

으뜸 파이썬



3강 제어와 함수

- 본 강의노트는 으뜸 파이썬(박동규, 강영민 著) 1판의 강의자료를 활용하여 교양수업에 맞게 편집되었습니다.

순차문 sequential statements

- 순차적 구조
 - 먼저 나타나는 코드가 먼저 실행되는 구조

코드 3-1 : 순차적 실행 구조를 이용한 변수의 덧셈

seq_test.py

num = 100

print('num = ', num) # 100이 출력됨

num = num + 100

print('num = ', num) # num에 100이 더해져 200이 출력됨

num = num + 100

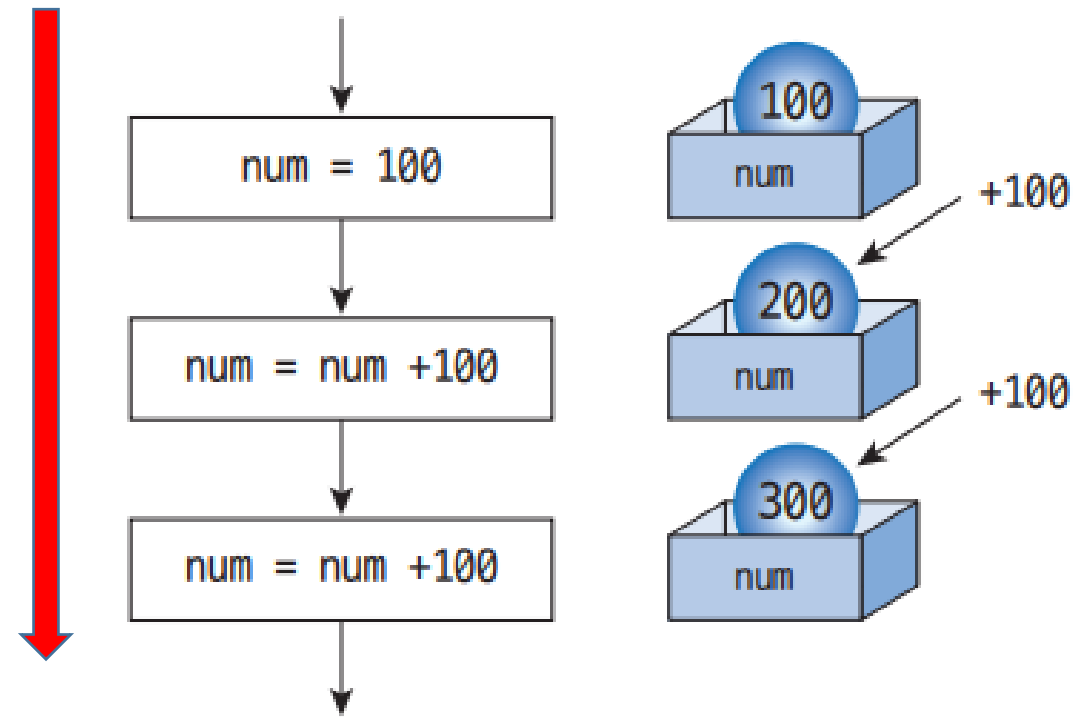
print('num = ', num) # num에 다시 100이 더해져 300이 출력됨

실행결과

num = 100

num = 200

num = 300



[그림 3-1] [코드 3-1]의 순차적인 실행 순서와 변수 값의 변화

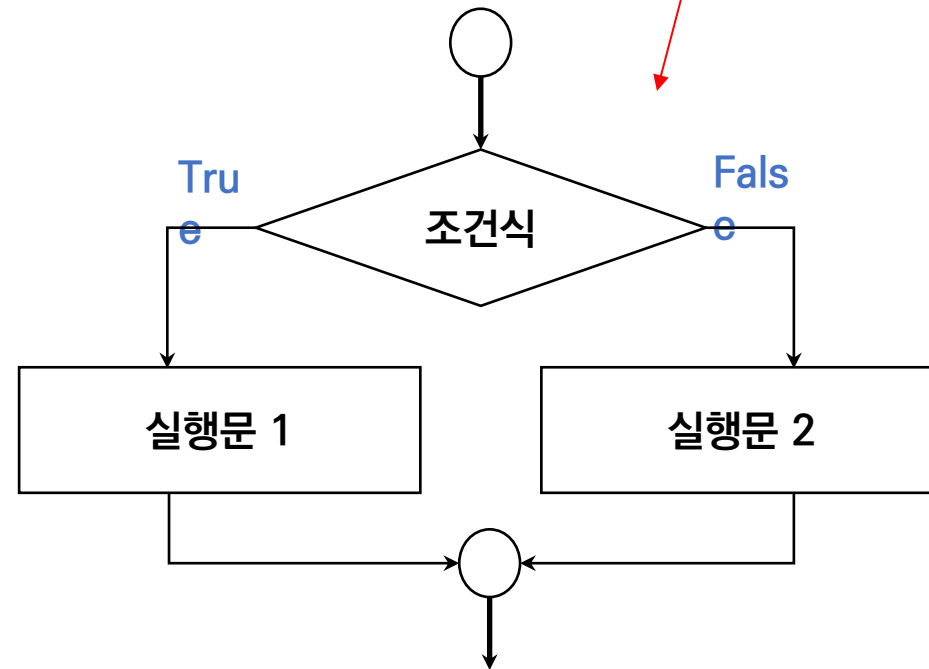
순차문 sequential statements 이외의 흐름문 flow statements

- 제어문 control statements
 - 프로그램의 흐름을 제어하는 역할
 - 조건문 conditional statements
 - if 문, if-else 문
 - 반복문
 - for 문, while 문
 - 반복문의 흐름 변경
 - break, continue

if 조건문

- 조건문 conditional statements
 - 실행을 달리하는 여러 개의 실행문이 있음
 - 특정한 조건에 따라서 실행됨
 - 조건식은 True 혹은 False를 반환

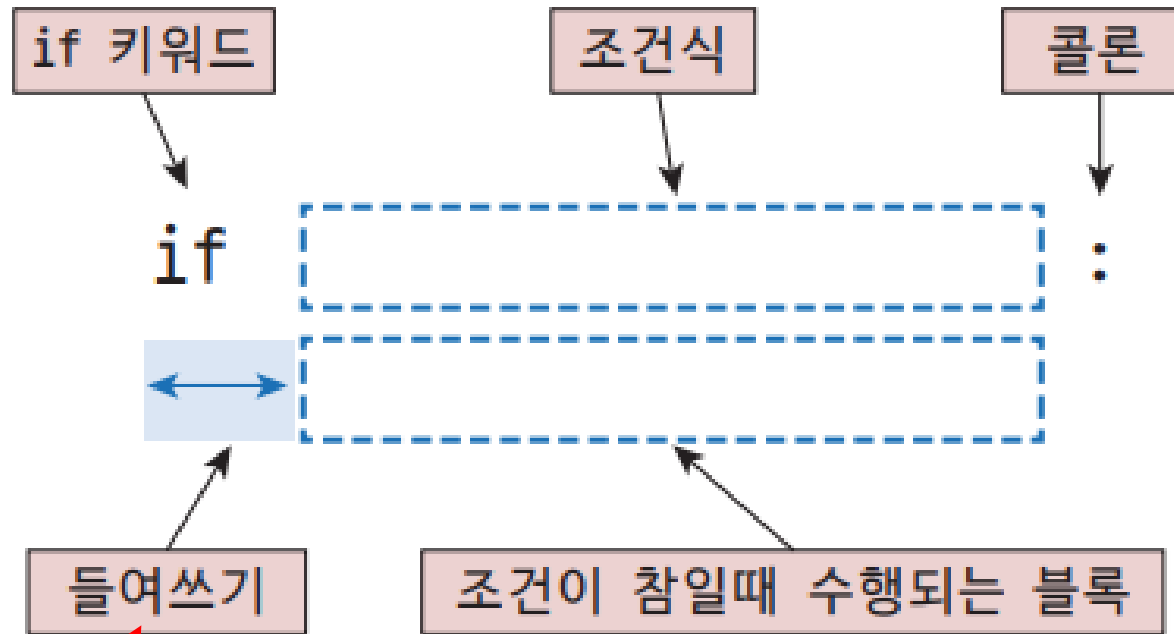
흐름도 :
이와 같은 작업의 흐름을 나타내는 그림을 흐름도라고 한다.



조건문의 구체적 상황

- 상황 1 : 나이가 20세 미만이면 '청소년 할인'을 출력하는 기능
- 상황 2 : 1000 걸음 이상을 걸으면 '목표 달성'을 출력하는 기능
- 상황 3 : 시간이 12시가 안되면 '오전입니다', 12시 이후이면 '오후입니다'를 출력하는 기능

if 조건문의 사용법



[그림 3-3] if 조건문의 사용법

파이썬은 들여쓰기가 아주 중요하다

상황 1 :
나이(age)가 20세 미만이면 '청소년
할인'을 출력



```
if age < 20 :  
    print('청소년 할인')
```

상황 2 :
걸음(walk_count)이 1000 이상이면
'목표 달성' 출력



```
if walk_count >= 1000 :  
    print('목표 달성')
```

- (상황 1) - 콜론(:)앞에 나타나는 조건문 절에서 < 연산자를 이용해 나이 (age)가 20세 미만인 경우에만 print('청소년 할인')이라는 코드를 실행
- (상황 2) - 조건문 절에서 >= 연산자를 이용해 걸음(walk_count)이 1,000 이상이 되면 print('목표 달성')이라는 코드를 실행

상황 1

코드 3-2 : if 조건문을 이용한 출력기능(조건을 만족하는 경우)

```
if_youth_discount.py
```

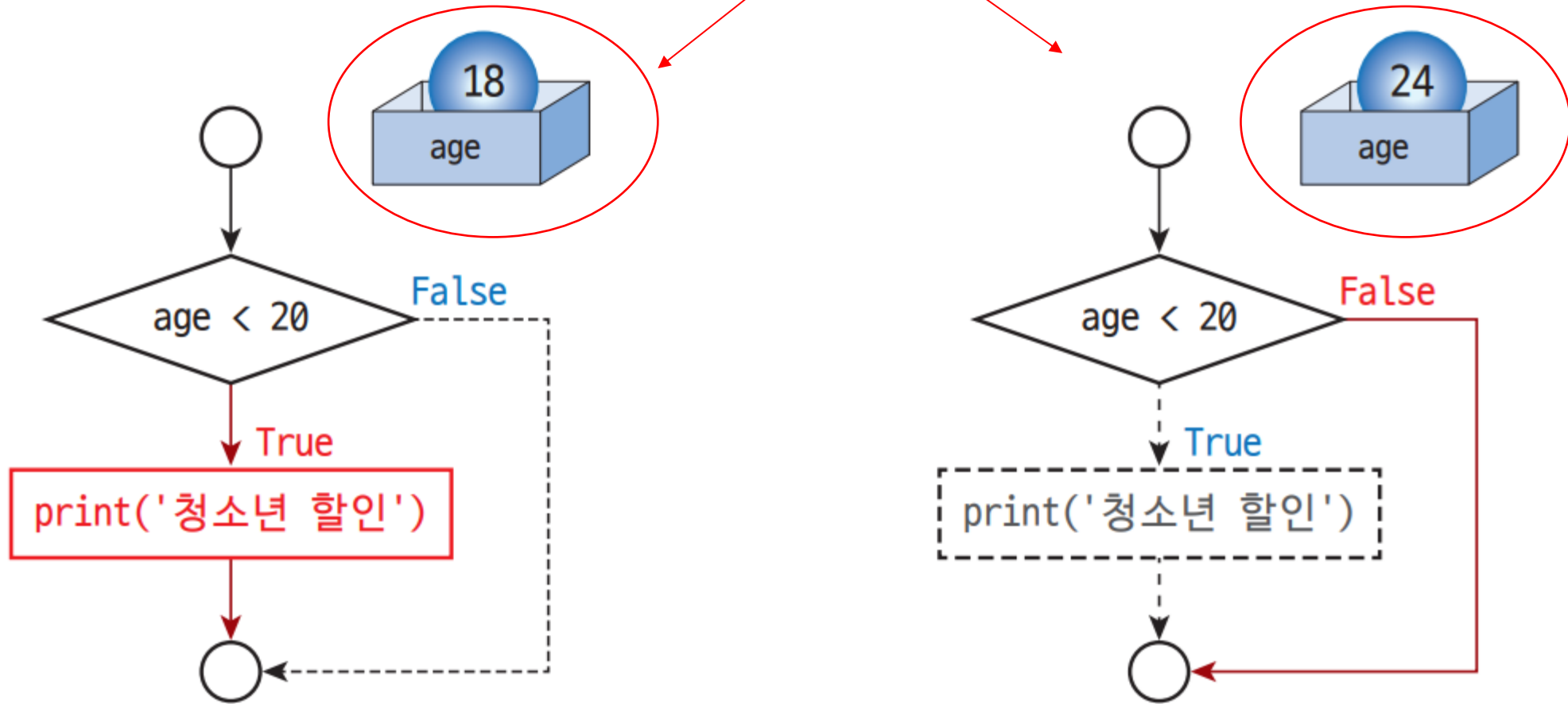
```
age = 18          # age가 20 미만의 값
```

```
if age < 20:      # age < 20 조건식의 결과는 True임  
    print('청소년 할인')
```

실행결과

청소년 할인

age 변수의 값에 따라 다른 흐름으로 이동



[그림 3-6] 변수 age의 변화에 따라 제어되는 [코드 3-2]의 실행 흐름



LAB 3-1 : if 문의 사용법

1. 게임 사용자의 게임점수(game_score)가 1000점 이상이면 '당신은 고수입니다'를 출력하는 프로그램을 if 문을 이용하여 작성하시오. 이때 다음과 같이 game_score값을 화면에 출력하여라. game_score에 800점, 1300점을 각각 입력하여 출력문을 확인하시오.

```
game_score = 800
```

혹은

```
game_score = 1300
```

당신은 고수입니다

2. num_a와 num_b에 할당된 값이 같으면 '두 값이 일치합니다.'를 출력하는 프로그램을 if 문을 이용하여 작성하시오. num_a와 num_b에 각각 100과 200이 할당되어 있는 경우와 num_a와 num_b에 300과 300이 할당되어 있는 경우에 대하여 각각 코드를 작성하고 출력문을 확인하시오.

```
num_a = 100, num_b = 200
```

혹은

```
num_a = 300, num_b = 300
```

두 값이 일치합니다.

조건문과 블록

- 블록 **block**
 - 어떤 조건을 만족하는 경우에 특정한 코드를 선택적으로 실행하는 구조
 - 이때 실행될 코드 덩어리
 - 블록은 반드시 들여쓰기를 해야한다

코드 3-5 : 들여쓰기 없는 print() 문

if_youth_error.py

age = 18

if age < 20:

print('청소년 할인') # 들여쓰기 없는 print()문

들여쓰기(indentation) 블록이 필요하다는 의미

실행결과

IndentationError: expected an indented block

파이썬의 들여쓰기

- 파이썬은 들여쓰기가 매우 중요한 의미를 가지는 프로그래밍 언어
 - C/C++이나 Java, Pascal 등 전통적인 프로그래밍 언어와 다른 특징

[들여쓰기 코드 1] 조건을 만족하는 경우	[들여쓰기 코드 2] 조건을 만족하지 않는 경우
<pre>age = 18 if age < 20: print('청소년 할인') print('입장을 환영합니다')</pre>	<pre>age = 24 if age < 20: print('청소년 할인') print('입장을 환영합니다')</pre>
수행결과 - if문 내부와 외부 print문이 수행됨	수행결과 - if문 외부의 print문만 수행됨
청소년 할인 입장을 환영합니다	입장을 환영합니다

[들여쓰기 코드 3] 조건을 만족하는 경우	[들여쓰기 코드 4] 조건을 만족하지 않는 경우
<p>age = 18</p> <pre> if age < 20: print('나이', age) print('청소년 환영') print('청소년 할인') </pre>	<p>age = 24</p> <pre> if age < 20: print('나이', age) print('청소년 환영') print('청소년 할인') </pre>
<p>수행결과</p> <p>- 들여쓰기 블록 전체가 수행됨</p>	<p>수행결과</p> <p>- 들여쓰기 블록 전체가 수행되지 않음</p>
<p>나이 18</p> <p>청소년 환영</p> <p>청소년 할인</p>	

- 블록은 흔히 코드 블록이라고도 함
- 소스 코드에서 함께 묶을 수 있는 코드의 덩어리를 말한다
- 파이썬은 if문 다음에 :(콜론)이 나오면 다음에 들여쓰기 코드 블록이 나와야 하며 else, elif, for, while, def, class 등에서도 코드 블록이 사용됨

- [들여쓰기 코드 5]와 같이 동일한 블록에 대해 들여쓰기의 칸 수가 일정하지 않으면 “IndentationError: unexpected indent” 라는 들여쓰기 오류가 발생
- 동일한 코드 블록에서는 들여쓰기의 칸 수를 반드시 일치시켜야 함
- 스페이스space 4칸을 권장

들여쓰기 코드 5 : 들여쓰기가 잘못된 경우

```
age = 18
```

```
if age < 20:
```

```
    print('나이', age)
```

```
    print('청소년 환영')
```

```
print('청소년 할인')
```

수행결과

IndentationError: unexpected indent 오류 발생

3의 배수 판단

코드 3-6 : 3의 배수를 판단하기 위한 모듈로 연산과 조건문

if_modulo1.py

```
number = int(input('정수를 입력하세요 : ')) # 입력값을 정수형으로 변환
if number % 3 == 0:                          # 모듈로 3의 값이 0이면 3의 배수임
    print(number, '은(는) 3의 배수입니다.')
```

실행결과

정수를 입력하세요 : 15
15 은(는) 3의 배수입니다.

실행결과

정수를 입력하세요 : 16

3과 5의 배수 판단

코드 3-7 : 3과 5의 배수를 판단하기 위한 모듈로 연산과 and 조건문

if_modulo2.py

```
number = int(input('정수를 입력하세요 : '))  
if number % 3 == 0 and (number % 5) == 0:  
    print(number, '은(는) 3의 배수이면서 5의 배수입니다.')
```

실행결과

정수를 입력하세요 : 15

15 은(는) 3의 배수이면서 5의 배수입니다.



LAB 3-2 : 변수와 if 조건식 사용하기

1. 1에서 100 사이의 임의의 정수 n 을 입력받아서 1) n 을 화면에 출력한 후, 2) n 이 짝수이면 "...은(는) 짝수입니다."를 다음과 같이 출력하는 프로그램을 작성하여라.

정수를 입력하세요 : 50

$n = 50$

50 은(는) 짝수입니다.

또는

정수를 입력하세요 : 75

$n = 75$

if-else 조건문

상황 3 : 24시 체계→12시 체계

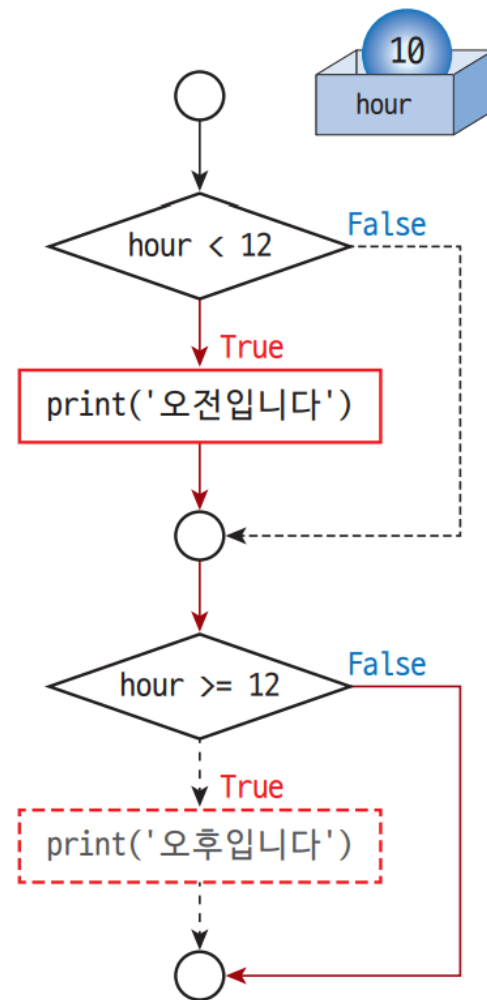
코드 3-8 : if 문을 이용한 '오전' 혹은 '오후'의 출력기능

if_hour_test.py

```
hour = 10
if hour < 12:
    print('오전입니다.')
if hour >= 12:
    print('오후입니다.')
```

실행결과

오전입니다.



[그림 3-10] hour 값이 10일 때 코드 if_hour_test.py의 흐름도

if-else 문을 이용한 출력 : 배타적 관계

코드 3-9 : if-else 문을 이용한 '오전' 혹은 '오후'의 출력 기능

if_else_hour_test.py

```
hour = 10
```

```
if hour < 12:
```

```
    print('오전입니다.')
```

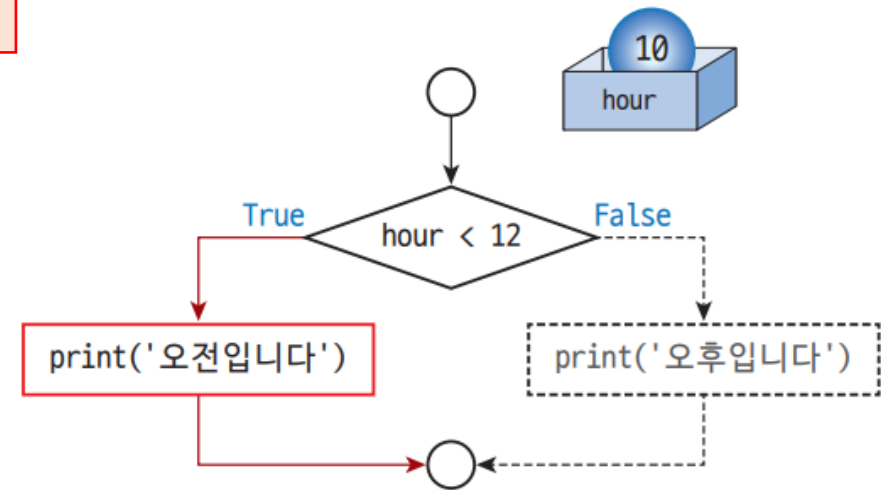
```
else:
```

```
    print('오후입니다.')
```

배타적인 관계

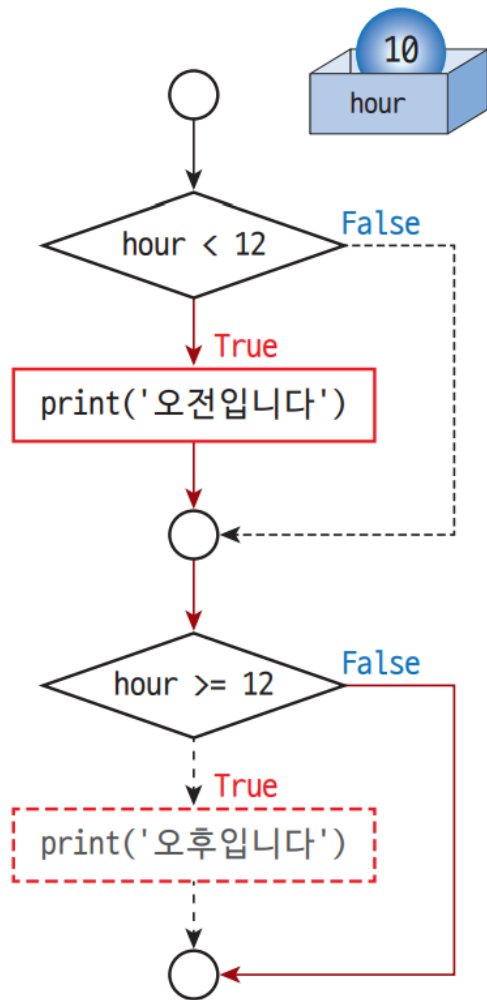
실행결과

오전입니다.

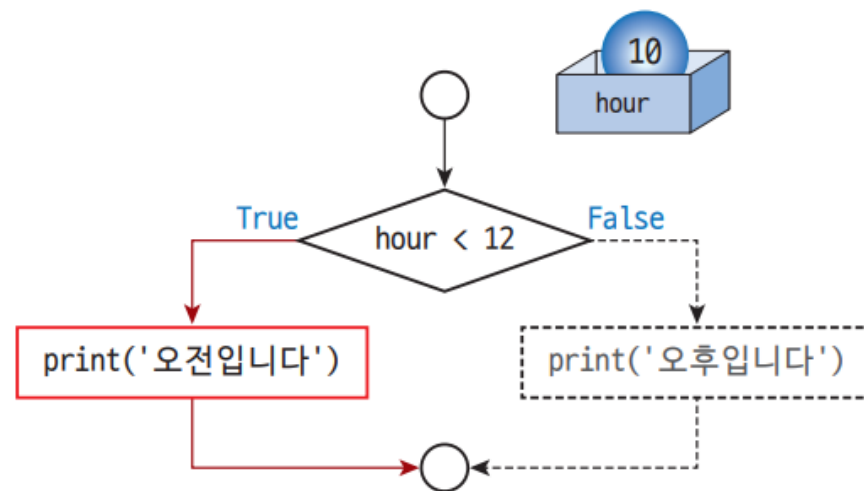


[그림 3-11] hour 값이 10일 때 if_else_hour_test.py 코드의 흐름도

비교 : 어느쪽이 명확해 보이세요?



[그림 3-10] hour 값이 10일 때 코드 `if_hour_test.py`의 흐름도



[그림 3-11] hour 값이 10일 때 `if_else_hour_test.py` 코드의 흐름도



LAB 3-3 : if 조건문의 응용

1. 게임 사용자의 게임점수(game_score)을 입력받아서 1000점 이상이면 '고수입니다.'를 출력하고 1000점 미만이면 '입문자입니다.'를 출력하는 프로그램을 if-else 문을 이용하여 작성하시오.

게임점수를 입력하시오 : 800

```
game_score = 800
```

입문자입니다.

혹은

게임점수를 입력하시오 : 1300

```
game_score = 1300
```

고수입니다.

복합 조건식

- 더 정교한 조건을 걸어주기 위해 조건 연산자와 논리 연산자를 조합
- 모두 부울 값(True, False)을 반환한다는 공통점이 있음

비교 연산 (<, <=, >=, >, !=, ==)

부울 연산 (True, False)

논리 연산 (and, or, not), in, not in, is, ...

- 비교 연산자는 연산자 왼쪽의 값과 오른쪽의 값이 해당 연산자의 조건을 만족할 시 True 아니면 False를 반환

대화창 실습 : 조건식 실습

```
>>> 0 < 10      # 조건식이 참, 'True' 출력
```

```
True
```

```
>>> 4 > 10      # 조건식이 거짓, 'False' 출력
```

```
False
```

```
>>> 3 <= 10     # 조건식이 참, 'True' 출력
```

```
True
```

```
>>> 15 >= 10    # 조건식이 참, 'True' 출력
```

```
True
```

```
>>> 1 == 2      # 조건식이 거짓, 'False' 출력
```

```
False
```

```
>>> True or False # 조건식이 참, 'True' 출력
```

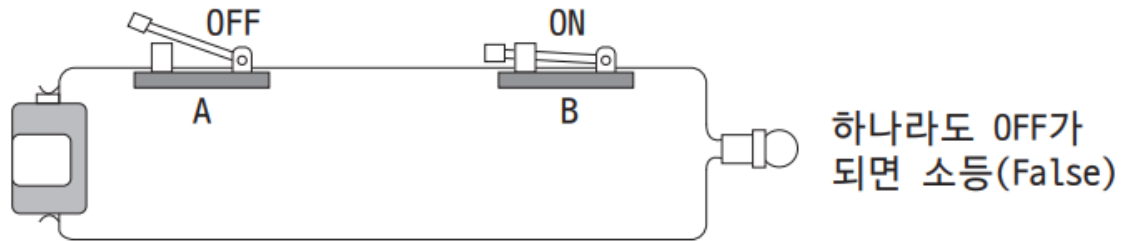
```
True
```

```
>>> True and False # 조건식이 거짓, 'False' 출력
```

```
False
```

논리 연산 and

- 입력 값 중에서 **False** 상태에 영향을 받는 특징이 있음



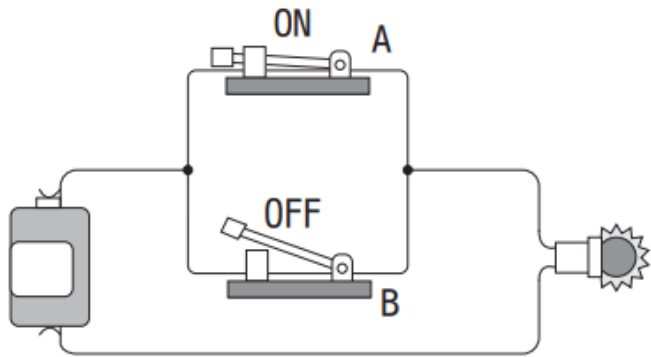
and

입력		출력
A	B	
True	True	True
True	False	False
False	True	False
False	False	False

[그림 3-13] and 연산을 수행하는 직렬 회로도와 논리 연산의 결과

논리 연산 or

- 출력 값이 입력 값의 True 상태에 영향을 받음



하나라도 ON이
되면 점등(True)

or

입력		출력
A	B	
True	True	True
True	False	True
False	True	True
False	False	False

[그림 3-14] or 연산을 수행하는 병렬 회로도와 논리 연산의 결과

- 변수 a와 b에 저장된 값은 각각 10, 14이므로 두 조건문의 조건식이 모두 참(True)
- 실행 결과는 두 개의 print()문이 모두 실행됨

코드 3-13 : and와 or 조건문의 사용법

if_and_or_test.py

```
a = 10
```

```
b = 14                # 13으로 수정하면 첫 번째 조건문을 만족하지 않음
```

```
if (a % 2 == 0) and (b % 2 == 0):    # 첫 번째 조건문
```

```
    print('두 수 모두 짝수입니다.')
```

```
if (a % 2 == 0) or (b % 2 == 0):    # 두 번째 조건문
```

```
    print('두 수 중 하나 이상이 짝수입니다.')
```

실행결과

두 수 모두 짝수입니다.

두 수 중 하나 이상이 짝수입니다.

- (b = 13으로 수정하면) 첫 번째 조건문의 조건식을 만족하지 못해 두 번째 조건문 내부의 print문만 실행됨

실행결과

두 수 중 하나 이상이 짝수입니다.



LAB 3-4 : 복합 조건식의 이해

1. and 연산자를 사용하여 num 변수가 1과 10사이의 값을 가지면 True를 출력하는 부울식을 완성하여라.

```
>>> num = 2
```

```
>>> _____
```

```
True
```

2. and 연산자를 사용하여 age가 10보다 크고 19보다 작으면 '청소년입니다.'를 출력하는 부울식을 작성하여라. 그리고 age에 9와 12를 넣어서 그 결과를 다음과 같이 확인하여라.

```
age = 10
```

혹은

```
age = 12
```

```
청소년입니다.
```

복합 조건식으로 윤년 검사하기

- 윤년^{leap year}의 규칙

- 1) 연수가 4로 나누어 떨어지는 해는 윤년으로 한다(예를 들어 1992년, 2004년등)
- 2) 연수가 1의 조건에 만족함에도 100으로 나누어 떨어지는 해는 평년으로 한다 (예를 들어 1900년, 2100년, 2200년, 2300년 등)
- 3) 연수가 2의 조건에 만족함에도 400으로 나누어 떨어지는 해는 윤년으로 한다(예를 들어 2000년, 2400년등)

if-elif-else 문

- 많은 if문을 사용해 점수대별로 등급을 나누는 학점 산출기
 - 여러 개의 if문과 if문내의 and 조건을 적용하여 문제를 해결

점수	등급
100점 ~ 90점 이상	A
90점 미만 ~ 80점 이상	B
80점 미만 ~ 70점 이상	C
70점 미만 ~ 60점 이상	D
60점 미만	F

코드 3-15 : 'A','B','C','D','F' 등급 계산을 위한 if 문

if_grade1.py

```
score = int(input('점수를 입력하세요 : '))  
if score >= 90 :           # 90 이상인 경우 'A'  
    grade = 'A'  
if score < 90 and score >= 80 : # 90 미만 80 이상인 경우 'B'  
    grade = 'B'  
if score < 80 and score >= 70 : # 80 미만 70 이상인 경우 'C'  
    grade = 'C'  
if score < 70 and score >= 60 : # 70 미만 60 이상인 경우 'D'  
    grade = 'D'  
if score < 60 :           # 60 미만인 경우 'F'  
    grade = 'F'  
print('당신의 등급은 :', grade)
```

실행결과

점수를 입력하세요 : 77

당신의 등급은 : C

- 이전에 살펴본 간단한 if문보다 복잡하고 코드를 읽기가 어려워졌음
- 세 번째 조건문에서 다음과 같은 잘못된 조건식이 들어가도 한눈에 오류를 파악하기가 힘들

```
if score < 80 and score > 70 : # if score < 80 and score >= 70 : 의 오류 코드
```

- 각각의 if문의 의미를 하나하나 파악해야하기 때문에 오류의 가능성이 높아짐
- 이를 해결하기 위하여 다음과 같이 if-else 문을 적용

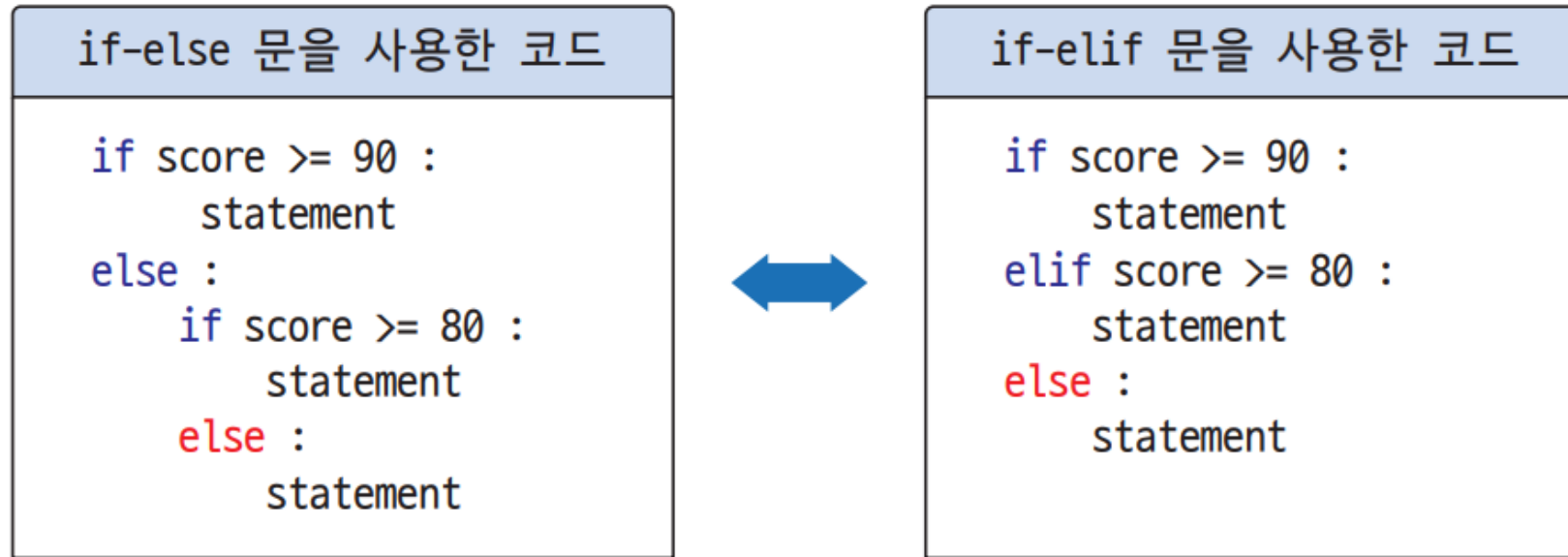
코드 3-16 : 'A','B','C','D','F' 등급 계산을 위한 복합 if 문

if_grade2.py

```
score = int(input('점수를 입력하세요 : '))
if score >= 90:           # 90 이상인 경우 'A'
    grade = 'A'
else:
    if score >= 80 :      # 90 미만 80 이상인 경우 'B'
        grade = 'B'
    else:
        if score >= 70:  # 80 미만 70 이상인 경우 'C'
            grade = 'C'
        else:
            if score >= 60:  # 70 미만 60 이상인 경우 'D'
                grade = 'D'
            else:          # 60 미만인 경우 'F'
                grade = 'F'
print('당신의 등급은 :', grade)
```

- 이전의 if문으로만 구성되어있던 [코드 3-15]보다는 읽기가 편해짐
- 오류의 가능성도 이전에 비해서 줄어듦
- if-else가 조건을 2개밖에 나타낼 수밖에 없기 때문에 가독성은 여전히 떨어짐
- 조건이 여러 개인 경우 if문에서 else문까지 가기 전에 조건을 더 걸어줄 수는 없을까? -> elif문 사용하기

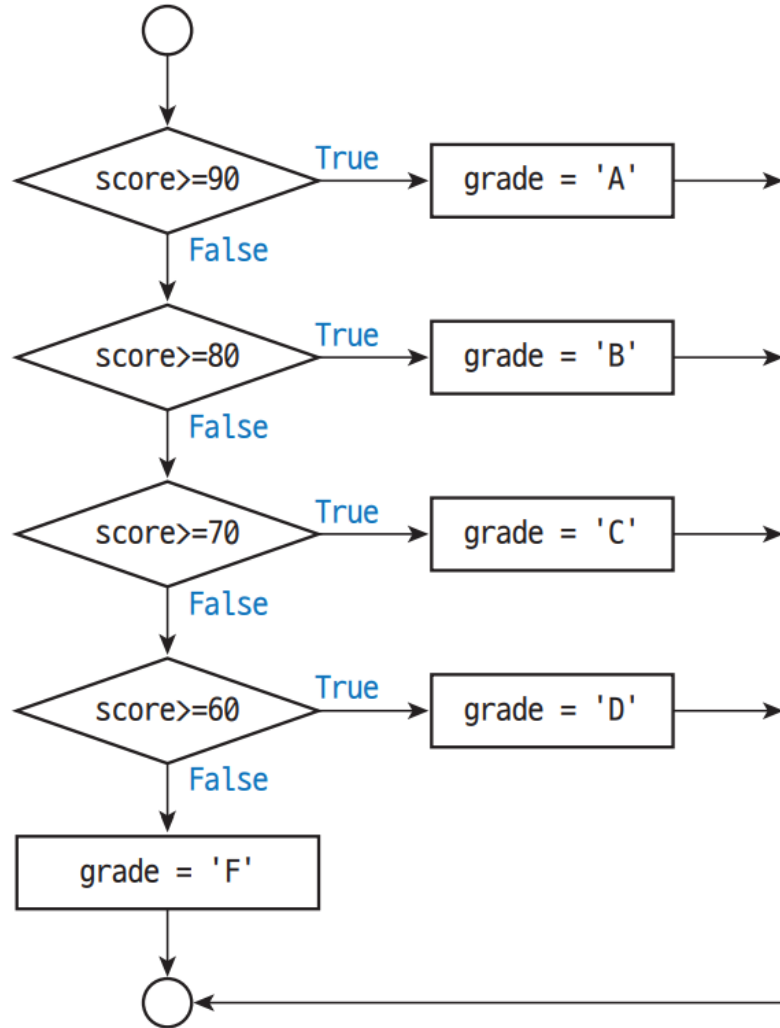
if-else문과 elif문의 비교



[그림 3-15] 복합 if-else 문과 if-elif 문을 사용한 동일한 코드

- 왼쪽과 오른쪽의 코드는 동일
- 오른쪽의 코드가 들여쓰기도 더 적게하고 줄의 수도 더 줄어들어 코드를 이해하기가 더 편리해짐

if-elif-else 문의 실행 흐름도



[그림 3-16] if-elif-else 문의 실행 흐름도



LAB 3-5 : if-elif-else 문을 사용한 다중 조건식

1. 사용자로부터 자동차의 속도(speed)를 km/h 단위의 정수로 입력받도록 하자. 자동차의 속도가 100km/h 이상이면 '고속', 100km/h 미만 60km/h 이상이면 '중속', 60km/h 미만이면 '저속'을 출력하는 프로그램을 if-elif-else 문을 이용하여 작성하여라.

자동차의 속도를 입력하세요(단위 : km/h): 13

저속

혹은

자동차의 속도를 입력하세요(단위 : km/h): 130

고속

for 반복문

- 특정한 작업을 여러 번 되풀이해서 수행하고 싶을 경우 사용
- 반복문에는 for 문과 while 문 두 종류가 있음
- for 문은 반복의 횟수가 미리 정해져 있는 경우, while 문은 반복 횟수는 알지 못하지만 반복하는 조건이 명확한 경우에 사용

- 반복문을 사용하지 않고 5번 반복해 출력하려면 다음과 같이 print() 함수를 5번에 걸쳐 사용
- 1000회 반복시 매우 비효율적임

코드 3-18 : print() 함수의 호출을 통한 반복적 수행

print_welcome.py

```
print('Welcome to everyone!!')  
print('Welcome to everyone!!')  
print('Welcome to everyone!!')  
print('Welcome to everyone!!')  
print('Welcome to everyone!!')
```

실행결과

```
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!
```

- range() 함수는 특정한 구간의 정수 열sequence을 반복해서 생성함
- for 문에서 순환을 위한 용도이다

```
for i in range( n ):
```



새로운 변수



반복 실행 횟수

횟수가 정해져있거나 1씩 증가하는 숫자를 써야할 때 사용
i는 0부터 n-1까지 증가함

[그림 3-18] for in range() 구문의 사용법

코드 3-19 : for 문을 이용한 반복적 수행

```
print_welcome_with_for1.py  
  
for i in range(5):  
    print('Welcome to everyone!!')
```

실행결과

```
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!
```

10번 반복시 range() 괄호 내의 값만 10으로 고쳐주면 됨

```
for i in range(10):  
    print('Welcome to everyone!!')
```



NOTE : 루프 제어변수의 익명화

위 반복문에서 새롭게 할당되는 변수 `i`는 실행문에서 사용되지 않는 루프변수이므로 다음과 같이 언더스코어(`_`)를 대신 넣어서 익명화 시킬 수 있다.

```
for _ in range(10):  
    print('Welcome to everyone!!!!')
```

코드 3-20 : for 문을 이용한 반복적 수행 - 10회 수행

print_welcome_with_for2.py

```
for i in range(10):  
    print(i, 'Welcome to everyone!!')
```

실행결과

```
0 Welcome to everyone!!  
1 Welcome to everyone!!  
2 Welcome to everyone!!  
3 Welcome to everyone!!  
4 Welcome to everyone!!  
5 Welcome to everyone!!  
6 Welcome to everyone!!  
7 Welcome to everyone!!  
8 Welcome to everyone!!  
9 Welcome to everyone!!
```

- 반복문에서 사용되는 변수는 `i, j, k, l, ...` 과 같은 알파벳 문자를 할당
- 이러한 변수를 C나 Java에서는 루프 `loop` 제어변수 라고 함



LAB 3-6 : 반복문을 이용해서 다음 코드를 작성해 보자

1. for 반복문에서 언더스코어 루프 제어변수를 사용하여 다음과 같이 Hello, Python!을 5번 출력하는 프로그램을 작성해 보자.

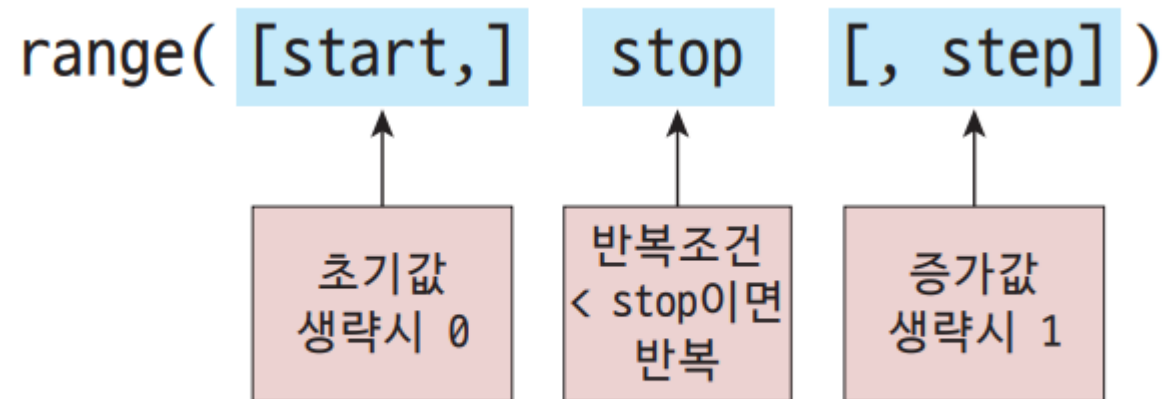
```
Hello, Python!  
Hello, Python!  
Hello, Python!  
Hello, Python!  
Hello, Python!
```

2. i라는 루프변수를 사용해서 0에서 4까지의 정수를 출력하는 프로그램을 작성하시오.

```
0  
1  
2  
3  
4
```

range() 함수의 사용법

- range(0, 5)와 같이 주어진 시작 값에서 마지막 값 사이의 연속적인 정수들을 생성할 수도 있으며, range(0, 5, 2)와 같이 마지막에 증가치 값을 넣어 줄 수도 있다.
- range(0, 5, 1)과 같이 호출할 경우 마지막의 1은 디폴트 간격(step) 값으로 1씩 더하면서 값을 변경하라는 의미



[그림 3-19] range() 함수의 사용법 : [start]와 [step]값은 생략할 수 있다.



NOTE : range() 함수의 인자

range() 함수는 실수 값을 인자로 가질 수 없다. 반면 나중에 배우게 될 Numpy 모듈의 arange()라는 함수는 실수의 시작 값, 종료 값, 스텝 값을 가질 수 있다.

3.5.1 반복문의 활용

- 1에서 10까지의 정수의 합 구하기

코드 3-25 : 연속적인 값의 생성과 누적 덧셈

```
for_sum_ex1.py
```

```
s = 0
```

```
for i in range(1, 11):
```

```
    s = s + i
```

```
print('1에서 10까지의 합:', s)
```

실행결과

1에서 10까지의 합: 55



LAB 3-8 : 누적 덧셈의 응용

1. 앞서 배운 누적 덧셈을 응용하여 1에서 100까지 정수의 합을 구하여 출력하여라(힌트 : range()의 구간이 1에서 100까지가 되도록 하자).
2. range() 함수의 step 값을 이용하여 1에서 100까지 정수 중에서 짝수의 합을 구하여 출력하여라(힌트 : range() 함수의 시작 값은 0으로 하고 step 값을 2로 하여라).
3. range() 함수의 step 값을 이용하여 1에서 100까지 정수 중에서 홀수의 합을 구하여 출력하여라(힌트 : range() 함수의 시작 값은 1로 하고 step 값을 2로 하여라).

팩토리얼factorial 구하기

- n 이 임의의 자연수 일때 1에서 n 까지의 모든 자연수를 곱한 값이다.

$$n! = \prod_{k=1}^n k = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

코드 3-29 : for 반복문을 이용한 5 팩토리얼(5!) 계산

for_factorial.py

```
n = int(input('수를 입력하세요 : '))  
fact = 1  
for i in range(1, n+1):  
    fact = fact * i  
  
print('{}! = {}'.format(n, fact))
```

- 변수 fact의 초기 값은 1로 두고 덧셈(+) 연산 대신 곱셈(*) 연산을 for문 안에서 사용

실행결과

수를 입력하세요 : 5

5! = 120



NOTE : 파이썬과 정수 표현의 한계

팩토리얼을 구하는 [코드 3-29]를 수정하여 n 값을 50으로 바꾼다면 다음과 같은 엄청나게 큰 수를 출력한다.

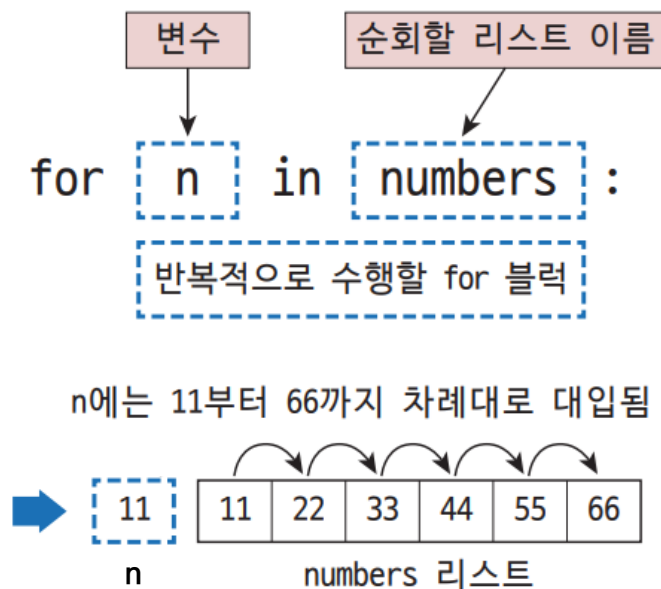
$50! = 30414093201713378043612608166064768844377641568960512000000000000.$

C나 Java와 같은 프로그래밍 언어는 정수의 크기가 4 바이트 형으로 정해져 있어서 일정한 크기 이상의 정수는 표현할 수 없다. 예를 들어 Java의 long형 정수의 범위는 $-9223372036854775808 \sim 9223372036854775807$ 까지이다. 이와 달리 파이썬은 정수 표현의 한계가 없다. 이 점이 파이썬의 또 다른 큰 장점이다.

for 문과 리스트

- for in 구문

- 반복문 키워드 for 와 in 사이에 계속 새롭게 할당할 변수 n을 선언
- in 뒤에 리스트 자료형을 넣어 리스트를 차례대로 순회하는 실행이 가능



[그림 3-22] for - in 구문에서 적용되는 numbers 리스트의 순회 방문 원리

코드 3-30 : for 문을 이용한 리스트의 정수 객체 순회

for_in_numbers.py

```
numbers = [11, 22, 33, 44, 55, 66]
```

```
for n in numbers:  
    print(n, end = ' ')
```

실행결과

11 22 33 44 55 66

코드 3-31 : for 문을 이용한 리스트의 실수 객체 순회

for_in_f_numbers.py

```
f_numbers = [1.1, 2.5, 3.7, 5.6, 9.2, 11.3, 6.8]
```

```
for f in f_numbers:  
    print(f, end = ' ')
```

실행결과

1.1 2.5 3.7 5.6 9.2 11.3 6.8

- for in 구문의 in 다음에 범위를 지정하는 함수 range()가 아닌 numbers 라는 리스트가 있음
- 실수 리스트도 for in문을 통해 순회가 가능

- for문을 이용하여 문자열을 원소로 가지는 리스트의 원소들을 출력

코드 3-32 : for 문을 이용한 리스트의 문자열 객체 순회

for_in_str_list.py

```
summer_fruits = ['수박', '참외', '체리', '포도']
```

```
for fruit in summer_fruits:
```

```
    print(fruit, end = ' ')
```

실행결과

수박 참외 체리 포도

- 누적 덧셈의 기능을 활용하여 리스트 내에 있는 정수 항목 값들의 합을 구하는 프로그램

코드 3-33 : 리스트 항목내 정수 값들의 누적 덧셈

for_sum1.py

```
numbers = [10, 20, 30, 40, 50]
```

```
s = 0
```

```
for n in numbers:
```

```
    s = s + n
```

```
print('리스트 항목 값의 합 :', s)
```

실행결과

리스트 항목 값의 합 : 150

- 리스트 원소들의 합은 for 문을 사용하지 않고 내장함수 `sum()`을 사용하여 간편하게 합계를 구하는 것도 가능

코드 3-34 : `sum()` 함수의 사용

`for_sum2.py`

```
numbers = [10, 20, 30, 40, 50]
```

```
print('리스트 항목 값의 합 :', sum(numbers))
```

실행결과

리스트 항목 값의 합 : 150

대화창 실습 : 대화형 모드를 통한 1에서 100까지의 합

```
>>> print('1에서 100까지의 합 :', sum(range(1, 101)))
```

```
1에서 100까지의 합 : 5050
```

- 문자열 자료형은 `list()` 함수를 이용하여 리스트 객체로 만드는 것이 가능

대화창 실습 : 대화형 모드를 통한 str 자료형의 리스트화

```
>>> st = 'Hello'
```

```
>>> list(st)
```

```
['H', 'e', 'l', 'l', 'o']
```

중첩 for 루프

- 이중 for문 `nested for loop`
 - for문 안에 for문을 다시 넣음
- 구구단의 구조에 대해 살펴보면, 2~9단까지 있으며 1에서 9를 단마다 곱하여 화면에 출력함
- 이를 구현하기 위해서는 for 문 안에 for 문을 다시 넣는 이중 for문이 필요

코드 3-36 : 중첩 for 문을 사용한 구구단 출력

double_for.py

```
for i in range(2, 10):          # 외부 for 루프
    for j in range(1, 10):      # 내부 for 루프
        print('{}*{} = {:2d}'.format(i, j, i*j), end = ' ')
    print()                    # 내부 루프 수행 후 줄바꿈을 함
```

실행결과

```
2*1= 2 2*2= 4 2*3= 6 2*4= 8 2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
3*1= 3 3*2= 6 3*3= 9 3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
4*1= 4 4*2= 8 4*3=12 4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
5*1= 5 5*2=10 5*3=15 5*4=20 5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
6*1= 6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36 6*7=42 6*8=48 6*9=54
7*1= 7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49 7*8=56 7*9=63
8*1= 8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64 8*9=72
9*1= 9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
```

- double_for.py의 이중 for 루프는 내부 루프와 외부 루프를 가짐

```
외부
[ 내부
  for i in range(2,10):
    for j in range(1,10):
      print('{} * {} = {}'.format(i, j, i*j))
```

• 루프의 실행구조를 표로 나타내어 보기

i = 2 일 때	i = 3일 때	i = 4일 때	...	i = 9일 때
j = 1 : 2 * 1	j = 1 : 3 * 1	j = 1 : 4 * 1	...	j = 1 : 9 * 1
j = 2 : 2 * 2	j = 2 : 3 * 2	j = 2 : 4 * 2	...	j = 2 : 9 * 2
j = 3 : 2 * 3	j = 3 : 3 * 3	j = 3 : 4 * 3	...	j = 3 : 9 * 3
j = 4 : 2 * 4	j = 4 : 3 * 4	j = 4 : 4 * 4	...	j = 4 : 9 * 4
j = 5 : 2 * 5	j = 5 : 3 * 5	j = 5 : 4 * 5	...	j = 5 : 9 * 5
j = 6 : 2 * 6	j = 6 : 3 * 6	j = 6 : 4 * 6	...	j = 6 : 9 * 6
j = 7 : 2 * 7	j = 7 : 3 * 7	j = 7 : 4 * 7	...	j = 7 : 9 * 7
j = 8 : 2 * 8	j = 8 : 3 * 8	j = 8 : 4 * 8	...	j = 8 : 9 * 8
j = 9 : 2 * 9	j = 9 : 3 * 9	j = 9 : 4 * 9	...	j = 9 : 9 * 9

소수 검사하기

- 이중 for문을 활용해 소수인지 확인하는 프로그램을 구현
- 소수란 1과 자기 자신 이외의 약수를 가지지 않는 수
- 다음과 같은 코드로 구현할 수 있다

```
n = int(input('수를 입력하세요 :'))  
is_prime = True  
for num in range(2, n):      # 2부터 (n-1) 사이의 수 num에 대하여  
    if n % num == 0:         # 이 수 중에서 n의 약수가 있으면  
        is_prime = False    # 소수가 아님  
print(n, 'is prime :', is_prime)
```


2부터 100까지의 소수 구하기

코드 3-43 : 2부터 100까지의 소수 구하기

get_primes.py

소수를 담을 리스트 초기화

primes = []

for n in range(2, 101):

일단 n을 소수라고 두자

is_prime = True

for num in range(2, n): # 2~(n-1) 사이의 수 num에 대하여

if n % num == 0: # 이 수 중에서 n의 약수가 있으면

is_prime = False # 소수가 아님

if is_prime: # 소수일 경우 primes라는 리스트에 추가한다

primes.append(n) # append() 메소드는 리스트에 n을 추가함

print(primes)

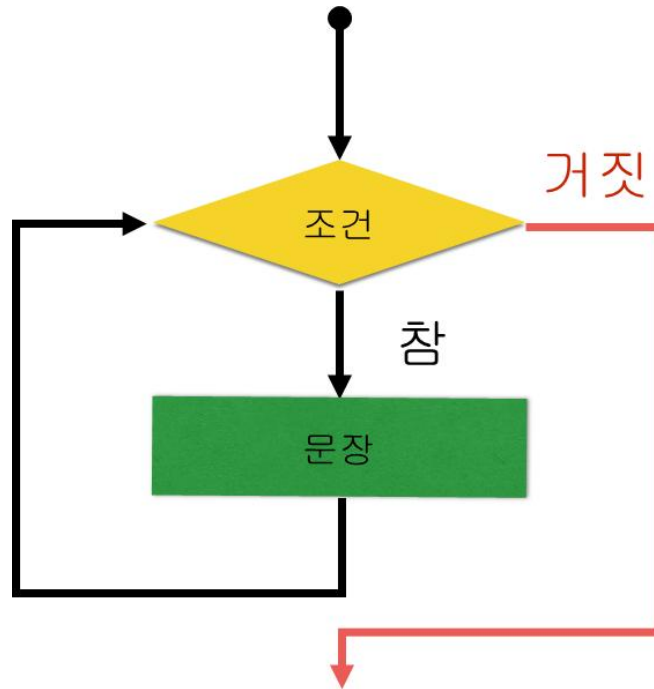
실행결과

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
89, 97]

while 반복문

- while 반복문의 구조

- 조건이 참인 경우에 계속 실행하는 반복문



while문의 문법

- if문과 매우 유사
- 조건식이 참이라면 계속 반복하여 해당 코드를 실행

형 식	예 시
초기값 지정 while 조건식 : 실행할 코드 블록	i = 0 # 초기 값 지정 while i<5 : # 조건식 print('Welcome to everyone!!') i += 1

코드 3-44 : for 문을 이용한 'Welcome to everyone!!'의 반복 출력기능

print_welcome_with_for.py

```
for i in range(5):  
    print('Welcome to everyone!!')
```

코드 3-45 : while 문을 이용한 'Welcome to everyone!!'의 반복 출력기능

print_welcome_with_while.py

```
i = 0 # 초기 값  
while i < 5: # 루프의 조건식이 참이면 내부 블록이 실행됨  
    print('Welcome to everyone!!')  
    i += 1 # 조건 값의 변경
```

실행결과

```
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!  
Welcome to everyone!!
```

- i를 0부터 1씩 증가시키며 5보다 작을 때까지 while 내부의 코드를 반복함
- 다음과 같이 새로 작성 가능

코드 3-46 : 지정된 수까지의 누적 합을 구하는 기능

while_sum_input.py

```
n = int(input('합계를 구할 수를 입력하세요 : '))
s = 0
i = 1
while i <= n:
    s = s + i
    i += 1
print('1부터 {}까지의 합은 {}'.format(n, s))
```

실행결과

합계를 구할 수를 입력하세요 : 100

1부터 100까지의 합은 5050

- 반복실행의 횟수가 명확한 경우는 while문의 코드가 길어지기 때문에 for문을 사용하는 것이 더 나은 방법

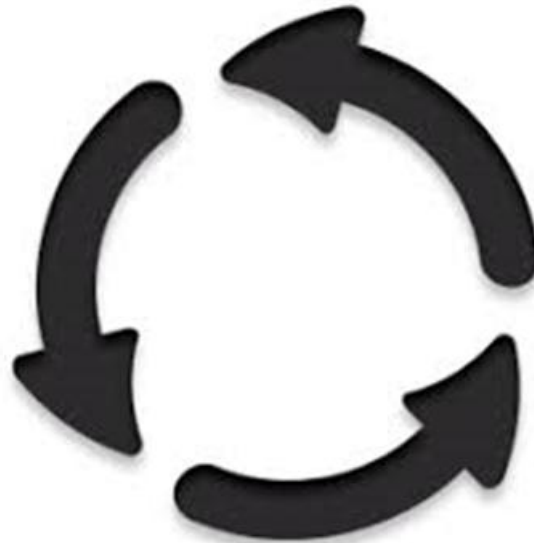
while문과 for문 비교

- while문은 수행횟수를 정확히 모르지만 수행의 조건이 명확한 경우에 더 적합
- 반복 횟수가 명확한 경우 for문이 적합

1에서 n까지의 합을 구하는 코드 비교	
while 문	for 문
<pre>s = 0 i = 1 while i <= n: s = s + i i += 1</pre>	<pre>s = 0 for i in range(1, n+1) : s = s + i</pre>

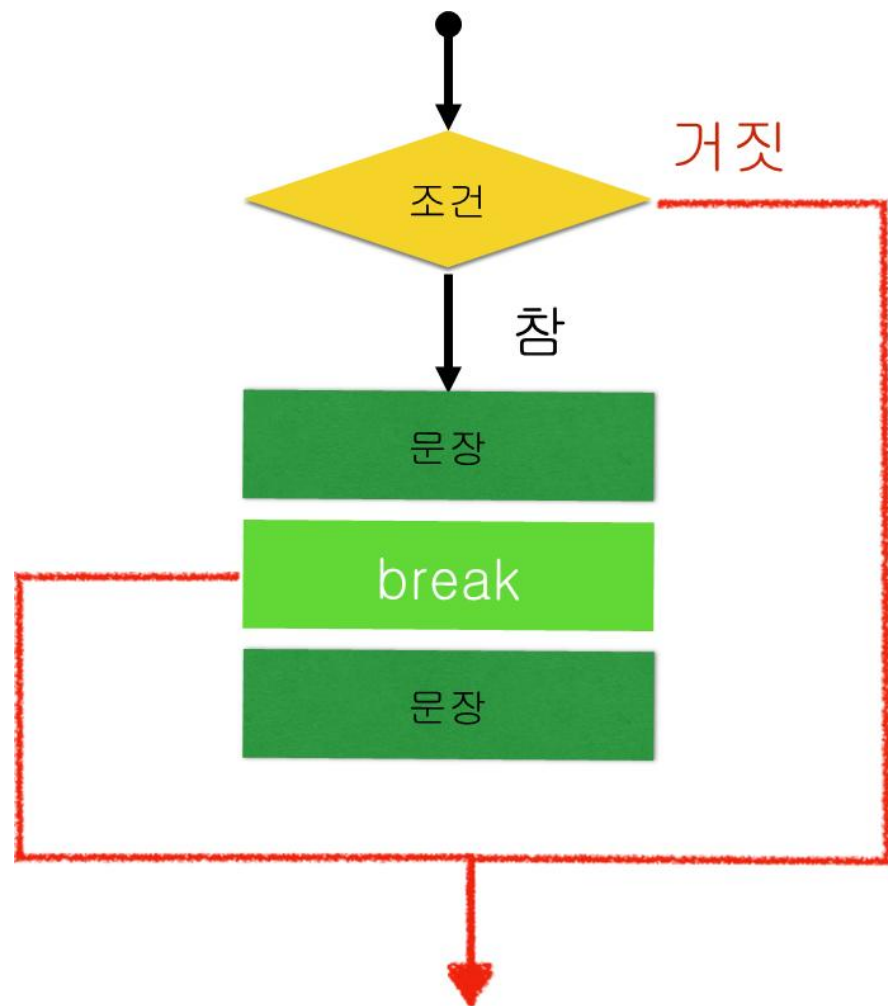
break와 continue

- 반복문을 제어하는 키워드
 - 반복 실행을 종료 -> break
 - 반복문 루프 내의 나머지 실행부를 건너뛰고 계속해서 반복 루프를 실행 -> continue
 - continue는 반복 실행을 종료하지 않음



break를 표현한 흐름도

- while이나 반복문은 조건이 참이면 블록 내의 문장을 수행
- 도중에서 break를 만나면 그 즉시 반복 실행을 종료하고 루프를 빠져나옴



코드 3-50 : break를 사용하여 모음이 나타나면 즉시 반복문을 종료하는 기능

```
skip_vowel_break.py
```

```
st = 'Programming'
```

```
# 자음이 나타나는 동안만 출력하는 기능
```

```
for ch in st:
```

```
    if ch in ['a','e','i','o','u']:
```

```
        break # 모음일 경우 반복문을 종료한다.
```

```
    print(ch)
```

```
print('The end')
```

실행결과

P

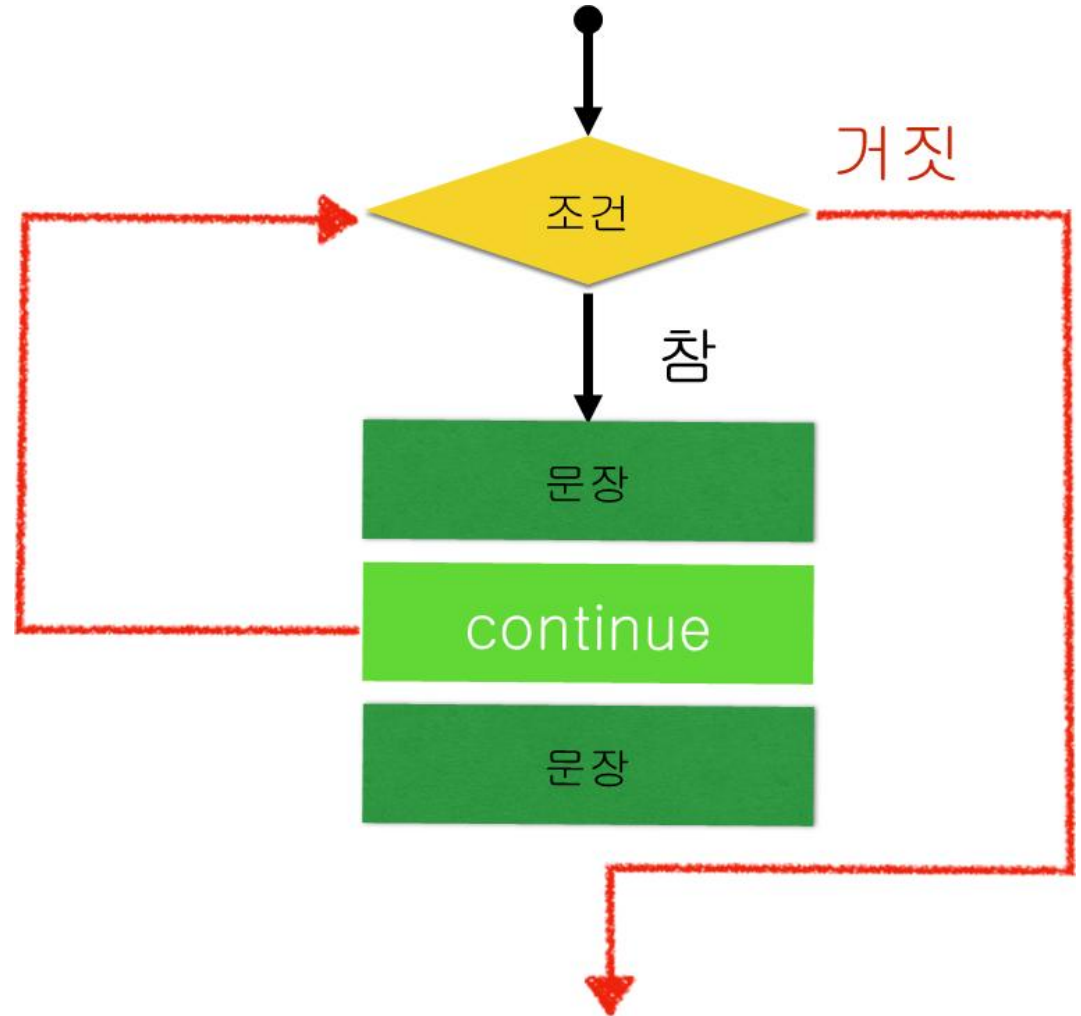
r

The end

- 모음에 해당하면 break
- 그렇지 않으면 print를 이용하여 출력
- break를 작동시키면 반복문의 나머지 부분을 실행하지 않고 루프를 중지시킴

continue를 표현한 흐름도

- 루프를 빠져나오지 않고 continue 아래의 문장만을 건너 뛰는 역할
- 반복문이 종료되는 것은 조건이 거짓일 때에만 해당



코드 3-51 : continue를 사용하여 모음일 경우 출력을 건너뛰는 기능

skip_vowel_continue.py

```
st = 'Programming'
```

```
# 자음이 나타날때만 출력하는 기능
```

```
for ch in st:
```

```
    if ch in ['a','e','i','o','u']:
```

```
        continue # 모음일 경우 아래 출력을 건너뛴다
```

```
    print(ch)
```

```
print('The end')
```

실행결과

P

r

g

r

m

m

n

g

The end

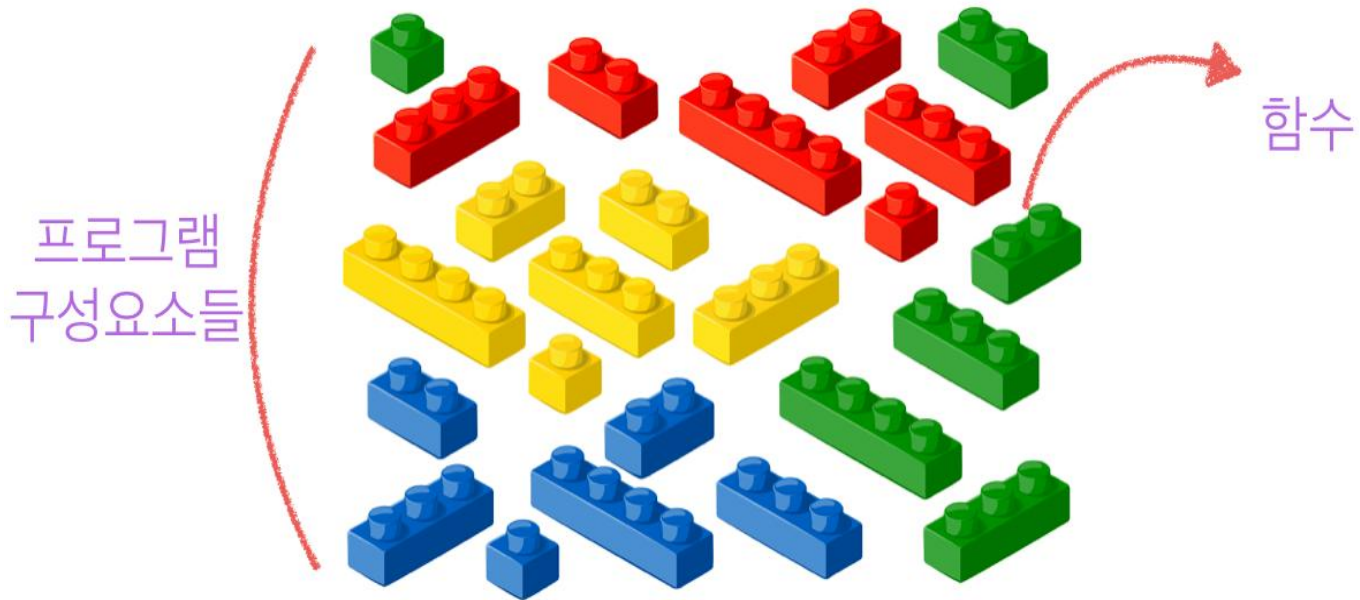
- continue를 넣게 되면 아래에 있는 아래의 나머지 부분을, 즉 print를 실행하지 않고 반복문의 처음으로 돌아가는 기능을 함



주의 : break와 continue의 흐름제어의 위험성

break와 continue는 프로그램의 제어를 효율적으로 하는데 편리하게 사용할 수 있다. 하지만, break와 continue 문이 너무 많이 사용되는 경우 제어의 흐름에 일관성이 없어 프로그램을 이해하는 것이 어려워진다. 따라서 continue와 break는 필요한 경우에만 제한적으로 사용하는 것이 좋다.

함수의 역할



[그림 4-1] 레고 블록

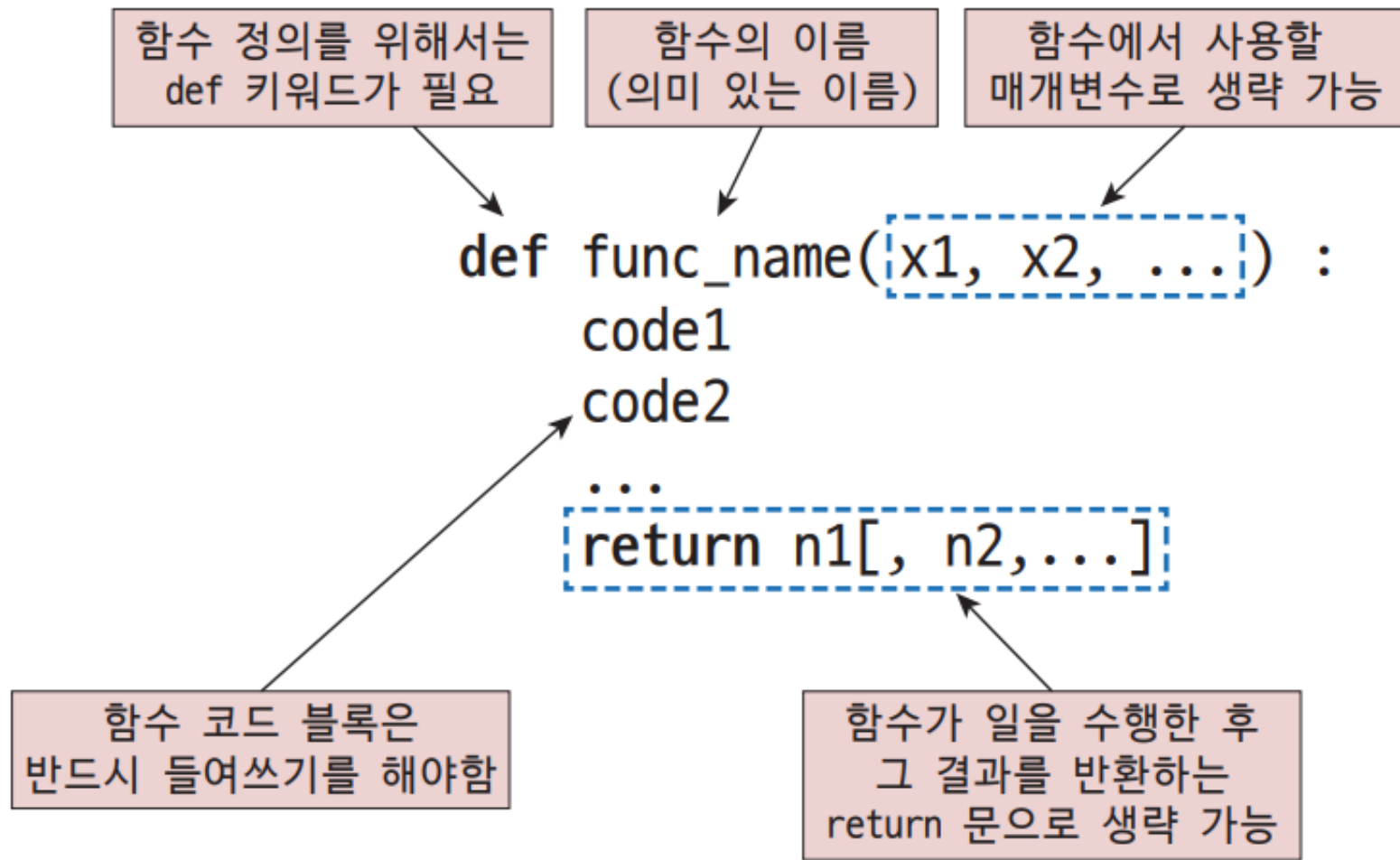


[그림 4-2] 레고 블록을 조립해서 만든 자동차

(출처: bricklink.com)

여러분이 사용하는 프로그램도 많은 부품(함수나 클래스)으로 이루어져 있습니다

- 반복적으로 사용되는 코드 - 덩어리(혹은 블록`block`) 이라고 함
- 기능에 따라 미리 만들어진 블록은 필요할 때 호출`function call`함
- 파이썬에서 미리 만들어서 제공하는 함수는 인터프리터에 포함되어 배포되는데 이러한 함수를 내장함수`built-in function` 라고 함
 - 대표적으로 `print()`가 있음
- 사용자가 직접 필요한 함수를 만들 수 있음
- 이러한 함수를 사용자 정의 함수`user defined function`라고 함
- `def` 키워드 사용 : `define`의 약자
 - `def`를 이용한 함수 정의 방법을 배워볼 예정



[그림 4-3] 파이썬에서 함수를 정의하는 문법

- **return문이 없는 간단한 코드로 함수를 정의하고 호출하기**

코드 4-1 : 별표 출력을 위한 함수 정의와 호출

print_star_func.py

```
def print_star():                # 별표 출력을 위한 함수 정의
    print('*****')

print_star()                     # 별표 출력을 위한 함수 호출
```

실행결과

```
*****
```


코드 4-2 : 별표 출력을 위한 함수 정의와 반복 호출

print_star_4.py

```
def print_star():  
    print('*****')  
  
print_star()  # 별표 출력함수 호출 1  
print_star()  # 별표 출력함수 호출 2  
print_star()  # 별표 출력함수 호출 3  
print_star()  # 별표 출력함수 호출 4
```

실행결과

```
*****  
  
*****  
  
*****  
  
*****
```

- **print_star()**라는 함수는 어떤 일을 하도록 정의된 명령어들의 집합(혹은 블록)이며 이 집합은 외부에서 호출할 때마다 수행되는 것을 확인해 볼 수 있다.



LAB 4-1 : 함수 정의와 호출

1. [코드 4-1]의 함수 호출문을 삭제하면 어떻게 되는가?
2. [코드 4-2]를 수정하여 6줄의 별표를 출력해 보시오. 이때 함수 호출을 6회 하시오.

```
*****  
  
*****  
  
*****  
  
*****  
  
*****  
  
*****
```

코드 4-3 : 3줄 별표 출력을 위한 함수 정의와 호출 방법

print_star3.py

```
def print_star3():
```

```
    print('*****')
```

```
    print('*****')
```

```
    print('*****')
```

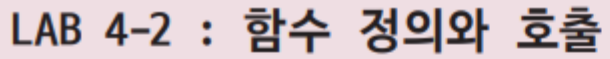
```
print_star3() # 3줄의 별표가 출력됨
```

```
print_star3() # 3줄의 별표가 출력됨
```

```
print_star3() # 3줄의 별표가 출력됨
```

실행결과

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

[illegible]

코드 4-4 : 별표 출력을 위한 함수 정의와 호출 방법의 수정

print_star_plus.py

```
def print_star(): # 별표 기호를 한 줄 출력함
    print('*****')

def print_plus(): # 더하기 기호를 한 줄 출력함
    print('+++++')

print_star()      # 별표 기호 출력
print_plus()      # 더하기 기호 출력
print_star()
print_plus()
```

실행결과

```
*****
+++++
*****
+++++
```

- 한번 만들어진 함수는 다른 프로그램에서 재사용이 가능
- 프로그램 개발의 시간과 비용을 절약할 수 있다



LAB 4-3 : 함수 정의와 호출

1. [코드 4-4]를 수정하여 해시마크(#)를 한 줄 출력하는 `print_hash()` 함수를 추가로 구현하십시오.
2. `print_star()`, `print_plus()`, `print_hash()` 함수를 모두 이용하여 다음과 같은 출력이 나타나도록 함수를 호출하십시오.

```
#####  
*****  
+++++++  
+++++++  
*****  
#####
```

함수와 매개변수

전달받을 값 3, 4를 가지는 변수 m, n : 매개변수

```
def foo(m, n) :  
    code  
    ...  
    return n1[, n2,...]
```

foo(3, 4)

foo 라는 함수에 넘겨줄 값 3, 4 : 인자

[그림 4-4] 매개변수와 인자의 개념과 사용방법



NOTE : 인자와 매개변수

- **매개변수**Parameter : 함수나 메소드 헤더부에 정의된 변수로 함수가 호출될 때 실제 값을 전달받는 변수이다.

예: def foo(m, n): 의 m과 n

- **전달인자**Argument : 함수나 메소드가 호출될 때 전달되는 실제 값을 말하며, 간단하게 인자라고도 한다.

예: foo(3, 4)의 3과 4

코드 4-5 : 매개변수를 가진 별표 출력 함수와 인자를 이용한 호출

print_star_param.py

별표 출력을 매개변수 n번만큼 반복하는 프로그램

```
def print_star(n):
```

```
    for _ in range(n):
```

```
        print('*****')
```

```
print_star(4) # 별표 출력을 위해 4라는 인자 값을 준다.
```

실행결과

```
*****
```

```
*****
```

```
*****
```

```
*****
```


매개변수를 활용한 2차 방정식의 근 구하기

$$ax^2 + bx + c = 0$$

[수식 4-1] x에 대한 2차 방정식

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

[수식 4-2] 2차 방정식의 근의 공식

코드 4-8 : 2차 방정식의 근을 구하는 기능

root_ex1.py

```
a = 1
```

```
b = 2
```

```
c = -8
```

```
# ( a * x^2 ) + ( b * x ) + c = 0
```

```
r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
```

```
r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
```

```
print('해는', r1, '또는', r2)
```

실행결과

해는 2.0 또는 -4.0

- $a(a \neq 0)$, b , c 를 해당하는 값을 방정식에 맞게 입력
- a , b , c 에 해당하는 해를 변수 $r1$, $r2$ 에 저장하여 출력

- 변수 a, b, c의 값을 2, -6, -8로 바꾼 방정식의 해를 구하고 싶을 때

코드 4-9 : 2차 방정식의 근을 구하는 기능의 반복 사용

root_ex2.py

```
a = 1
b = 2
c = -8
# 근의 공식으로 해를 한 번 더 구한다.(반복되는 코드)
r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
print('해는', r1, '또는', r2)

a = 2
b = -6
c = -8
# 근의 공식으로 해를 한 번 더 구한다.(반복되는 코드)
r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
print('해는', r1, '또는', r2)
```

실행결과

해는 2.0 또는 -4.0

해는 4.0 또는 -1.0

- 문제점

- 변수 a , b , c 에 원하는 계수를 입력하고, 다시 $r1$, $r2$ 의 수식을 구해줘야 함
- 복사, 붙여 넣기를 한다 해도 코드가 중복되는 부분이 많고 불필요하게 긴 것을 한눈에 알 수 있다.

코드 4-10 : 2차 방정식의 근을 구하는 기능을 함수로 만들기

root_ex3.py

```
def print_root(a, b, c):  
    r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    print('해는', r1, '또는', r2)
```

계수 값이 다른 2차 방정식의 해를 구함

```
print_root(1, 2, -8)
```

```
print_root(2, -6, -8)
```

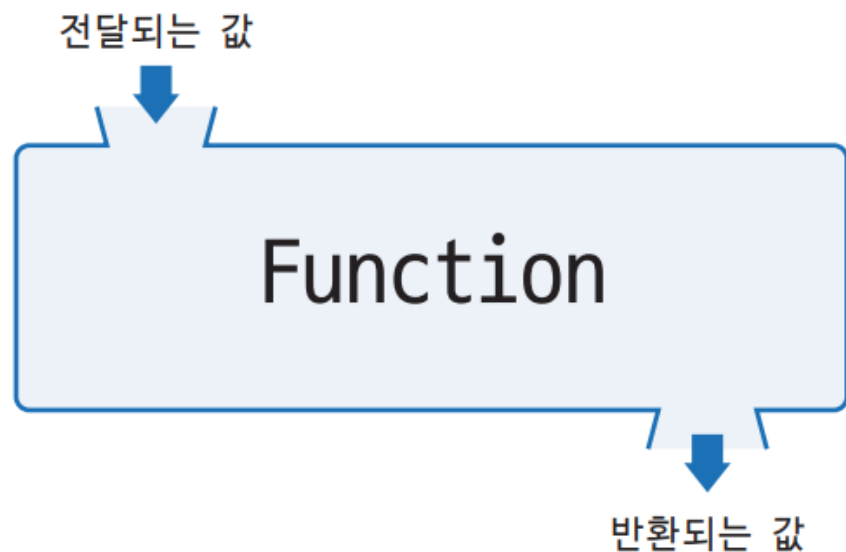
- 밖에서 넘겨준 계수 3개 a, b, c를 매개변수로 받고, 함수 몸체에 근의 공식 연산을 한 후, 결과 r1, r2를 출력하는 코드
- 코드가 훨씬 간결해지고, 사용하기 편리함

실행결과

해는 2.0 또는 -4.0

해는 4.0 또는 -1.0

반환문 return



[그림 4-5] 값의 전달과 반환 : 함수는 값을 전달받아 처리하고 결과를 반환할 수 있다

```
def func_name([x1, x2, ...]) :  
    code1  
    code2  
    ...  
    return n1[, n2,...]
```

함수가 일을 수행한 후 그 결과를 반환하는 기능

[그림 4-6] 함수의 구성과 반환문 return

반환문 return

- 일반적으로 함수 내부는 블랙박스black box라고 가정
- 함수의 내부는 특정한 코드를 가지고 있으며 주어진 일을 수행하고 결과를 반환할 수 있음
- return 키워드를 사용하여 하나 이상의 값을 반환해 줄 수 있음

코드 4-11 : 두 값의 합을 반환하는 get_sum() 함수와 return 문의 사용

sum_with_return1.py

```
def get_sum(a, b):                # 두 수의 합을 반환하는 함수
    result = a + b
    return result                 # return 문을 사용하여 result를 반환

n1 = get_sum(10, 20)
print('10과 20의 합 =', n1)

n2 = get_sum(100, 200)
print('100과 200의 합 =', n2)
```

실행결과

10과 20의 합 = 30

100과 200의 합 = 300

전역 변수

- 전역변수 **global variable**
 - 함수 바깥에서 선언되거나 전체 영역에서 사용 가능한 변수

코드 4-14 : 매개변수를 사용하지 않고 외부 변수를 사용하는 경우

sum_func_global1.py

```
def print_sum():  
    result = a + b  
    print('print_sum() 내부 :', a, '과', b, '의 합은', result, '입니다.')
```



```
a = 10    # 전역변수 a  
b = 20    # 전역변수 b  
  
print_sum()  
  
result = a + b  
  
print('print_sum() 외부 :', a, '과', b, '의 합은', result, '입니다.')
```

실행결과

print_sum() 내부 : 10 과 20 의 합은 30 입니다.

print_sum() 외부 : 10 과 20 의 합은 30 입니다.

코드 4-15 : 함수 외부에서 정의된 값을 함수 내부에서 변경하는 경우

sum_func_global2.py

```
def print_sum():  
    a = 100  
    b = 200  
    result = a + b  
    print('print_sum() 내부 :', a, '과', b, '의 합은', result, '입니다.')
```

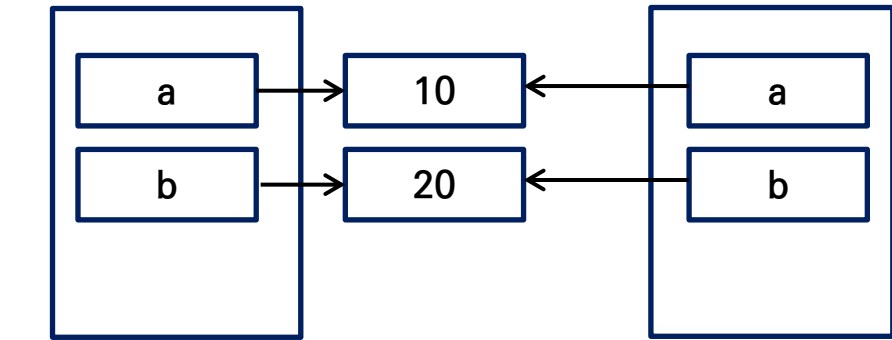
a = 10

b = 20

print_sum()

실행결과

print_sum() 내부 : 100 과 200 의 합은 300 입니다.



파이썬
스크립트 파일

print_sum() 함수
전역변수의 사용

[그림 4-8] 파이썬 스크립트 파일과 전역변수,
그리고 이 전역변수를 사용하는
print_sum() 함수

코드 4-16 : 함수 내부에서 값을 변경하고, 그 값을 외부에서 확인하기

sum_func_global3.py

```
def print_sum():  
    a = 100  
    b = 200  
    result = a + b  
    print('print_sum() 내부 : ', a, '과', b, '의 합은', result, '입니다.')
```

a = 10
b = 20
print_sum()
result = a + b
print('print_sum() 외부 : ', a, '과', b, '의 합은', result, '입니다.')

100, 200을 참조하는 새로운
a, b 변수 생성

10, 20을 참조하는 a, b 변수 생성

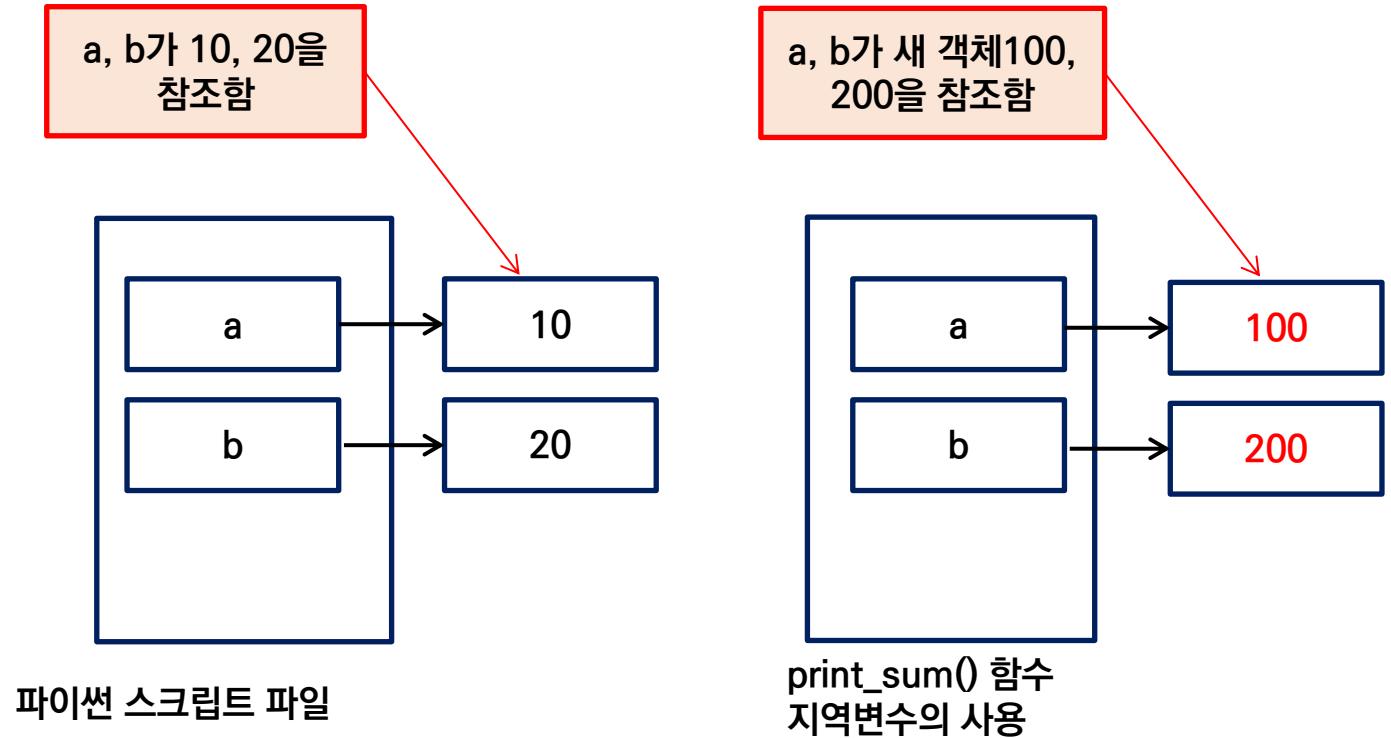
실행결과

print_sum() 내부 : 100 과 200 의 합은 300 입니다.

print_sum() 외부 : 10 과 20 의 합은 30 입니다.

- print_sum()을 수행한 다음 함수 외부에서 다시 한번 a와 b를 합하여 result에 대입하고 그 결과를 출력

- 할당 **assign**
 - `a = 100, b = 200`
- 지역 변수 **local variable**
- 참조 **reference**



[그림 4-9] 파이썬 스크립트 파일과 전역변수, 그리고 지역변수를 사용하는 `print_sum()` 함수. 이 함수 내부의 `a`, `b`는 지역변수가 참조하는 객체가 아닌 별개의 객체를 참조함

코드 4-17 : global 키워드를 사용한 전역변수의 참조 방법

sum_func_global4.py

```
def print_sum():
```

```
    global a, b
```

a, b는 함수외부에서 선언된 a, b를 사용한다.

```
    a = 100
```

```
    b = 200
```

```
    result = a + b
```

```
    print('print_sum() 내부 :', a, '과', b, '의 합은', result, '입니다.')
```

```
a = 10
```

```
b = 20
```

```
print_sum()
```

```
result = a + b
```

```
print('print_sum() 외부 :', a, '과', b, '의 합은', result, '입니다.')
```

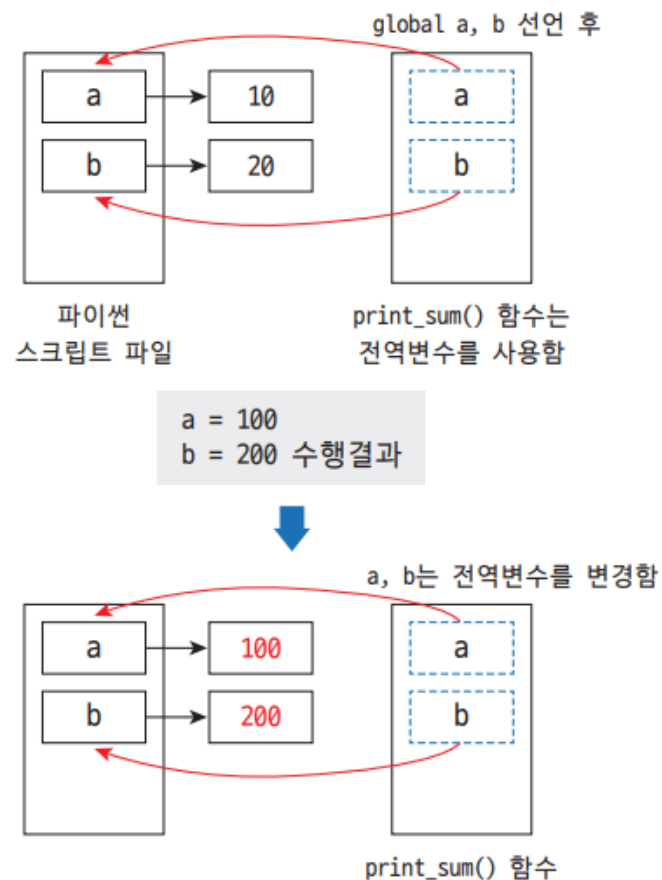
전역변수 a, b가 100, 200을 참조함

실행결과

print_sum() 내부 : 100 과 200 의 합은 300 입니다.

print_sum() 외부 : 100 과 200 의 합은 300 입니다.

global a = 100 # 문법 오류 발생



[그림 4-9] 명시적 global 선언을 통하여 함수 내에서 전역변수 a, b를 사용하는 과정



주의 : 전역변수와 전역상수

전역변수를 사용하는 것은 파이썬뿐만 아니라 모든 프로그래밍 언어에서 매우 나쁜 습관이다. 특히 코드의 길이가 길어질 경우 전역변수는 예외의 주요 원인이 된다.

그러나 **전역상수** `global constant`의 경우는 반드시 나쁘다고 볼 수 없다. 전역상수는 다음과 같이 `global`이라는 키워드로 선언하는데 함수의 외부에서 선언해서 모듈 전체에서 참조할 수 있다. 전역 상수값은 일반적으로 **대문자**를 사용한다.

아래의 코드를 살펴보면, `GLOBAL_VALUE`라는 이름의 변수에 1024라는 값을 할당한 후 `foo()` 함수에서 이 변수 값을 불러서 사용하기 위해 `global GLOBAL_VALUE`라는 이름으로 선언했다.

전역상수의 예 :

```
GLOBAL_VALUE = 1024
...
def foo():
    global GLOBAL_VALUE
    a = GLOBAL_VALUE * 100
```

수학 연산을 위해 사용되는 `math` 모듈의 경우 원주율 `pi`와 오일러 상수 `e`를 프로그램 전체에서 참조하여 사용하는데, 이러한 상수 값의 경우는 예외적으로 소문자로 표기한다.

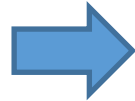
4.6 함수의 인자 전달 방식

코드 4-18 : 인자를 빠뜨린 호출

print_star_param_error.py

```
def print_star(n): # 인자를 필요로 함
    for _ in range(n):
        print('*****')
```

print_star() # 인자가 없으므로 에러 발생



코드 4-19 : 디폴트 값을 가지는 print_star() 함수

print_default_param.py

```
def print_star(n = 1): # 매개변수 n은 디폴트 값 1을 가짐
    for _ in range(n):
        print('*****')
```

print_star() # 인자가 없더라도 에러 없이 수행됨

실행결과

TypeError: print_star() missing 1 required positional argument: 'n'

- 함수에 특정한 작업을 위임하기 위하여 정확한 인자를 넣어주는 것도 필요하지만 가끔씩은 위와 같은 에러를 예방하고, 좀 더 유연성 있는 작업을 위해서 디폴트 값을 사용하는 것이 편리할 때가 있음
- 이때 사용하는 것이 디폴트 매개변수 `default parameter`
- [코드 4-19]와 같이 매개변수에 `= 1`과 같이 디폴트 값을 할당

- 인자가 없이 호출해도 디폴트 값 1을 매개변수 n에 전달하므로 한 줄의 별표 라인이 정상적으로 출력됨

코드 4-19 : 디폴트 값을 가지는 print_star() 함수

print_default_param.py

```
def print_star(n = 1):  # 매개변수 n은 디폴트 값 1을 가짐
    for _ in range(n):
        print('*****')
```

```
print_star() # 인자가 없더라도 에러 없이 수행됨
```

실행결과

```
*****
```

키워드 인자 keyword argument

코드 4-22 : 2차 방정식의 근을 구하는 함수와 함수 호출문

root_func.py

```
def get_root(a, b, c):  
    r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)  
    return r1, r2
```

함수 호출시 인자를 1, 2, -8 인자를 사용함.

result1, result2를 이용해서 결과 값을 반환 받는다.

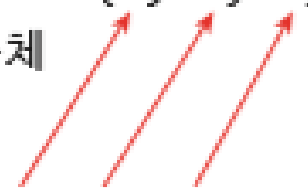
```
result1, result2 = get_root(1, 2, -8)  
print('해는', result1, '또는', result2)
```

실행결과

해는 2.0 또는 -4.0

- 함수를 호출할 때 인자의 값만을 전달하는 것이 아니라 그 인자의 이름을 함께 명시하여 전달하는 방식
- 파이썬의 기본 인자 전달 방식을 위치 인자 positional argument 방식이라고 함

```
def get_root(a, b, c):  
    함수 몸체  
  
get_root(1, 2, -8)  #함수 호출문
```



[그림 4-10] 위치 인자의 전달 방식 : 매개변수에 전달할 값을 전달할 때 a, b, c 순서에 따라 전달하므로 순서가 중요함

```
result1, result2 = get_root(-8, 2, 1) # 1, 2, -8을 인자로 줄 때와 그 결과가 다름
```

실행결과

해는 -0.25 또는 0.5

위치와 상관없이 키워드에 의해서 인자 값이 결정됨

```
result1, result2 = get_root(a = 1, b = 2, c = -8)
```

실행결과

해는 2.0 또는 -4.0

```
result1, result2 = get_root(a = 1, c = -8, b = 2)
```

위의 코드와 아래 코드는 그 결과가 동일하다,
키워드 인자를 사용하면 인자의 위치는 중요하지 않다

```
result1, result2 = get_root(c = -8, b = 2, a = 1)
```

실행결과

해는 2.0 또는 -4.0

```
def get_root(a, b, c):  
    함수 몸체  
  
get_root(a=1, c=-8, b=2)
```

[그림 4-11] 키워드 인자의 전달 방식 : a, b, c의 키워드를 통해서 매개변수에 전달할 값을 명시해 주므로 순서는 중요하지 않음

result1, result2 = get_root(c = -8, b = 2, 1)

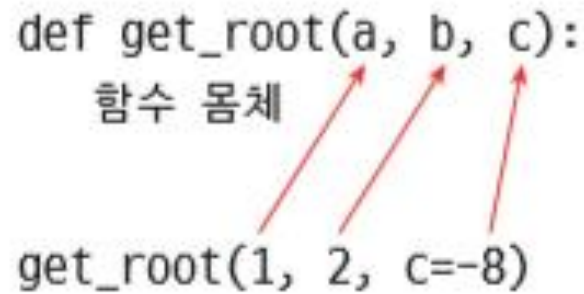
키워드 인자와 위치 인자를 섞어서 사용할 적에는 반드시 위치인자가 먼저 나타나야 한다(위의 경우는 오류)

SyntaxError: positional argument follows keyword argument

result1, result2 = get_root(1, 2, c = -8)

키워드 인자와 위치 인자를 섞어서 사용할 적에 위치인자를 먼저 적어주면 된다

```
def get_root(a, b, c):  
    함수 몸체  
  
get_root(1, 2, c=-8)
```



[그림 4-12] 위치 인자와 키워드 인자의 혼용 : 1, 2는 a, b에 전달되고 -8은 키워드를 통해 명시해준 c에 전달됨



NOTE : 위치 인자와 키워드 인자로 인자를 전달하기

파이썬의 함수에서는 인자를 전달할 때 위치 인자로 전달하는 방식과 키워드 인자로 전달하는 방식이 있다. 그리고 두 가지 방식을 혼합하는 방식도 있다. 그러나 두 가지 방식을 혼합하는 경우 **반드시** 위치 인자 뒤에 키워드 인자가 와야 한다.

```
result1, result2 = get_root(1, -8, b = 2)
```

TypeError: get_root() got multiple values for argument 'b'

```
>>> def func(a, b, c) :
```

```
...     print(a, b, c)
```

```
...
```

```
>>> func(1, 2, 3)
```

```
1 2 3
```

```
>>> func(1, c=2, b=3)
```

```
1 3 2
```

```
>>> func(1, b=2, 3)
```

SyntaxError: positional argument follows keyword argument



LAB 4-9 : 키워드 인자

1. 다음과 같이 성(last name)과 이름(first name), 존칭(honorifics)을 매개변수로 받아서 출력하는 함수 print_name이 있다.

```
def print_name(honorifics, first_name, last_name):  
    ''' 키워드 인자를 이용한 출력용 프로그램 '''  
    print(honorifics, first_name, last_name)
```

a) 다음과 같은 함수 호출의 결과는 무엇인가?

```
print_name(first_name='Gildong', last_name='Hong', honorifics='Dr.')
```

b) 다음과 같은 함수 호출의 결과는 무엇인가?

```
print_name('Gildong', 'Hong', 'Dr.')
```


코드 4-24 : 인자를 하나 가지는 함수

arg_greet1.py

```
def greet1(name):  
    print('안녕하세요', name, '씨')  
  
greet1('홍길동')
```

실행결과

안녕하세요 홍길동 씨

코드 4-25 : 인자를 2개 가지는 함수

arg_greet2.py

```
def greet2(name1, name2):  
    print('안녕하세요', name1, '씨')  
    print('안녕하세요', name2, '씨')  
  
greet2('홍길동', '홍길순')
```

실행결과

안녕하세요 홍길동 씨

안녕하세요 홍길순 씨

인자의 개수를 미리 알 수 없을 경우에는 어떻게 해야만 할까?

가변적인 인자전달

- 인자의 수가 정해지지 않은
가변 인자 `arbitrary argument`

→ 별표(*)를 매개변수의
앞에 넣어 사용

- 가변적 인자는 튜플이나
리스트와 비슷하게 `for - in`문에서 사용가능

코드 4-26 : 가변 인자를 가지는 함수의 정의와 호출

arg_greet.py

```
def greet(*names):
```

```
    for name in names:
```

```
        print('안녕하세요', name, '씨')
```

```
greet('홍길동', '양만춘', '이순신') # 인자가 3개
```

```
greet('James', 'Thomas') # 인자가 2개
```

실행결과

안녕하세요 홍길동 씨

안녕하세요 양만춘 씨

안녕하세요 이순신 씨

안녕하세요 James 씨

안녕하세요 Thomas 씨

코드 4-27 : 가변 인자를 가지는 함수에서 len() 함수 활용

arg_foo.py

```
def foo(*args):  
    print('인자의 개수:', len(args))  
    print('인자들 :', args)
```

```
foo(10, 20, 30)
```

실행결과

인자의 개수: 3

인자들 : (10, 20, 30)

- len() 함수를 이용하여 다음과 같이 가변적으로 전달된 인자의 개수를 출력하는 것도 가능

- 숫자의 합을 구하는 프로그램
- `sum_num()` 함수에 전달될 인자의 개수를 미리 알 수 없는 경우, 가변인자를 받는 `*numbers`라는 매개변수를 사용하여 전체 인자를 튜플 형식으로 받을 수 있음

코드 4-28 : 가변 인자를 가지는 함수를 이용한 합계 구하기

arg_sum_nums.py

```
def sum_nums(*numbers):  
    result = 0  
    for n in numbers:  
        result += n  
    return result  
  
print(sum_nums(10, 20, 30))          # 10, 20, 30 인자들의 합을 출력  
print(sum_nums(10, 20, 30, 40, 50))  # 10, 20, 30, 40, 50 인자들의 합을 출력
```

실행결과

60

150

재귀함수

- 재귀함수 **recursion**란 함수 내부에서 자기 자신을 호출하는 함수를 말함
- 절차적 기법으로 해결하기 어려운 문제를 직관적이고 간단하게 해결 가능

- 함수 factorial()은 $n! = n * (n-1)!$ 이라는 정의에 맞게 다음과 같이 다시 정의가 가능함

코드 4-29 : 재귀함수를 이용하여 정의한 팩토리얼

factorial_recursion.py

```
def factorial(n):    # n!의 재귀적 구현
    if n <= 1 :      # 종료조건이 반드시 필요하다
        return 1
    else :
        return n * factorial(n-1)    # n * (n-1)! 정의에 따른 구현

n = 5
print('{}! = {}'.format(n, factorial(n)))
```

실행결과

5! = 120



Questions?