range(start, end + 1, step):
        output += i
    # 리턴합니다.
    return output

# 함수를 호출합니다.
print("A.", sum_all(0, 100, 10))
print("B.", sum_all(end=100))
print("C.", sum_all(end=100, step=2))
```

5. 기본적인 함수

- 기본 매개변수 사용하여 조금 더 편리하게

```
# custom_max(<리스트>)
# 리스트 내부에서 최대값

def custom_max(input_list):
    output = input_list[0]

    for element in input_list:
        if output < element:
            output = element

    return output

print(custom_max([32, 32, 923, 2, 123]))
```

6. 재귀 함수 : 자기 스스로를 호출하는 함수

- 재귀함수(Recursive Function)는 자기 스스로를 호출하는 함수
- 함수가 자기 자신을 부르는 것을 재귀호출(Recursive Call)이라 함.

- 재귀 함수의 예

```
def recursion_function():
    print("안녕하세요")
    recursion_function()

recursion_function()
```

```
def recursion_function():
    print("안녕하세요")
    if i < 10:
        recursion_function(i + 1)

recursion_function(0)
```

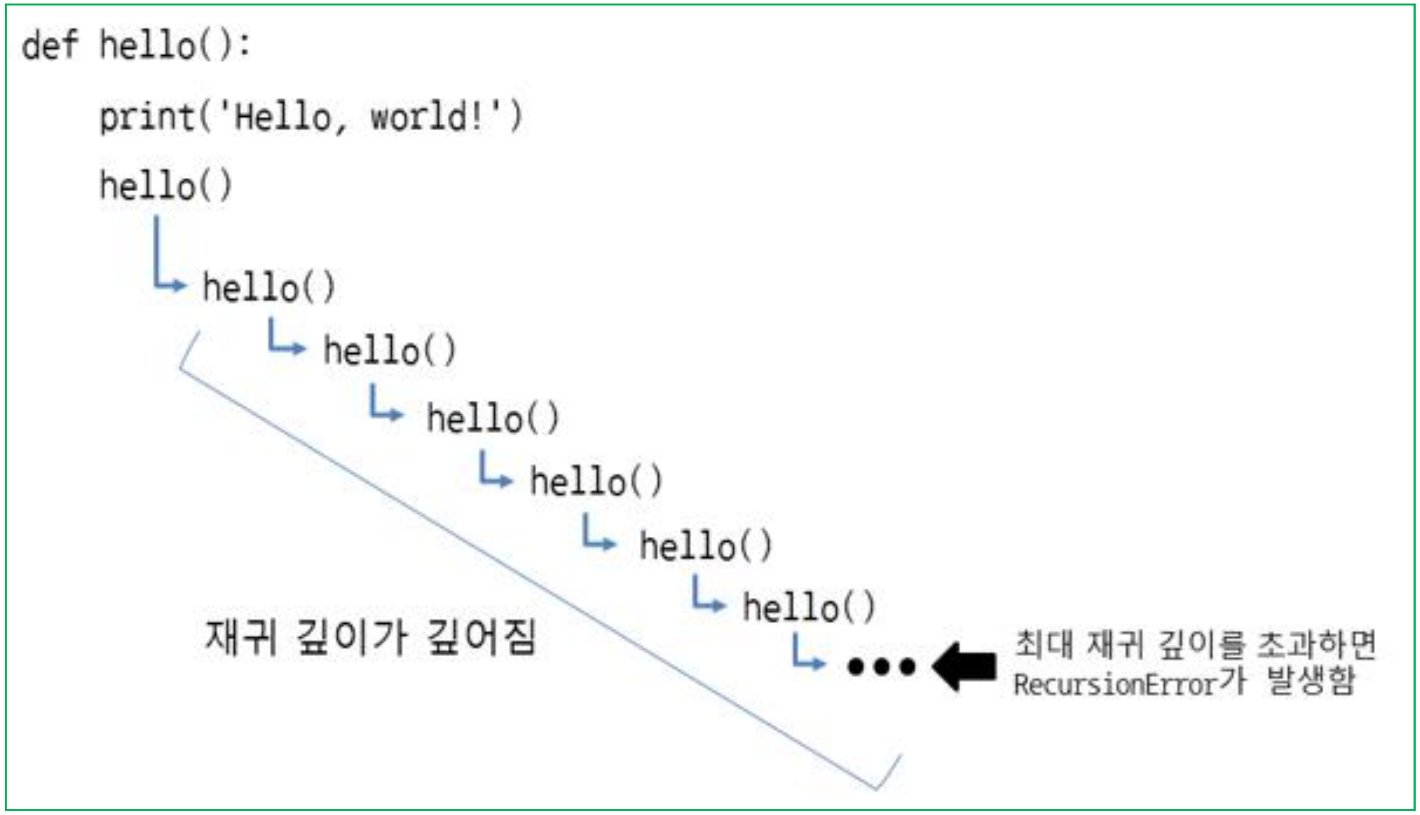
- 재귀함수 작성 시 종료할 수 있는 매개변수를 지정해주어야 함

6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ 재귀 함수의 예

- 재귀함수 호출과 스택 넘침 현상
- RecursionError 발생

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

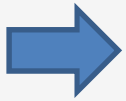


6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ 재귀호출에 종료 조건 만들기

- 재귀호출을 사용하려면 반드시 종료 조건을 만들어야 함

```
def hello(count):
    if count == 0:    # 종료 조건을 만듦. count 가 0 이면 다시 hello 함수를 호출하지 않고 끝냄
        return

    print('Hello, world!', count)

    count -= 1      # count 를 1 감소시킨 뒤
    hello(count)    # 다시 hello 에 넣음

hello(5)    # hello 함수 호출
```



```
Hello, world! 5
Hello, world! 4
Hello, world! 3
Hello, world! 2
Hello, world! 1
```

6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ 재귀호출에 종료 조건 만들기

- 재귀호출을 사용하려면 반드시 종료 조건을 만들어야 함

```
            5
            ↓
def hello(count):
    if count == 0:
        return

    print('Hello, world!', count)

    count -= 1
    hello(count)
    ↳ hello(4)
        ↳ hello(3)
            ↳ hello(2)
                ↳ hello(1)
                    ↳ hello(0) 종료 조건을 만족하므로 재귀호출을 끝냄
```

6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ **팩토리얼** (Factorial)

$$n! = n * (n - 1) * (n - 2) * \cdots * 1$$

❖ 반복문으로 팩토리얼 구하기

```
# 함수를 선언합니다.
def factorial(n):
    # 변수를 선언합니다.
    output = 1
    # 반복문을 돌려 숫자를 더합니다.
    for i in range(1, n + 1):
        output *= i
    # 리턴합니다.
    return output

# 함수를 호출합니다.
print("1!:", factorial(1))
print("2!:", factorial(2))
print("3!:", factorial(3))
print("4!:", factorial(4))
print("5!:", factorial(5))
```

```
1!: 1
2!: 2
3!: 6
4!: 24
5!: 120
```

6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ 재귀함수로 팩토리얼 구하기

- 재귀 (Recursion)
  - 자기 자신을 호출하는 것

$$n! = n * (n - 1) * (n - 2) * \cdots * 1$$

$$factorial(n) = n * factorial(n - 1) \ (n >= 2 \text{ 일 때})$$
$$factorial(1) = 1$$

- factorial(4) 구하기

```
f(4) = 4 * f(3)
     = 4 * 3 * f(2)
     = 4 * 3 * 2 * f(1)
     = 4 * 3 * 2 * 1
```

f(1)은 1이므로 곧바로 1로 변경합니다.

6. 재귀 함수 : 자기 스스로를 호출하는 함수

- 재귀함수 사용

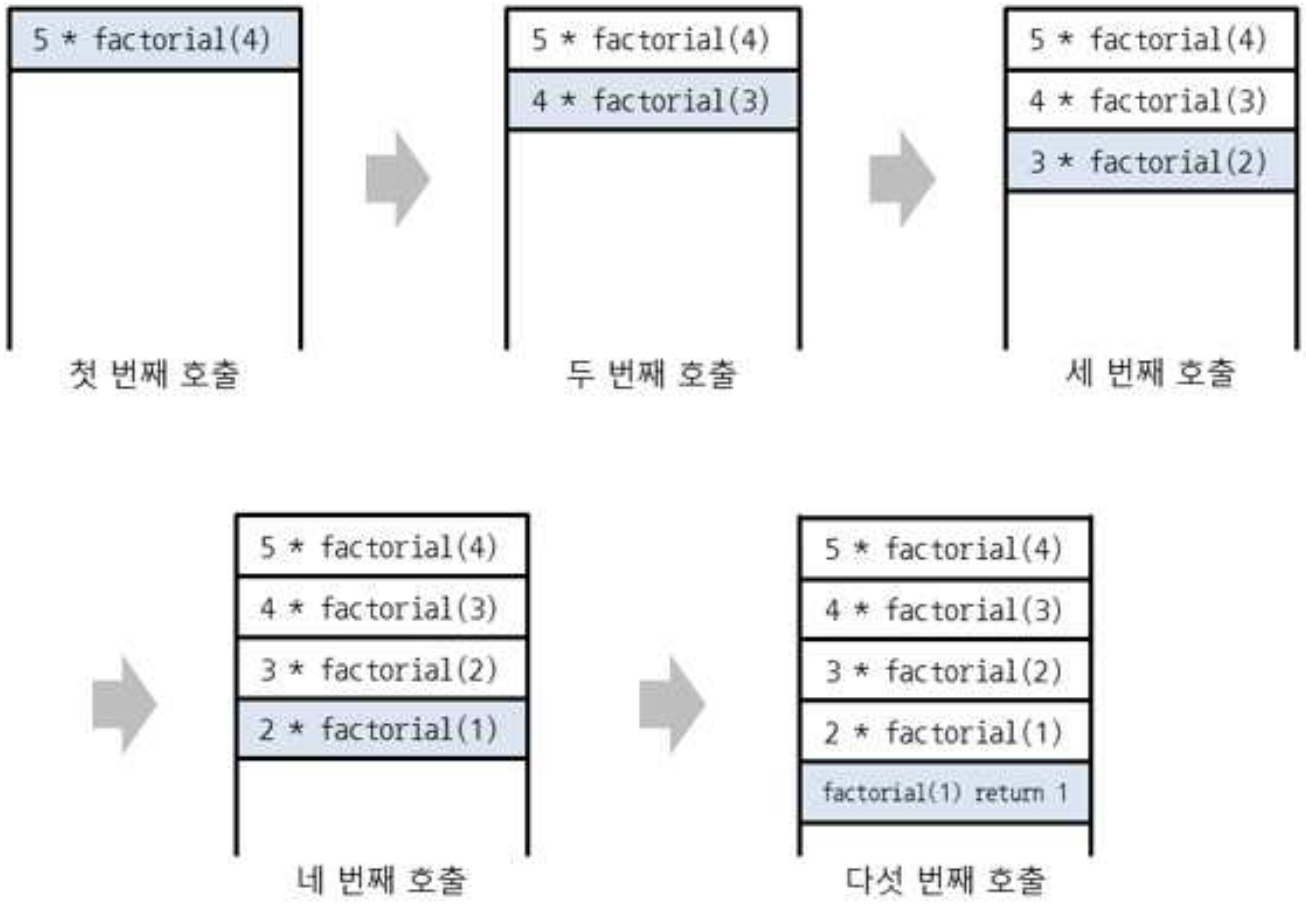
```
# 함수를 선언합니다.
def factorial(n):
    # n이 1이라면 1을 리턴
    if n == 1:
        return 1
    # n이 1이 아니라면 n * (n-1)!을 리턴
    else:
        return n * factorial(n - 1)

# 함수를 호출합니다.
print("1!:", factorial(1))
print("2!:", factorial(2))
print("3!:", factorial(3))
print("4!:", factorial(4))
print("5!:", factorial(5))
```

```
1!: 1
2!: 2
3!: 6
4!: 24
5!: 120
```

6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ 재귀함수로 팩토리얼 구하기

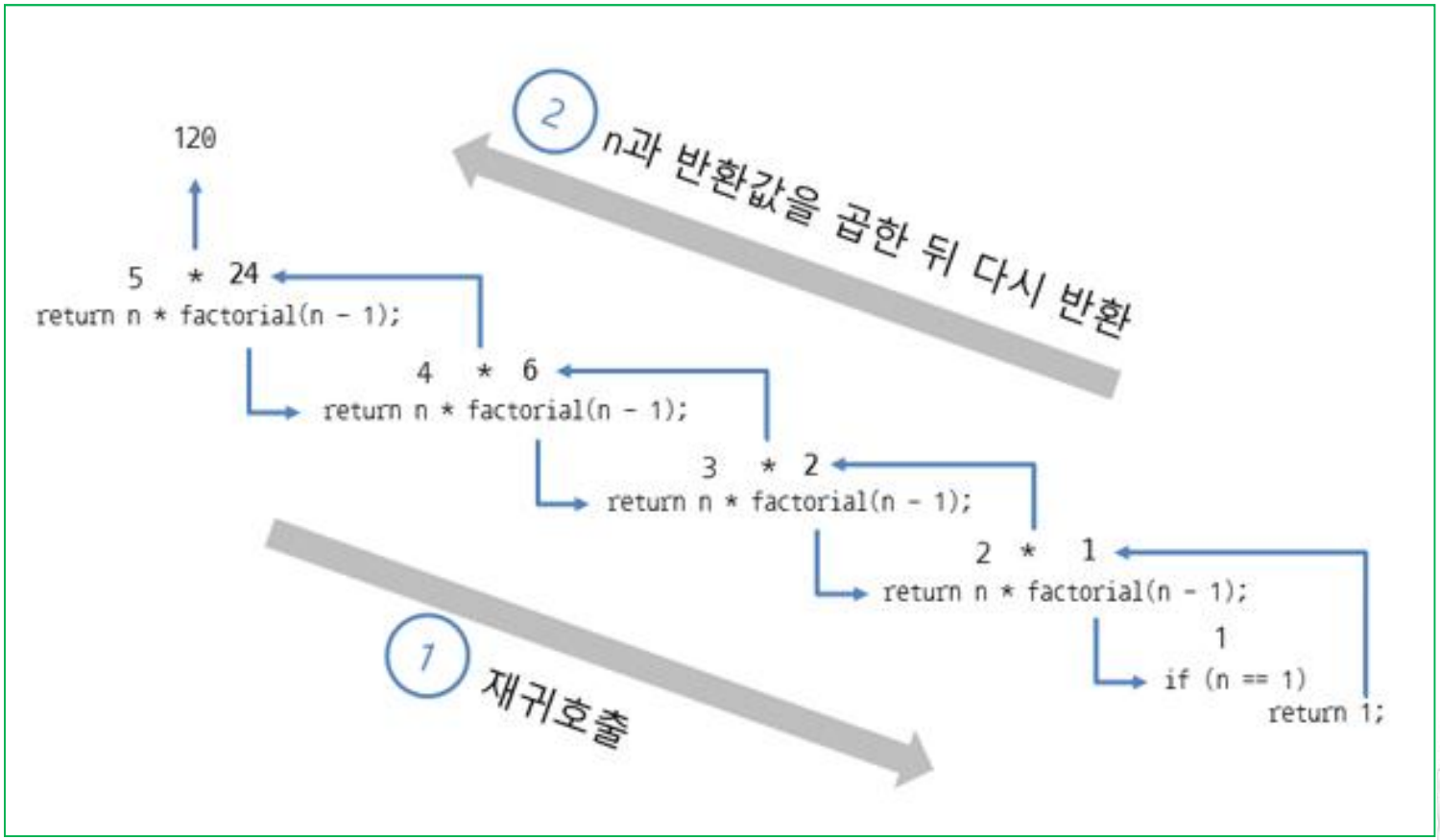
- factorial(5) 구하기



6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ 재귀함수로 팩토리얼 구하기

- factorial(5) 구하기



6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ 실습 1 (팩토리얼을 재귀 함수로 구현)

- 다음 재귀 관계식(Recurrence relation)을 파이썬 코드로 옮기는 예제

- $$n! = \begin{cases} 1, & n = 0 \\ (n-1)! \times n, & n > 0 \end{cases}$$

```
>>> def factorial(n):
        if n == 0:
                return 1
        elif n > 0:
                return factorial(n-1)*n

>>>
>>> factorial(5)
120
>>> factorial(10)
3628800
>>> factorial(100)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

6. 재귀 함수 : 자기 스스로를 호출하는 함수

- 재귀 호출의 단계가 깊어질 수록 메모리를 추가적으로 사용하기 때문에 재귀 함수가 종료될 조건을 분명하게 만들어야 함.
- 실습 2 (재귀함수를 사용할 때 주의할 점)

```
>>> def no_idea():
        print("나는 아무 생각이 없다.")
        print("왜냐하면")
        no_idea()

>>> no_idea()
나는 아무 생각이 없다.
왜냐하면
나는 아무 생각이 없다.
왜냐하면
나는 아무 생각이 없다.
왜냐하면
```

종료할 조건도 지정해주지 않은 채
무조건 재귀호출을 수행하면
스택 오버플로우가 발생합니다.

6. 재귀 함수 : 자기 스스로를 호출하는 함수

❖ 실습 2 결과 (재귀함수를 사용할 때 주의할 점)

```
>>> no_idea()
나는 아무 생각이 없다.
왜냐하면
나는 아무 생각이 없다.
왜냐하면...
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    no_idea()
  File "<pyshell#8>", line 4, in no_idea
    no_idea()
...
File "<pyshell#8>", line 2, in no_idea
    print("나는 아무 생각이 없다.")
  File "C:₩Python34₩lib₩idlelib₩PyShell.py", line 1342, in write
    return self.shell.write(s, self.tags)
RuntimeError: maximum recursion depth exceeded while calling a Python object
```

스택 오버 플로우가 발생하면 파이썬에서 지정해놓은 최대 재귀 단계를 초과했다는 에러가 출력됩니다.

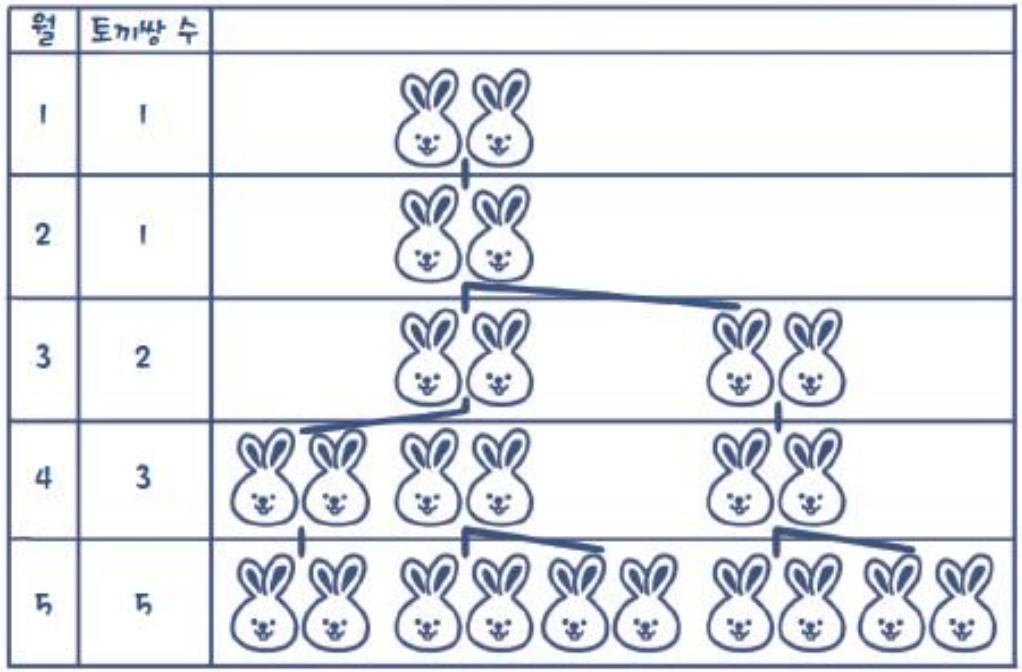
7. 메모화

❖ 메모화 (Memorize)

- 재귀 함수 사용으로 인해 발생하는 문제 해결
  - 상황에 따라 같은 것을 과도하게 반복하는 문제

- 피보나치 수열
  - Ex)

- 처음에는 토끼가 한 쌍만 존재한다.
- 두 달 이상 된 토끼는 번식할 수 있다.
- 번식한 토끼는 매달 새끼를 한 쌍씩 낳는다.
- 토끼는 죽지 않는다고 가정한다.



7. 메모화

- 코드로 구현

```
# 함수를 선언합니다.
def fibonacci(n):
    if n == 1:
        return 1
    if n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# 함수를 호출합니다.
print("fibonacci(1):", fibonacci(1))
print("fibonacci(2):", fibonacci(2))
print("fibonacci(3):", fibonacci(3))
print("fibonacci(4):", fibonacci(4))
print("fibonacci(5):", fibonacci(5))
```

```
fibonacci(1): 1
fibonacci(2): 1
fibonacci(3): 2
fibonacci(4): 3
fibonacci(5): 5
```

7. 메모화

- 문제 찾기

```
# 변수를 선언합니다.
counter = 0

# 함수를 선언합니다.
def fibonacci(n):
    # 어떤 피보나치 수를 구하는지 출력합니다.
    print("fibonacci({})를 구합니다.".format(n))
    global counter
    counter += 1
    # 피보나치 수를 구합니다.
    if n == 1:
        return 1
    if n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# 함수를 호출합니다.
fibonacci(10)
print("---")
print("fibonacci(10) 계산에 활용된 덧셈 횟수는 {}번입니다.".format(counter))
```

7. 메모화

- 참조 에러 (global counter 구문을 생략할 경우)

```
# 변수를 선언합니다.
counter = 0

# 함수를 선언합니다.
def fibonacci(n):
    counter += 1
    # 피보나치 수를 구합니다.
    if n == 1:
        return 1
    if n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# 함수를 호출합니다.
print(fibonacci(10))
```

```
Traceback (most recent call last):
  File "D:/5-22.py", line 16, in <module>
    print(fibonacci(10))
  File "D:/5-22.py", line 6, in fibonacci
    counter += 1
UnboundLocalError: local variable 'counter' referenced before assignment
```

7. 메모화

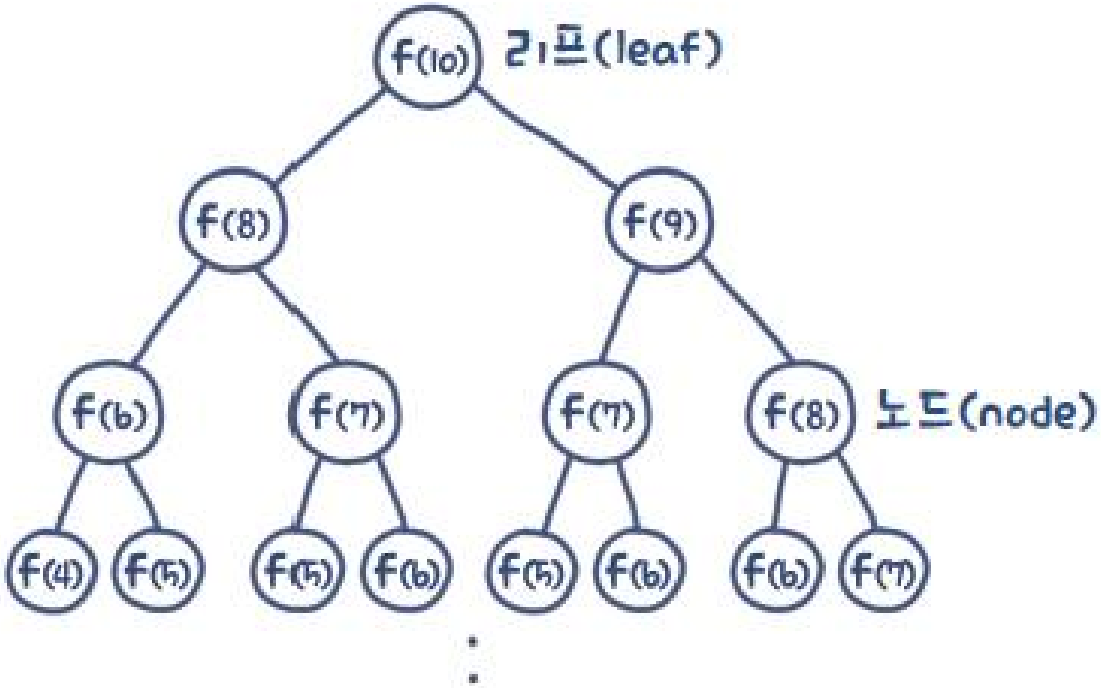
– 과도한 계산과정으로 시간 소요

```
fibonacci(10)를 구합니다.
fibonacci(9)를 구합니다.
fibonacci(8)를 구합니다.
...생략...
fibonacci(2)를 구합니다.
fibonacci(1)를 구합니다.
fibonacci(2)를 구합니다.
print("——")
fibonacci(10) 계산에 활용된 덧셈 횟수는 109번입니다.
```

```
fibonacci(10) 계산에 활용된 덧셈 횟수는 109번입니다.
fibonacci(11) 계산에 활용된 덧셈 횟수는 177번입니다.
fibonacci(12) 계산에 활용된 덧셈 횟수는 287번입니다.
fibonacci(13) 계산에 활용된 덧셈 횟수는 465번입니다.
fibonacci(14) 계산에 활용된 덧셈 횟수는 753번입니다.
fibonacci(35) 계산에 활용된 덧셈 횟수는 18454929번입니다.
```

7. 메모화

– 그 원인은?

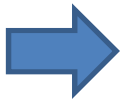


7. 메모화

❖ 메모 (Memo)

- 딕셔너리 사용하여 한 번 계산한 값 저장(같은 값 한 번만 계산)

```
# 메모 변수를 만듭니다.
dictionary = {
    1: 1,
    2: 1
}

# 함수를 선언합니다.
def fibonacci(n):
    if n in dictionary:
        # 메모 되어 있으면 메모된 값 리턴
        return dictionary[n]
    else:
        # 메모 되어 있지 않으면 값을 구함
        output = fibonacci(n - 1) + fibonacci(n - 2)
        dictionary[n] = output
        return output

# 함수를 호출합니다.
print("fibonacci(10):", fibonacci(10))
print("fibonacci(20):", fibonacci(20))
print("fibonacci(30):", fibonacci(30))
print("fibonacci(40):", fibonacci(40))
print("fibonacci(50):", fibonacci(50))
```



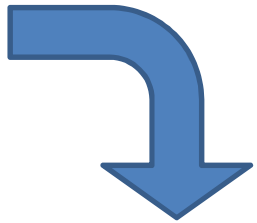
```
fibonacci(10): 55
fibonacci(20): 6765
fibonacci(30): 832040
fibonacci(40): 102334155
fibonacci(50): 12586269025
```

7. 메모화

❖ 조기 리턴 (Early Return)

- return을 중간에 사용하는 형태

```
# 함수를 선언합니다.
def fibonacci(n):
    if n in dictionary:
        # 메모 되어 있으면 메모 된 값 리턴
        return dictionary[n]
    else:
        # 메모 되어 있지 않으면 값을 구함
        output = fibonacci(n - 1) + fibonacci(n - 2)
        dictionary[n] = output
        return output
```



```
# 함수를 선언합니다.
def fibonacci(n):
    if n in dictionary:
        # 메모 되어 있으면 메모된 값 리턴
        return dictionary[n]
    # 메모 되어 있지 않으면 값을 구함
    output = fibonacci(n - 1) + fibonacci(n - 2)
    dictionary[n] = output
    return output
```

8. 튜플

- 사전적 의미는 Tuple과 List가 비슷
  - 리스트는 "목록"
  - 튜플은 "N개의 요소로 된 집합"

- 파이썬의 List와 Tuple의 차이
  - List는 데이터 변경 가능(리스트 생성 후 추가/수정/삭제 가능)
  - Tuple은 데이터 변경 불가능(튜플 생성 후 추가/수정 불가능)
  - List는 이름 그대로 목록 형식의 데이터를 다루는 데 적합
  - Tuple은 위-경도 좌표나 RGB 색상처럼 작은 규모의 자료구조를 구성하기에 적합

8. 튜플

- 변경이 불가능한 자료형이 필요한 이유?
  - 성능
    - 변경 가능한 자료형과는 달리 데이터를 할당할 공간의 내용이나 크기가 달라지지 않기 때문에 생성 과정이 간단
    - 데이터가 오염되지 않을 것이라는 보장이 있기 때문에 복사본을 만드는 대신 그냥 원본을 사용
  - 신뢰 가능한 코드
    - 변경되지 않아야 할 데이터를 오염시키는 버그를 만들 가능성 제거
    - 코드를 설계할 때부터 변경이 가능한 데이터와 그렇지 않은 데이터를 정리해서 코드에 반영

- 문자열도 변경이 불가능한 자료형

8. 튜플

❖ 튜플 (Tuple)

- 자료형 – 리스트와 유사
- 한 번 결정된 요소 바꿀 수 없음

(<식별자>, <식별자>, <식별자>, …)

```
# 튜플을 생성합니다.
tuple_test = (10, 20, 30)

# 튜플의 요소 출력하기
print("# 튜플의 요소 출력하기")
print("tuple_test[0]:", tuple_test[0])
print("tuple_test[1]:", tuple_test[1])
print("tuple_test[2]:", tuple_test[2])
print()

# 튜플의 요소 변경하기
print("# 튜플은 요소를 변경할 수 없어요(예외가 발생합니다)")
tuple_test[0] = 10
```

8. 튜플

- 결과

```
# 튜플의 요소 출력하기
tuple_test[0]: 10
tuple_test[1]: 20
tuple_test[2]: 30

# 튜플은 요소를 변경할 수 없어요(예외가 발생합니다)
Traceback (most recent call last):
  File "D:₩02.DO - 수업교안₩61. 파이썬₩11.헬로코딩파이썬₩chapter_5₩5-26.py", line 13, in <mo
dule>
    tuple_test[0] = 10
TypeError: 'tuple' object does not support item assignment
```

- 요소를 하나만 가지는 튜플
  - (273,)

8. 튜플

❖ 실습 1 (튜플 생성) **# 소괄호를 사용한다**

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

8. 튜플

❖ 실습 2 (튜플 생성2) **# ( ) 없이 콤마(,) 만 사용**

```
>>> a = 1, 2, 3, 4
>>>
>>> a
(1, 2, 3, 4)
>>>
>>> type(a)
<class 'tuple'>
```

❖ 파이썬은 튜플을 출력할 때 괄호 ( ) 를 포함한다.

❖ 튜플을 정의할 때는 ( )가 필요 없다. 그러나 값들을 괄호로 묶어서 튜플을 정의한다면, 이것이 튜플인지 구분하기가 쉬워진다.

8. 튜플

❖ 실습 3 (요소가 하나인 튜플 생성)

```
>>> a = (1,)
>>> a
(1,)
>>> type(a)
<class 'tuple'>
>>>
>>> b = 1, 2
>>> b
(1, 2)
>>> type(b)
<class 'tuple'>
```

요소가 하나인 경우엔
요소 뒤에 , 추가하며
콤마를 생략할 경우 정수로 인지한다.

요소가 두 개 이상인 경우에는
콤마를 생략해도 된다.

8. 튜플

❖ 실습 3 (요소가 하나인 튜플 생성)

- 요소가 하나인 경우에는 요소 뒤에 콤마 (,)를 추가해주어야 한다.
  콤마(,)를 생략할 경우에는 해당 요소를 정수로 인지하게 된다.
  요소가 두 개 이상인 경우에는 마지막 요소에는 생략해주어도 된다.

❖ 데이터 분석, 머신러닝, 딥러닝의 경우

```
import numpy as np

# 다차원 배열
a = np.array([1, 2, 3, 4, 5, 6])      # 1차원 배열 -> [리스트]
b = np.array([1, 2, 3], [4, 5, 6])    # 2차원 배열 -> [리스트]

print(a.shape)
print(b.shape)
```

8. 튜플

❖ 실습 4 (슬라이싱)

```
>>> a = (1, 2, 3, 4, 5, 6)
>>>
>>> a[:3]
(1, 2, 3)
>>>
>>> a[4:6]
(5, 6)
```

8. 튜플

❖ 실습 5 (+ 연산자를 이용한 튜플간 결합)

```
>>> a = (1, 2, 3)
>>>
>>> b = (4, 5, 6)
>>>
>>> c = a + b
>>>
>>> a
(1, 2, 3)
>>>
>>> b
(4, 5, 6)
>>>
>>> c
(1, 2, 3, 4, 5, 6)
```

8. 튜플

❖ 실습 6 튜플간 값 교환

- 선언된 튜플의 값을 교환하시오.

```
>>> password = 'swordfish'
>>>
>>> icecream = 'tuttifrutti'
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```

8. 튜플

❖ 실습 7 (변경 불가능 테스트)

```
>>> a[0]
1
>>>
>>> a[0] = 7
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    a[0] = 7
TypeError: 'tuple' object does not support item assignment
```

8. 튜플

❖ 실습 8 (len() 함수)

```
>>> a = (1, 2, 3)
>>>
>>> len(a)
3
```

8. 튜플 - 패킹과 언패킹

❖ 실습 9 (튜플 패킹(Tuple Packing))

 # 패킹 : 여러 데이터를 튜플로 묶는 것

```
>>> a = (1, 2, 3)
>>>
>>> a
(1, 2, 3)
```

8. 튜플 - 패킹과 언패킹

❖ 실습 10 (튜플 언패킹(Tuple Unpacking))

 # 언패킹 : 튜플의 각 요소를 여러 개의 변수에 할당하는 것

```
>>> a = 1, 2, 3
>>>
>>> one, two, three = a
>>>
>>> one
1
>>> two
2
>>> three
3
>>> a
(1, 2, 3)
```

8. 튜플 - 패킹과 언패킹

❖ 실습 11 (언패킹 실패)

 # 튜플 요소 수와 언패킹할 요소의 수가 일치하지 않음

```
>>> a = 1, 2, 3
>>>
>>> one, two = a
Traceback (most recent call last):
  File "<pyshell#92>", line 1, in <module>
    one, two = a
ValueError: too many values to unpack (expected 2)
```

8. 튜플 - 패킹과 언패킹

❖ 실습 12 (언패킹을 이용한 변수 다중 할당)

```
>>> city, latitude, longitude = 'Seoul', 37.541, 126.986
>>>
>>> city
'Seoul'
>>>
>>> latitude          # 위도
37.541
>>>
>>> longitude         # 경도
126.986
```

'Seoul', 37.541, 126.986는 괄호 없이 만들어진 튜플

8. 튜플 - 메소드

❖ 실습 13 (index( )를 이용한 일치하는 데이터 찾기)

- 매개변수로 입력한 데이터와 일치하는 튜플 내 요소의 첨자를 알려준다.
- 찾고자 하는 데이터와 일치하는 요소가 없으면 에러가 발생한다.

```
>>> a = ('abc', 'def', 'ghi')
>>>
>>> a.index('def')
1
>>>
>>> a.index('jkl')
Traceback (most recent call last):
  File "<pyshell#116>", line 1, in <module>
    a.index('jkl')
ValueError: tuple.index(x): x not in tuple
```

8. 튜플 - 메소드

❖ 실습 14 (count( )를 이용한 요소 갯수 찾기)

- 매개변수로 입력한 데이터와 일치하는 요소가 몇 개 존재하는지 찾는다.

```
>>> a = (1, 100, 2, 100, 3, 100)
>>>
>>> a.count(100)
3
>>>
>>> a.count(200)
0
```

8. 튜플

❖ 튜플을 사용한 할당

- 변수 선언하고 할당하기

```
# 리스트와 튜플의 특이한 사용
[a, b] = [10, 20]
(c, d) = (10, 20)

# 출력합니다.
print("a:", a)
print("b:", b)
print("c:", c)
print("d:", d)
```

```
a: 10
b: 20
c: 10
d: 20
```

8. 튜플

- 괄호 생략
  - 괄호 생략해도 튜플로 인식할 수 있는 경우

```
# 괄호가 없는 튜플
tuple_test = 10, 20, 30, 40
print("# 괄호가 없는 튜플의 값과 자료형 출력")
print("tuple_test:", tuple_test)
print("type(tuple_test:)", type(tuple_test))
print()

# 괄호가 없는 튜플 활용
a, b, c = 10, 20, 30
print("# 괄호가 없는 튜플을 활용한 할당")
print("a:", a)
print("b:", b)
print("c:", c)
```

8. 튜플

❖ 튜플을 사용한 여러 값 리턴

- 여러 개 값 리턴하고 할당 가능

```
# 함수를 선언합니다.
def test():
    return (10, 20)

# 여러 개의 값을 리턴 받습니다.
a, b = test()

# 출력합니다.
print("a:", a)
print("b:", b)
```

9. 람다 : 함수의 매개변수로 함수 전달하기

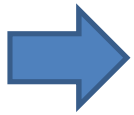
❖ 람다 표현식 사용하기 : 익명 함수(anonymous function)

- 식 형태로 되어 있다고 해서 람다 표현식(lambda expression)이라고 함
- 함수를 간편하게 작성
- 다른 함수의 인수로 넣을 때 주로 사용

```
--------------------------------
 구문   # 람다
--------------------------------

lambda <매개변수> : <리턴값>
```

- 숫자를 받은 뒤 10을 더해서 반환하는 함수

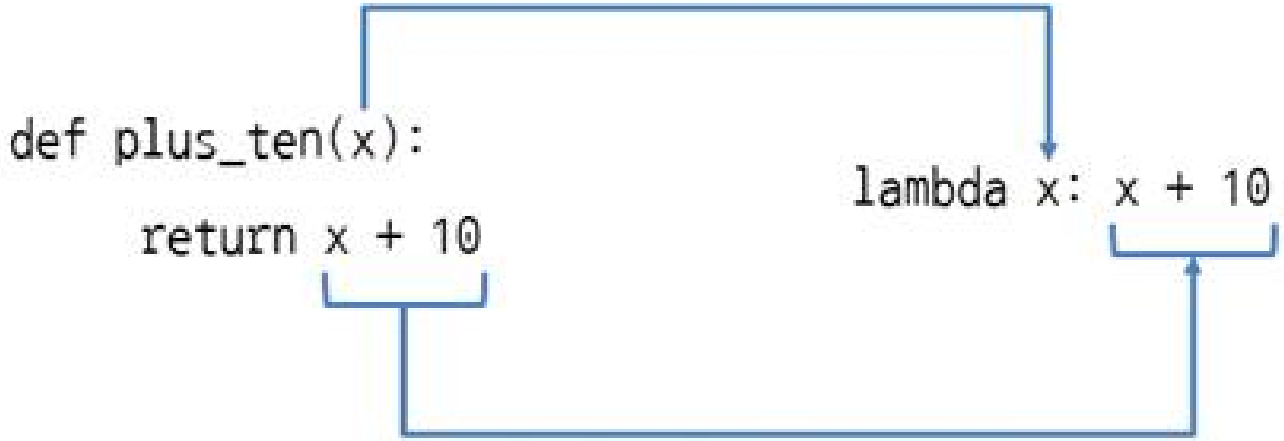
```
>>> def plus_ten(x):
        return x + 10

>>> plus_ten(1)
11
```



```
>>> lambda x: x + 10
<function <lambda> at 0x0000028FD304D2F0>
>>>
>>> plus_ten(1)
11
```

9. 람다 : 함수의 매개변수로 함수 전달하기

- 람다 표현식 사용하기 : 익명 함수(anonymous function)
  - def로 만든 함수와 람다 표현식



9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 람다 표현식 자체를 호출하기

- 람다 표현식은 변수에 할당하지 않고 자체를 바로 호출할 수 있다.
- lambda 매개변수들: 식)(인수들)

```
>>> (lambda x: x + 10)(1)
11
```

❖ 람다 표현식 안에서는 변수를 만들 수 없다.

- 변수가 필요한 코드일 경우에는 def로 함수를 작성하는 것이 좋다.

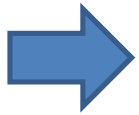
```
>>> (lambda x: y = 10; x + y)(1)
SyntaxError: invalid syntax
```

```
>>> y = 10
>>> (lambda x: x + y)(1)
11
```

9. 람다 : 함수의 매개변수로 함수 전달하기

- 람다 표현식을 인수로 사용하기
  - 함수의 인수 부분에서 간단하게 함수를 만든다
  - 대표적인 예가 map() 함수

```
>>> def plus_ten(x):
...     return x + 10
...
>>> list(map(plus_ten, [1, 2, 3]))
[11, 12, 13]
```



```
>>> list(map(lambda x: x + 10, [1, 2, 3]))
[11, 12, 13]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

- 람다 표현식으로 매개변수가 없는 함수 만들기
  - lambda 뒤에 아무것도 지정하지 않고 콜론(:) 사용
  - 콜론(:)뒤에는 반드시 반환할 값이 있어야 함(표현식은 값으로 평가되어야 한다)

```
>>> (lambda : 1)()

1

>>> x = 10

>>> (lambda : x)()

10
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 람다 표현식에 조건부 표현식 사용하기

- 조건부 표현식

```
lambda 매개변수들: 식 1 if 조건식 else 식 2
```

- map( )을 사용하여 리스트 a에서 3의 배수를 문자열로 변환하기

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> list(map(lambda x: str(x) if x % 3 == 0 else x, a))

[1, 2, '3', 4, 5, '6', 7, 8, '9', 10]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ map( ) 내장 함수

- 입력 집합(X)과 사상 함수(f)가 주어져 있을 때, Y = f(X)를 구한다.
- map( ) 함수는 두 개 이상의 인수를 받는다.
  - 첫번째 인수는 함수(f)이며
  - 두번째 인수는 입력집합(X)인 시퀀스 자료형(문자열, 리스트, 튜플 등)이어야 한다.
- 첫번째 인수인 함수는 입력 집합 수만큼의 인수를 받는다.

```
>>> def f(x) :
        return x * x

>>> X = [1, 2, 3, 4, 5]
>>>
>>> map(f, X)
<map object at 0x0000261929604E0>
>>>
>>> list(map(f, X))
[1, 4, 9, 16, 25]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ map( ) 내장 함수

- map( ) 함수의 결과로 반환되는 map 객체는 반복자이다.
- 값이 필요한 시점에 실제 계산이 이루어진다.

```
>>> def f(x) :
        print('calculationg..f', x)
        return x * x

>>> X = [1, 2, 3, 4, 5]
>>>
>>> m = map(f, X)
>>> next(m)
calculationg..f 1
1
>>> next(m)
calculationg..f 2
4
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ **map( ) 내장 함수 : 람다 함수에 적용 --- (1)**

```
>>> X = [1, 2, 3, 4, 5]
>>>
>>> Y = map(lambda a: a * a, X)
>>>
>>> Y
<map object at 0x00000261929D2AC8>
>>>
>>> list(Y)
[1, 4, 9, 16, 25]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ **map( ) 내장 함수 : 람다 함수에 적용 --- (2)**

```
>>> X = range(10)
>>>
>>> Y = map(lambda x : x * x + 4 * x + 5, X)
>>>
>>> list(Y)
[5, 10, 17, 26, 37, 50, 65, 82, 101, 122]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ map( ) 내장 함수 : 람다 함수에 적용 --- (3)

- map( ) 함수는 두 개 이상의 입력을 받을 수 있다.

```
>>> X = [1, 2, 3, 4, 5]
>>>
>>> Y = [6, 7, 8, 9, 10]
>>>
>>> Z = map(lambda x, y : x + y, X, Y)
>>>
>>> list(Z)
[7, 9, 11, 13, 15]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ map( ) 내장 함수 : 람다 함수에 적용 --- (4)

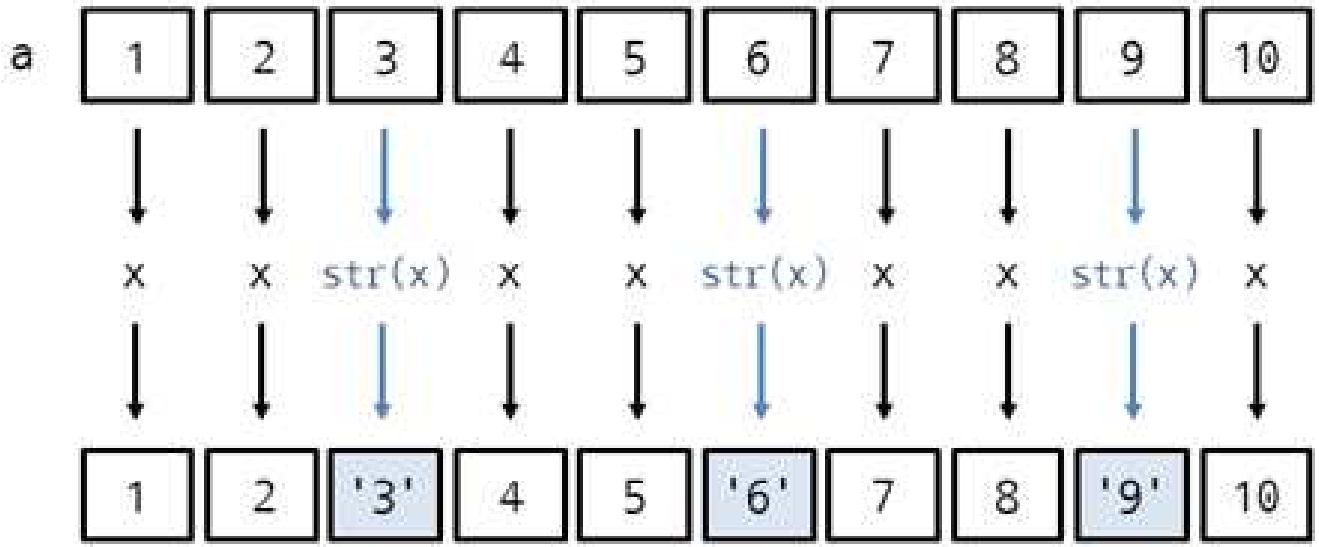
- map( ) 함수에 넘겨주는 함수가 이미 파이썬에서 정의한 연산일 경우에는 operator 모듈을 이용할 수 있다.

```
>>> import operator
>>>
>>> X = [1, 2, 3, 4, 5]
>>>
>>> Y = [6, 7, 8, 9, 10]
>>>
>>> Z = map(operator.add, X, Y)
>>>
>>> list(Z)
[7, 9, 11, 13, 15]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

- 람다 표현식에 조건부 표현식 사용하기
  - map( )에 람다 표현식 사용하기
  - 람다 표현식 안에서 조건부 표현식 if, else를 사용할 때는 :(콜론)을 붙이지 않음
  - 람다 표현식에서 if를 사용했다면 반드시 else를 사용

```
list(map(lambda x: str(x) if x % 3 == 0 else x, a))
```



9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 람다 표현식에 조건부 표현식 사용하기

- map( )에 람다 표현식 사용하기
- 람다 표현식 안에서 조건부 표현식 if, else를 사용할 때는 :(콜론)을 붙이지 않음
- 람다 표현식에서 if를 사용했다면 반드시 else를 사용
- 람다 표현식에서 if 만 사용하면 에러 발생

```
>>> list(map(lambda x: str(x) if x % 3 == 0, a))

SyntaxError: invalid syntax
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 람다 표현식에 조건부 표현식 사용하기

- 람다 표현식 안에서는 elif를 사용할 수 없다

```
lambda 매개변수들: 식 1 if 조건식 1 else 식 2 if 조건식 2 else 식 3
```

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(map(lambda x: str(x) if x == 1 else float(x) if x == 2 else x + 10, a))
['1', 2.0, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
>>> def f(x):
...     if x == 1:
...         return str(x)
...     elif x == 2:
...         return float(x)
...     else:
...         return x + 10
...
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(map(f, a))
['1', 2.0, 13, 14, 15, 16, 17, 18, 19, 20]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

- 람다 표현식 사용하기 : map( )에 여러 개의 객체 넣기
  - map은 리스트 등의 반복 가능한 객체를 여러 개 넣을 수도 있다.
  - 두 리스트의 요소를 곱해서 새 리스트를 만드는 예제

```
>>> a = [1, 2, 3, 4, 5]

>>> b = [2, 4, 6, 8, 10]

>>> list(map(lambda x, y: x * y, a, b))

[2, 8, 18, 32, 50]
```

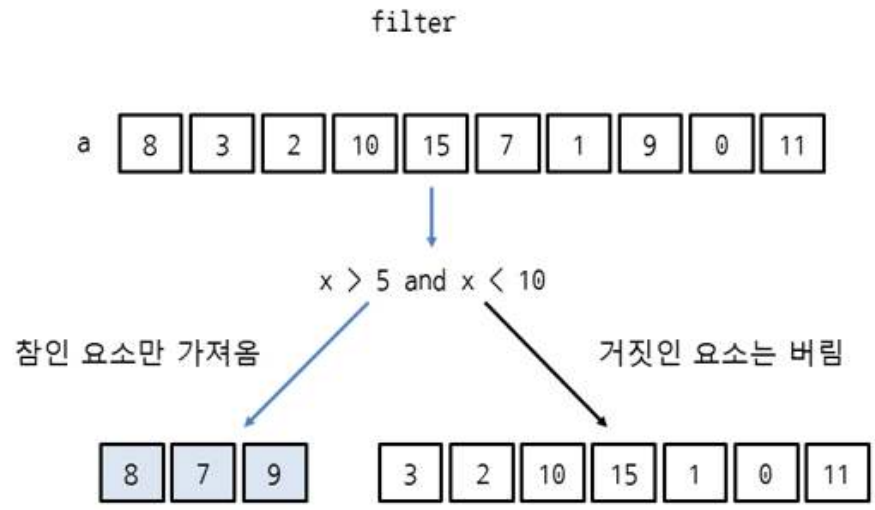
9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 람다 표현식 사용하기 : filter( )에 여러 개의 객체 넣기

- filter( )는 반복 가능한 객체에서 특정 조건에 맞는 참인 요소만 가져온다.
- filter( )에 지정한 함수의 반환값이 True일 때만 해당 요소를 가져온다.
- **리스트에서 5보다 크면서 10보다 작은 숫자를 가져오는 예제**

```
filter(함수, 반복가능한객체)
```

```
>>> def f(x):
...     return x > 5 and x < 10
...
>>> a = [8, 3, 2, 10, 15, 7, 1, 9, 0, 11]
>>> list(filter(f, a))
[8, 7, 9]
```



9. 람다 : 함수의 매개변수로 함수 전달하기

- 람다 표현식 사용하기 : map, filter와 리스트 표현식
  - 리스트(딕셔너리, 세트) 표현식으로 처리할 수 있는 경우에는 map, filter와 람다 표현식 대신 리스트 표현식을 사용하는 것이 좋다
  - 리스트 표현식이 좀 더 알아보기 쉽고 속도가 더 빠르다

```
>>> a = [8, 3, 2, 10, 15, 7, 1, 9, 0, 11]

>>> [i for i in a if i > 5 and i < 10]

[8, 7, 9]
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 함수의 매개변수로 함수 전달하기

```
# 매개변수로 받은 함수를 10번 호출하는 함수
def call_10_times(func):
    for i in range(10):
        func()

# 간단한 출력하는 함수
def print_hello():
    print("안녕하세요")

# 조합하기
call_10_times(print_hello)
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ map( ) 함수와 filter( ) 함수
- 함수를 매개변수로 전달하는 대표적인 표준 함수

```
map(<함수>, <리스트>)

filter(<함수>, <리스트>)
```

```
# 함수를 선언합니다.
def power(item):
    return item * item
def under_3(item):
    return item < 3

# 변수를 선언합니다.
list_input_a = [1, 2, 3, 4, 5]

# map() 함수를 사용합니다.
output_a = map(power, list_input_a)
print("# map() 함수의 실행 결과")
print("map(power, list_input_a):", output_a)
print("map(power, list_input_a):", list(output_a))
print()

# filter() 함수를 사용합니다.
output_b = filter(under_3, list_input_a)
print("# filter() 함수의 실행 결과")
print("filter(under_3, output_b):", output_b)
print("filter(under_3, output_b):", list(output_b))
```

9. 람다 : 함수의 매개변수로 함수 전달하기

- 람다는 '간단한 함수를 쉽게 선언' 하는 방법이다
- 람다를 활용한 map() 함수와 filter() 함수

```
----------------------------------
람다
----------------------------------
lambda <매개변수> : <리턴값>
```

```
# 함수를 선언합니다.
power = lambda x: x * x
under_3 = lambda x: x < 3

# 변수를 선언합니다.
list_input_a = [1, 2, 3, 4, 5]

# map() 함수를 사용합니다.
output_a = map(power, list_input_a)
print("# map() 함수의 실행 결과")
print("map(power, list_input_a):", output_a)
print("map(power, list_input_a):", list(output_a))
print()

# filter() 함수를 사용합니다.
output_b = filter(under_3, list_input_a)
print("# filter() 함수의 실행 결과")
print("filter(under_3, output_b):", output_b)
print("filter(under_3, output_b):", list(output_b))
```

9. 람다 : 함수의 매개변수로 함수 전달하기

- 람다는 '간단한 함수를 쉽게 선언' 하는 방법이다
- 람다를 활용한 map( ) 함수와 filter( ) 함수

```
# 변수를 선언합니다.
list_input_a = [1, 2, 3, 4, 5]

# map() 함수를 사용합니다.
output_a = map(lambda x: x * x, list_input_a)
print("# map() 함수의 실행 결과")
print("map(power, list_input_a):", output_a)
print("map(power, list_input_a):", list(output_a))
print()

# filter() 함수를 사용합니다.
output_b = filter(lambda x: x < 3, list_input_a)
print("# filter() 함수의 실행 결과")
print("filter(under_3, output_b):", output_b)
print("filter(under_3, output_b):", list(output_b))
```

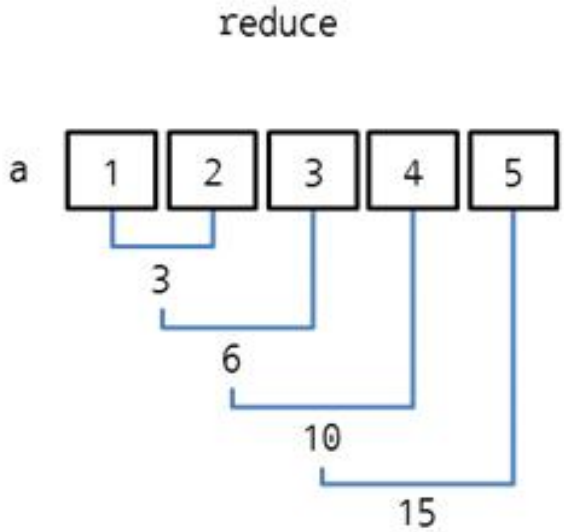
9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 람다 표현식 사용하기 : reduce 사용하기

- reduce는 반복 가능한 객체의 각 요소를 지정된 함수로 처리한 뒤 이전 결과와 누적해서 반환하는 함수
- 리스트에 저장된 요소를 순서대로 더한 뒤 누적된 결과를 반환하는 예제

```
from functools import reduce
reduce(함수, 반복가능한객체)
```

```
>>> def f(x, y):
...     return x + y
...
>>> a = [1, 2, 3, 4, 5]
>>> from functools import reduce
>>> reduce(f, a)
15
```



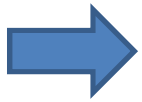
9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 람다 표현식 사용하기 : reduce 사용하기

- reduce는 반복 가능한 객체의 각 요소를 지정된 함수로 처리한 뒤 이전 결과와 누적해서 반환하는 함수
- 리스트에 저장된 요소를 순서대로 더한 뒤 누적된 결과를 반환하는 예제

```
from functools import reduce
reduce(함수, 반복가능한객체)
```

```
>>> def f(x, y):
...     return x + y
...
>>> a = [1, 2, 3, 4, 5]
>>> from functools import reduce
>>> reduce(f, a)
15
```



```
>>> a = [1, 2, 3, 4, 5]
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, a)
15
```

9. 람다 : 함수의 매개변수로 함수 전달하기

❖ 람다 표현식 사용하기 : reduce와 리스트 표현식

- for, while 반복문으로 처리할 수 있는 경우에도 reduce 대신, for, while을 사용하는 것
  이 코드를 한 눈에 알아보기 쉽기 때문에 편리하다

```
>>> a = [1, 2, 3, 4, 5]

>>> x = a[0]

>>> for i in range(len(a) - 1):

...     x = x + a[i + 1]

...

>>> x

15
```

실습 1

❖ 아래 제시한 조건에 알맞은 프로그램을 작성하시오

[조건] : 세균 번식

한 마리의 새로 생겨난 세균은 그 다음날과 이틀 후 각각 한 마리의 세균을 번식한 후 죽는다.

현재 한 마리의 새로운 세균이 있다.

오늘 부터 n 일째에 생겨날 세균 수를 계산하는 프로그램을 작성하시오.

단, n 일은 사용자로부터 입력 받는다.

실습 2

❖ 아래 제시한 조건에 알맞은 프로그램을 작성하시오

[조건] : 반장 선거

10명으로 구성된 학급에서 반장 선거가 있다.

반 학생 10명 모두가 후보이며 각각 1에서 10 사이의 후보 기호를 받는다.

반 학생들의 투표 내용을 인자로 전달받아 후보 별 득표수를 계산하여 반환하는 함수를 작성한다.

주 프로그램은 10명의 투표 내용을 기호 순으로 입력 받은 다음 함수의 반환값으로부터 후보 별 득표 수를 출력하고, 최다득표자의 기호를 출력한다. (단, 무효표, 기권표는 없다고 전제한다.)

실습 3

❖ 아래 제시한 조건에 알맞은 프로그램을 작성하시오

[조건] : 도서 대출 시스템

10권의 도서가 있을 때, 일련번호는 1~10이다.

프로그램을 실행하면 이름을 입력받고 해당이름이 있을경우 빌릴 책의 숫자를 입력한다. 빌리는 책이 3권이 되거나 q를 입력하면 대출을 멈춘다.

대출자와 대출리스트는 {Key : list[Value]}로 보관하라.

```
book = {1:"도레미제라블", 2:"위대한 소츠비", 3:"어른왕자", 4:"아이와 산", 5:"곤충 농장", 6: "춘추전국시대", 7:"분노의 사과", 8:"강철북", 9:"첫번째 잎새", 10:"그리고 아무말이나 했다"}
```

실습 4

❖ 아래 제시한 조건에 알맞은 프로그램을 작성하시오

[조건] : 무인 커피 주문

입력받은 고객 이름과 커피 종류를 출력하는 함수 프로그램 작성
(단, 고객 이름을 작성하지 않으면 1, 2, .. 등 숫자를 순서대로 출력)

```
커피종류를 입력하세요 : 아이스 아메리카노
고객명을 입력하세요 :
1 고객님, 주문하신 아이스 아메리카노 나왔습니다.

커피종류를 입력하세요 : 카페라떼
고객명을 입력하세요 : 하늘
하늘 고객님, 주문하신 카페라떼 나왔습니다.

커피종류를 입력하세요 : 홍차
고객명을 입력하세요 :
3 고객님, 주문하신 홍차 나왔습니다.

커피종류를 입력하세요 : 마감
>>>
```

Thank You !

파이썬