AI Programming

Python

02. Python Basic 2

first coding

프로그램의 흐름제어

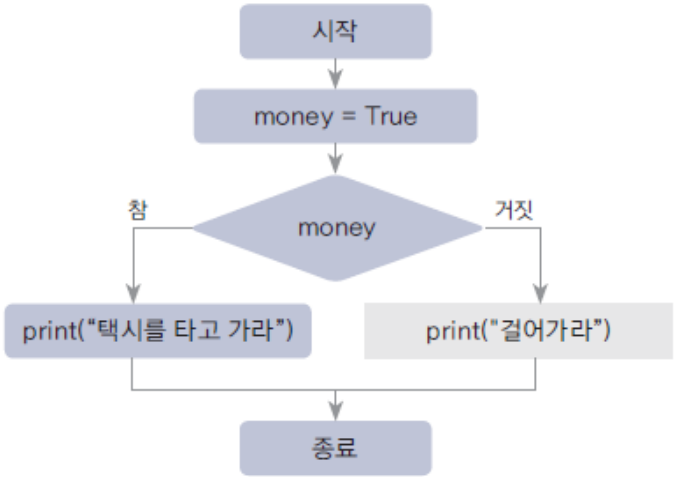
if 문

while 문

for 문

if 문

- if 문
  - 주어진 조건을 판단한 후 그 상황에 맞게 처리해야 할 경우



'돈이 있으면 택시를 타고 가고, 돈이 없으면 걸어간다.'

```
>>> money = True
>>> if money:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
```

if 문

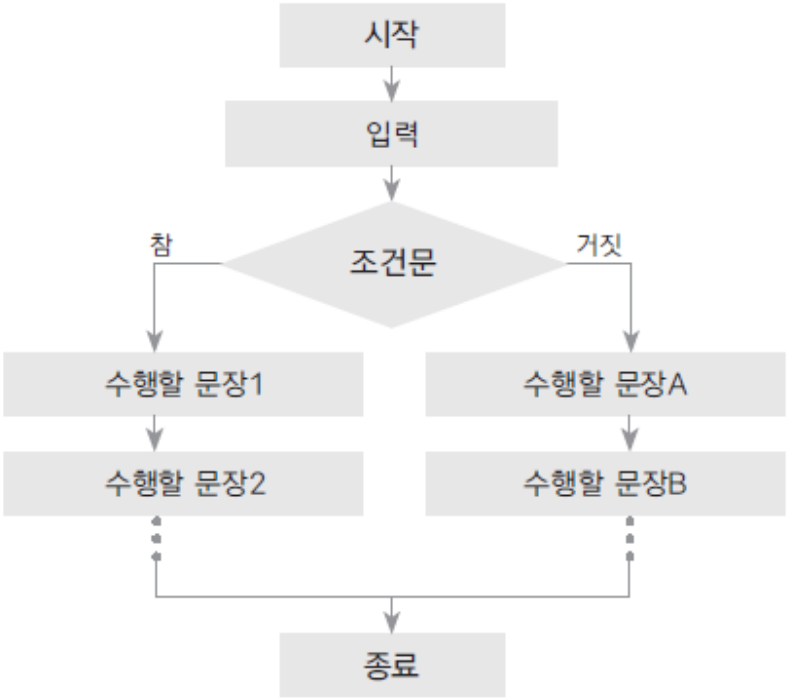
- if 문의 기본 구조

  - if와 else를 사용한 조건문의 기본 구조

```
if 조건문:
    수행할_문장1
    수행할_문장2
        ⋮
else:
    수행할_문장A
    수행할_문장B
        ⋮
```

  - 조건문이 참이면 if 블록 수행
  - 조건문이 거짓이면 else 블록 수행

프로그램의 흐름



if 문

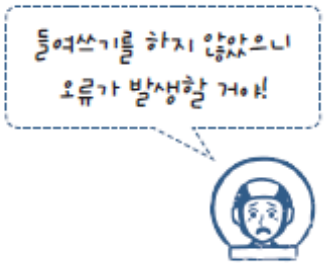
- **들여쓰기 방법 알아보기**
  - if 문을 만들 때는
    if 조건문 바로 다음 문장부터
    모든 문장에 **들여쓰기(indentation)**

```
if 조건문:
    수행할_문장1
    수행할_문장2
    수행할_문장3
```

```
if 조건문:
    수행할_문장1
    수행할_문장2
        수행할_문장3
```

  - 들여쓰기를 무시하는 경우 오류 발생

```
if 조건문:
    수행할_문장1
수행할_문장2
    수행할_문장3
```



if 문

- 조건문
  - if 조건문에서 '조건문'이란 참과 거짓을 판단하는 문장
  - 비교 연산자

비교 연산자	설명
x < y	x가 y보다 작다.
x > y	x가 y보다 크다.
x == y	x와 y가 같다.
x != y	x와 y가 같지 않다.
x >= y	x가 y보다 크거나 같다.
x <= y	x가 y보다 작거나 같다.

```
>>> money = True
>>> if money:
```

```
>>> x = 3
>>> y = 2
>>> x > y  <-- 3 > 2
True
```

```
>>> x < y  <-- 3 < 2
False
```

```
>>> x == y  <-- 3 == 2
False
```

```
>>> x != y  <-- 3 != 2
True
```

if 문

- 조건문
  - 비교 연산자
    - if 조건문에 비교 연산자를 사용하는 예시

> 만약 3000원 이상의 돈을 가지고 있으면 택시를 타고 가고, 그렇지 않으면 걸어가라.

```
>>> money = 2000  <-- 2,000원을 가지고 있다고 설정
>>> if money >= 3000:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
걸어가라
```

if 문

- 조건문
  - and, or, not

연산자	설명
x or y	x와 y 둘 중 하나만 참이어도 참이다.
x and y	x와 y 모두 참이어야 참이다.
not x	x가 거짓이면 참이다.

```
>>> money = 2000   <-- 2,000원을 가지고 있다고 설정
>>> card = True   <-- 카드를 가지고 있다고 설정
>>> if money >= 3000 or card:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
```

if 문

- 조건문
  - in, not in

in	not in
x in 리스트	x not in 리스트
x in 튜플	x not in 튜플
x in 문자열	x not in 문자열

```
>>> 1 in [1, 2, 3] <— 1이 [1, 2, 3] 안에 있는가?
True
>>> 1 not in [1, 2, 3] <— 1이 [1, 2, 3] 안에 없는가?
False
```

```
>>> 'a' in ('a', 'b', 'c')
True
>>> 'j' not in 'python'
True
```

if 문

- 다양한 조건을 판단하는 elif

  - if와 else만으로는
    조건 판단에 어려움이 있음

  - 조건 판단하는 부분

    1) 주머니에 돈이 있는지 판단
    2) 주머니에 돈이 없으면,
       주머니에 카드가 있는지 판단

if와 else만으로는 이해하기 어렵고 산만한 느낌

```
>>> pocket = ['paper', 'cellphone']  <-- 주머니 안에 종이, 휴대전화가 있다.
>>> card = True  <-- 카드를 가지고 있다.
>>> if 'money' in pocket:
...     print("택시를 타고 가라")
... else:
...     if card:
...         print("택시를 타고 가라")
...     else:
...         print("걸어가라")
...
택시를 타고 가라
```

if 문

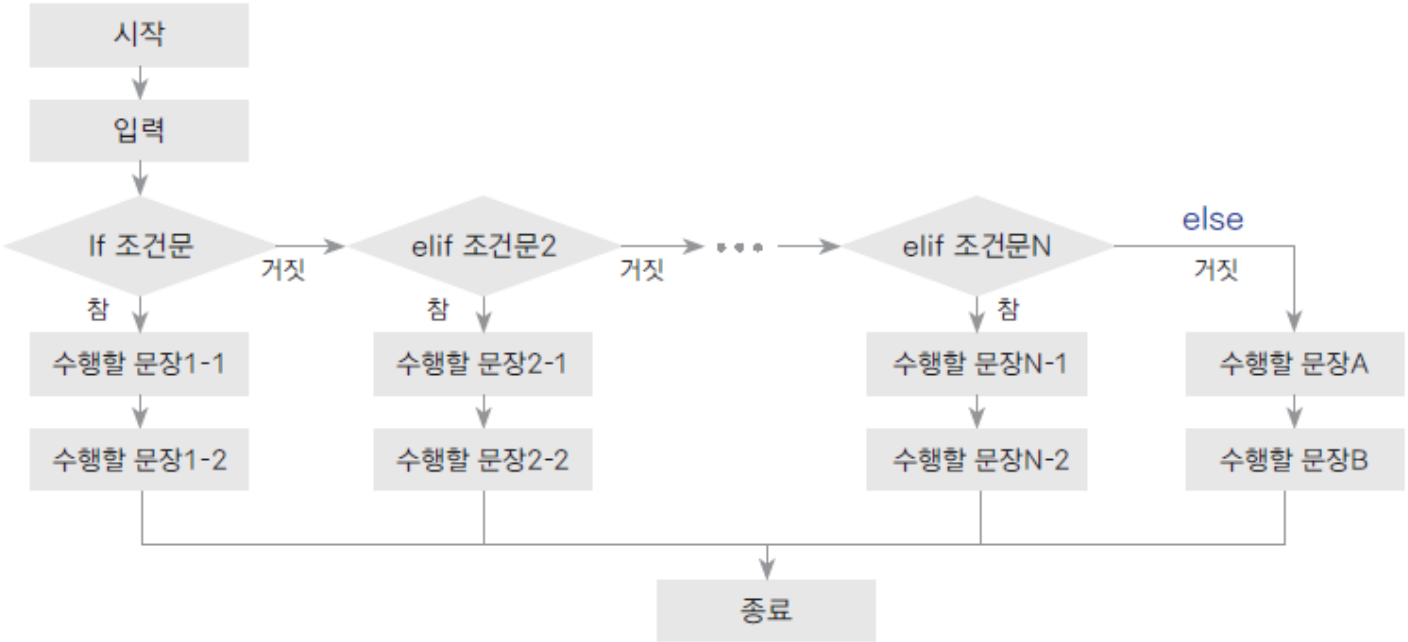
- 다양한 조건을 판단하는 elif
  - elif를 사용한다면?
    - elif는 이전 조건문이 거짓일 때 수행됨

```
if 조건문:
    수행할_문장1
    수행할_문장2
    ...
elif 조건문:
    수행할_문장1
    수행할_문장2
    ...
elif 조건문:
    수행할_문장1
    수행할_문장2
    ...
(...생략...)
else:
   수행할_문장1
   수행할_문장2
   ...
```

```
>>> pocket = ['paper', 'cellphone']
>>> card = True
>>> if 'money' in pocket: <-- 주머니에 돈이 있으면
...     print("택시를 타고 가라")
... elif card: <-- 주머니에 돈이 없고 카드가 있으면
...     print("택시를 타고 가라")
... else: <-- 주머니에 돈이 없고 카드도 없으면
...     print("걸어가라")
...
택시를 타고 가라
```

if 문

- 다양한 조건을 판단하는 elif
  - elif는 개수에 제한 없이 사용 가능



if 문

- **조건부 표현식**

  - 조건부 표현식

```
변수 = 조건문이_참인_경우의_값 if 조건문 else 조건문이_거짓인_경우의_값
```

  - 파이썬의 조건부 표현식(conditional expression) 사용

```
message = "success" if score >= 60 else "failure"
```

if 문

1 | 코딩해보기

01 숫자를 입력 받아 짝수, 홀수를 판별하는 프로그램

1 | 코딩해보기

02 숫자를 입력 받아 양수, 음수, 0을 판별하는 프로그램

1 | 코딩해보기

03 독감예방 접종이 가능한지 여부를 확인하는 프로그램

- 15세 미만 혹은 65세 이상의 경우 무료예방접종 가능 메시지 출력

1 | 코딩해보기

04 특정 연도 건강검진 대상 여부 판별 및 검진 종류 확인

조건

- 매개변수로 올해 연도와 태어난 해(연도)를 전달받음
- 대한민국 성인(20세)의 경우 무료로 2년마다 건강검진을 받을 수 있음
- 짝수 해에 태어난 사람은 올해가 짝수년이라면 검사 대상이 됨
- 40 이상의 경우는 암 검사도 무료로 검사를 할 수 있음

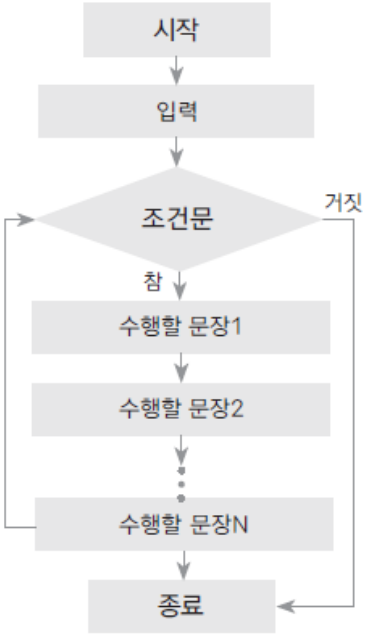
while 문

- while 문의 기본 구조

  - 반복해서 문장을 수행해야 할 경우 while 문 사용

```
while 조건문:
    수행할_문장1
    수행할_문장2
    수행할_문장3
    ...
```

  - while 문은 조건문이 참인 동안에
    while 문에 속한 문장이 반복해서 수행됨



while 문

- while 문 강제로 빠져나가기

  – 강제로 while 문을 빠져나가야 할 때
  break 문 사용


```
>>> coffee = 10  <-- 자판기에 커피가 10개 있다.
>>> money = 300  <-- 자판기에 넣을 돈은 300원이다.
>>> while money:
...     print("돈을 받았으니 커피를 줍니다.")
...     coffee = coffee - 1  <-- while문을 한 번 돌 때마다 커피가 1개씩 줄어든다.
...     print("남은 커피의 양은 %d개입니다." % coffee)
...     if coffee == 0:
...         print("커피가 다 떨어졌습니다. 판매를 중지합니다.")
...         break
...
```

while 문

- while 문의 맨 처음으로 돌아가기
  - while 문을 빠져나가지 않고 while 문의 맨 처음(조건문)으로 다시 돌아가야 할 때 continue사용
  - 1부터 10까지의 숫자 중 홀수만 출력하는 예시

```
>>> a = 0
>>> while a < 10:
...     a = a + 1
...     if a % 2 == 0: continue  <-- a를 2로 나누었을 때 나머지가 0이면 맨 처음으로 돌아간다.
...     print(a)
...
1
3
5
7
9
```

while 문

- **무한 루프**
  - 무한히 반복한다는 뜻의 무한 루프(endless loop)
  - 파이썬에서의 무한 루프는 while 문으로 구현

```
while True:
    수행할_문장1
    수행할_문장2
    ...
```

```
>>> while True:
...     print("Ctrl+C를 눌러야 while 문을 빠져나갈 수 있습니다.")
...
Ctrl+C를 눌러야 while 문을 빠져나갈 수 있습니다.
Ctrl+C를 눌러야 while 문을 빠져나갈 수 있습니다.
Ctrl+C를 눌러야 while 문을 빠져나갈 수 있습니다.
(...생략...)
```

for 문

- for 문의 기본 구조

```
for 변수 in 리스트(또는 튜플, 문자열):
    수행할_문장1
    수행할_문장2
    ...
```

  - 리스트나 튜플, 문자열의 첫 번째 요소부터 마지막 요소까지 차례로 변수에 대입되어 '수행할 문장1', '수행할 문장2' 등이 수행됨

for 문

- 예제를 통해 for 문 이해하기
  - 전형적인 for 문
  - 다양한 for 문의 사용

```
>>> test_list = ['one', 'two', 'three']
>>> for i in test_list:  <-- one, two, three를 순서대로 i에 대입
...     print(i)
...
one
two
three
```

```
>>> a = [(1, 2), (3, 4), (5, 6)]
>>> for (first, last) in a:
...     print(first + last)
...
3  <-- first: 1, last: 2
7  <-- first: 3, last: 4
11 <-- first: 5, last: 6
```

for 문

- 예제를 통해 for 문 이해하기

  – for 문의 응용

  : 총 5명의 학생이 시험을 보았는데 시험 점수가 60점 이상이면 합격이고
  그렇지 않으면 불합격이다.
  합격인지, 불합격인지 결과를 출력 하시오.

```
marks = [90, 25, 67, 45, 80]          # 학생들의 시험 점수 리스트

number = 0                            # 학생에게 붙여 줄 번호
for mark in marks:                    # 90, 25, 67, 45, 80을 순서대로 mark에 대입
    number = number + 1
    if mark >= 60:
        print("%d번 학생은 합격입니다." % number)
    else:
        print("%d번 학생은 불합격입니다." % number)
```

for 문

- for 문과 continue 문

  – for 문 안의 문장을 수행하는 도중 continue 문을 만나면 for 문의 처음으로 돌아감
  : 60점 이상인 사람에게는 축하 메시지를 보내고
  나머지 사람에게는 아무런 메시지도 전하지 않는 프로그램

```
marks = [90, 25, 67, 45, 80]

number = 0
for mark in marks:
    number = number + 1
    if mark < 60:
        continue
    print("%d번 학생 축하합니다. 합격입니다. " % number)
```

for 문

- for 문과 함께 자주 사용하는 range 함수

  - 숫자 리스트를 자동으로 만들어주는 함수
    - range(10)
      : 0부터 10 미만의 숫자를 포함하는 range 객체
  - range(a, b, c)
    - a: 시작 숫자
    - b: 끝 숫자 (반환 범위에 포함되지 않음 : 탈출 숫자)
    - c: step (숫자 사이의 간격)

```
>>> a = range(10)
>>> a
range(0, 10) <- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

```
>>> a = range(1, 11)
>>> a
range(1, 11) <- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

for 문

- for 문과 함께 자주 사용하는 range 함수

  - range 함수의 예시
    - for와 range 함수를 사용하여 1부터 10까지 더하기

```
>>> add = 0
>>> for i in range(1, 11):
...     add = add + i
...
>>> print(add)
55
```

for 문

- for 문과 함께 자주 사용하는 range 함수

  - for와 range를 이용한 구구단 (반복문의 중첩)
    - ①번 for문
      - 2부터 9까지의 숫자(range(2, 10))가 차례로 i에 대입됨
    - ②번 for문
      - 1부터 9까지의 숫자(range(1, 10))가 차례로 j에 대입됨
      - print(i*j) 수행

```
>>> for i in range(2, 10):          ← ①번 for 문
...     for j in range(1, 10):      ← ②번 for 문
...         print(i*j, end=" ")
...     print('')
...
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

for 문

- **리스트 컴프리헨션(list comprehension) 사용하기**

  – 리스트 안에 for 문 포함하기

    - a 리스트의 각 항목에 3을 곱한 결과를 result 리스트에 담기
    - **리스트 컴프리헨션을 사용하도록 수정**

```
>>> a = [1, 2, 3, 4]
>>> result = []
>>> for num in a:
...     result.append(num*3)
...
>>> print(result)
[3, 6, 9, 12]
```

```
>>> a = [1, 2, 3, 4]
>>> result = [num*3 for num in a]
>>> print(result)
[3, 6, 9, 12]
```

for 문

- **리스트 컴프리헨션**(list comprehension) **사용하기**

  - 리스트 안에 for 문 포함하기
    - 리스트 컴프리헨션 안에 'if 조건' 사용 가능
    - [1, 2, 3, 4] 중에서 짝수에만 3을 곱하여 담도록 수정

```
>>> a = [1, 2, 3, 4]
>>> result = [num*3 for num in a if num%2 == 0]
>>> print(result)
[6, 12]
```

for 문

- **리스트 컴프리헨션**(list comprehension) **사용하기**

  - 리스트 컴프리헨션 문법
    - 'if 조건문' 부분은 생략 가능
    - for 문 여러 개 사용 가능

```
[표현식 for 항목 in 반복_가능_객체 if 조건문]
```

```
[표현식 for 항목1 in 반복_가능_객체1 if 조건문1
       for 항목2 in 반복_가능_객체2 if 조건문2
       ...
       for 항목n in 반복_가능_객체n if 조건문n]
```

반복문

1 | 코딩해보기

01 1 부터 100까지의 합을 구하는 프로그램 작성

1 | 코딩해보기

02 for 문을 이용하여 1부터 10까지를 곱한 결과를 출력하는 프로그램 작성

1 | 코딩해보기

03 구구단의 짝수 단(2,4,6,8)만 출력하는 프로그램 작성

- 단, 2단은 2x2까지, 4단은 4x4까지 , 6단은 6x6까지, 8단은 8x8 까지 출력

1 | 코딩해보기

04 while문을 무한 루프로 구성하여 작성하는 예제

- 1부터 시작해서 모든 짝수와 3의 배수를 더해서 그 합이 언제 10000이 넘어서는지, 그리고 10000이 넘어선 값은 얼마가 되는지 계산하여 출력

first coding

함수

함수의 구조

함수

- **함수란 무엇인가?**
  - 우리는 믹서에 과일을 넣고, 믹서를 사용해서 과일을 갈아 과일 주스를 만듦
  - 믹서에 넣는 과일 = 입력
  - 과일주스 = 출력(결과 값)
  - 믹서 = ?

믹서는 과일을 입력받아 주스를 출력하는 함수와 같다.

함수

- 함수를 사용하는 이유는 무엇일까?

  - **반복되는 부분(코드 들)**이 있을 경우
    '**반복적으로 사용되는 부분**'을 한 묶음으로 묶어
    '**약속된 입력 값을 주었을 때 약속된 결과 값을 반환해 준다**'
    혁식으로 함수로 작성
    → **프로그램의 코드의 가독성 향상** : 오류 해결과 유지보수에 좋음

함수

- 파이썬 함수의 구조

  - def : 함수를 만들 때 사용하는 예약어
  - 함수 이름은 임의로 생성 가능
  - 매개변수는 함수에 입력 전달되는 값을 받는 변수
  - return : 함수의 결과 값(리턴 값)을 돌려주는 명령어

```
def 함수_이름(매개변수):
    수행할_문장1
    수행할_문장2
    ...
```

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

함수

- 파이썬 함수의 구조

  - 예) add 함수
    - add 함수 만들기
    - add 함수 사용하기

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

```
>>> a = 3
>>> b = 4
>>> c = add(a, b) <-- add(3, 4)의 리턴값을 c에 대입
>>> print(c)
7
```

함수

- 매개변수와 인수

  - **매개변수**와 인수는 혼용해서 사용되는 헷갈리는 용어로 잘 구분하는 것이 중요!
  - **매개변수**(parameter)
    - 함수에 입력 받는 값을 저장하기 위한 **변수**
  - **인수**(arguments)
    - 함수를 호출할 때 전달하는 값

```
def add(a, b): <- a, b는 매개변수
    return a + b

print(add(3, 4)) <- 3, 4는 인수
```

함수

- **입력 값과 리턴 값에 따른 함수의 형태**

  - 함수는 매개변수를 통해 값을 받은 후 어떤 처리를 하여 적절한 값을 반환해 줌

  - 함수의 형태는 입력 값과 리턴 값의 존재 유무에 따라 구분해서 사용

    → 매개변수와 return 은 선택적으로 사용

함수

- **입력 값과 리턴 값에 따른 함수의 형태**

  – 일반적인 함수

    - 입력 값이 있고 리턴 값이 있는 함수

```
def 함수_이름(매개변수):
    수행할_문장
    ...
    return 리턴값
```

```
리턴값을_받을_변수 = 함수_이름(입력_인수1, 입력_인수2, ...)
```

```
>>> def add(a, b):
...     result = a + b
...     return result  <-- a + b의 결괏값 리턴
```

```
>>> a = add(3, 4)
>>> print(a)
7
```

함수

- **입력 값과 리턴 값에 따른 함수의 형태**

  - 입력 값이 없는 함수 (매개변수가 정의되지 않은 함수)
    - 입력 값이 없는 함수도 존재함 → 처리의 목적인 데이터를 받아서 처리하지 않을 때
    - say 함수는 매개변수 부분을 나타내는 함수 이름 뒤의 괄호 안이 비어있음
    - 함수 사용 시 say( )처럼 괄호 안에 아무 값도 넣지 않아야 함

```
>>> def say():
...     return 'Hi'
```

```
>>> a = say()
>>> print(a)
Hi
```

함수

- **입력 값과 리턴 값에 따른 함수의 형태**

  - 리턴 값이 없는 함수
    - 리턴 값이 없는 함수는 호출해도 리턴 되는 값이 없음

```
>>> def add(a, b):
...     print("%d, %d의 합은 %d입니다." % (a, b, a + b))
```

```
>>> add(3, 4)
3, 4의 합은 7입니다.
```

함수

- **입력 값과 리턴 값에 따른 함수의 형태**

  – 리턴 값이 진짜 없을까?

    - None 데이터 반환

      → None : 거짓을 나타내는 자료형

```
>>> a = add(3, 4) <— add 함수의 리턴값을 a에 대입
3, 4의 합은 7입니다.
>>> print(a) <— a 값 출력
None
```

함수

- **입력 값과 리턴 값에 따른 함수의 형태**

  - 입력 값도, 리턴 값도 없는 함수
    - 매개변수도 없고 return 문도 없는 함수

```
>>> def say():
...     print('Hi')
```

```
>>> say()
Hi
```

함수

- **매개변수를 지정하여 호출하기**

  – 함수 호출 시 매개변수 지정 가능

    - 매개변수를 지정하여 사용

      → **매개변수 순서에 상관없이 사용**할 수 있는 장점

```
>>> def sub(a, b):
...     return a - b
```

```
>>> result = sub(a=7, b=3) <— a에 7, b에 3을 전달
>>> print(result)
4
```

```
>>> result = sub(b=5, a=3) <— b에 5, a에 3을 전달
>>> print(result)
-2
```

함수

- **입력 값이 몇 개가 될지 모를 때는 어떻게 해야 할까?**

  - 파이썬에서의 해결 방법
    - 일반 함수 형태에서 괄호 안의 매개변수 부분이 *매개변수로 바뀜

```
def 함수_이름(*매개변수):
    수행할_문장
    ...
```

함수

- **입력 값이 몇 개가 될지 모를 때는 어떻게 해야 할까?**

  – 여러 개의 입력 값을 받는 함수 만들기

  - 매개변수 이름 앞에 *을 붙이면

    → 입력 값을 전부 모아 튜플로 만들어 줌

```
>>> def add_many(*args):
...     result = 0
...     for i in args:
...         result = result + i  <— *args에 입력받은 모든 값을 더한다.
...     return result
```

```
>>> result = add_many(1, 2, 3) <— add_many 함수의 리턴값을 result 변수에 대입
>>> print(result)
6
>>> result = add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(result)
55
```

함수

- **입력 값이 몇 개가 될지 모를 때는 어떻게 해야 할까?**

  - 여러 개의 입력 값을 받는 함수 만들기
    - *args 매개변수 앞에 choice 매개변수를 추가할 수도 있음

```
>>> def add_mul(choice, *args):
...     if choice == "add":  <- 매개변수 choice에 "add"를 입력받았을 때
...         result = 0
...         for i in args:
...             result = result + i  <- args에 입력받은 모든 값을 더한다.
...     elif choice == "mul":  <- 매개변수 choice에 "mul"을 입력받았을 때
...         result = 1
...         for i in args:
...             result = result * i  <- *args에 입력받은 모든 값을 곱한다.
...     return result
```

```
>>> result = add_mul('add', 1, 2, 3, 4, 5)
>>> print(result)
15
>>> result = add_mul('mul', 1, 2, 3, 4, 5)
>>> print(result)
120
```

함수

- 키워드 매개변수, kwargs
  - 매개변수 앞에 별 2개(**)를 붙임

```
>>> def print_kwargs(**kwargs):
...     print(kwargs)
```

```
>>> print_kwargs(a=1)
{'a': 1}
>>> print_kwargs(name='foo', age=3)
{'age': 3, 'name': 'foo'}
```

함수

- 함수의 리턴 값은 언제나 하나이다

  - 2개의 입력 인수를 받아 리턴 하는 함수

    → 하나의 튜플 값을 2개의 값으로 분리하여 리턴

  - return 문을 2번 사용하면 리턴 값은 하나뿐

    → return 의 또 다른 의미는 함수의 종료

```
>>> def add_and_mul(a, b):
...     return a+b, a*b
```

```
>>> result = add_and_mul(3, 4)
```

```
result = (7, 12)
```

```
>>> result1, result2 = add_and_mul(3, 4)
```

```
>>> def add_and_mul(a, b):
...     return a + b
...     return a * b
```

```
>>> result = add_and_mul(2, 3)
>>> print(result)
5
```

함수

- **매개변수에 초기 값 미리 설정하기**

  – 매개변수에 초기 값을 미리 설정

    - man=True처럼 매개변수에 미리 값을 넣어 함수의 매개변수 초기 값을 설정

```
def say_myself(name, age, man=True):
    print("나의 이름은 %s입니다." % name)
    print("나이는 %d살입니다." % age)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

함수

- 매개변수에 초기 값 미리 설정하기

  - 주의할 점
    - 초기화하고 싶은 매개변수는 항상 뒤쪽에 놓아야 함
    - say_myself('king', 27)
      → 파이썬 인터프리터는 27을 man 매개변수와 age
      매개변수 중 어느 곳에 대입해야 할지 판단이 어려워 오류 발생

```
def say_myself(name, man=True, age):
    print("나의 이름은 %s입니다." % name)
    print("나이는 %d살입니다." % age)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

실행 결과

```
SyntaxError: non-default argument follows default argument
```

함수

- **함수 안에서 선언한 지역 변수의 효력 범위**

  - 함수 안에서 사용할 변수의 이름을 함수 밖에서도 동일하게 사용한다면?

    → **매개변수 a는 함수 안에서만 사용하는 변수**일 뿐, 함수 밖의 변수 a와는 전혀 상관없음

```
a = 1              # 함수 밖의 변수 a
def vartest(a):    # vartest 함수 선언
    a = a + 1

vartest(a)         # vartest 함수의 입력값으로 a를 대입
print(a)           # a 값 출력
```

함수

- **함수 안에서 선언한 변수의 효력 범위**

  - 함수 안에서 선언한 매개변수는 함수 안에서만 사용될 뿐, 함수 밖에서는 사용되지 않음
  - print(a)에서 사용한 a 변수는 어디에도 선언되지 않았기 때문에 오류 발생

```
def vartest(a):
    a = a + 1

vartest(3)
print(a)
```

함수

- **함수 안에서 함수 밖에서 선언된 변수 사용 방법**

  - global 명령어 사용하기
  - 단, 함수는 독립적으로 존재하는 것이 좋기 때문에 특정 변수에 의존하지 않도록 해야함
    → 이 방법은 피하는 것이 좋음

```
a = 1
def vartest():
    global a
    a = a + 1

vartest()
print(a)
```

함수

- lambda 예약어

  - 함수를 생성할 때 사용하는 예약어
  - 함수를 한 줄로 간결하게 만들 때 사용
  - def와 동일한 역할
  - 함수를 매개변수로 받아서 처리하는 함수에 주로 사용

```
함수_이름 = lambda 매개변수1, 매개변수2, ... : 매개변수를_이용한_표현식
```

함수

- lambda 예약어

  – add는 2개의 인수를 받아
    서로 더한 값을 리턴 하는 lambda 함수

```
>>> add = lambda a, b: a + b
>>> result = add(3, 4)
>>> print(result)
7
```

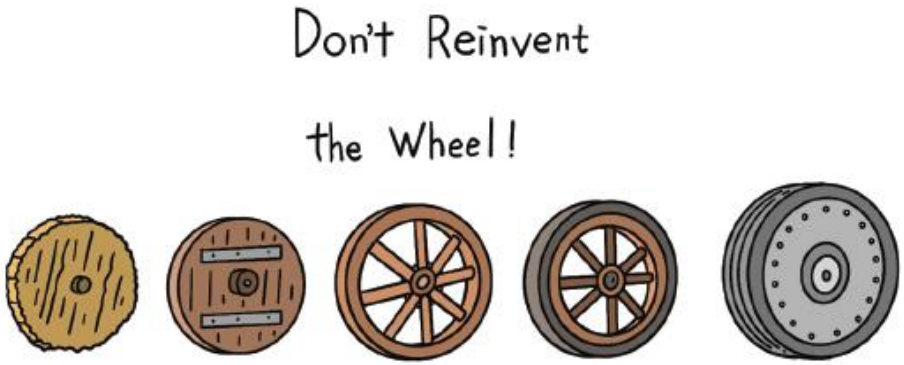
```
>>> def add(a, b):
...     return a + b
...
>>> result = add(3, 4)
>>> print(result)
7
```

first
coding

내장함수

내장 함수

- **파이썬 내장(built-in) 함수**
  - 파이썬 모듈과 달리 import가 필요 없기 때문에 **아무런 설정 없이 바로 사용 가능**



이미 있는 것을 다시 만드느라 시간을 낭비하지 말라.

내장 함수

- abs(x)
  - 숫자를 입력 받아
    그 숫자의 절대 값을 돌려주는 함수
- all(x)
  - 시퀀스 데이터 x를 입력 받아
    x의 요소가 모두 참이면 True,
    하나라도 거짓이면 False를 리턴
- any(x)
  - 시퀀스 데이터 x를 입력 받아
    x의 요소 중 하나라도 참이면 True,
    x가 모두 거짓일 때만 False를 리턴

```
>>> abs(3)
3
>>> abs(-3)
3
>>> abs(-1.2)
1.2
```

```
>>> all([1, 2, 3])
True
```

```
>>> any([1, 2, 3, 0])
True
```

```
>>> all([1, 2, 3, 0])
False
```

```
>>> any([0, ""])
False
```

```
>>> all([])
True
```

```
>>> any([])
False
```

내장 함수

- chr(i)
  - 유니코드 숫자 값을 받아
    그 코드에 해당하는 문자를 리턴
- dir(x)
  - 객체가 지닌 변수나 함수를 보여 주는 함수
    - 리스트와 딕셔너리가 지닌 함수(메서드)
- divmod(a, b)
  - a를 b로 나눈 몫과 나머지를 튜플로 리턴
  - 몫을 구하는 연산자 //와 나머지를 구하는 연산자 %를 각각 사용한 결과와 비교

```
>>> chr(97)
'a'
>>> chr(44032)
'가'
```

```
>>> dir([1, 2, 3])
['append', 'count', 'extend', 'index', 'insert', 'pop',...]
>>> dir({'1':'a'})
['clear', 'copy', 'get', 'has_key', 'items', 'keys',...]
```

```
>>> divmod(7, 3)
(2, 1)
```

```
>>> 7 // 3
2
>>> 7 % 3
1
```

내장 함수

- enumerate(x)
  - '열거하다'라는 뜻
  - 시퀀스 데이터(리스트, 튜플, 문자열)를 입력 받아
    인덱스 값을 포함하는 enumerate 객체 리턴
- hex(x)
  - 정수 값을 입력 받아
    16진수(hexadecimal) 문자열로 리턴
- id(object)
  - 객체(object)를 입력 받아
    객체 고유 주소 값(레퍼런스)을 리턴

```
>>> for i, name in enumerate(['body', 'foo', 'bar']):
...     print(i, name)
...
0 body
1 foo
2 bar
```

```
>>> hex(234)
'0xea'
>>> hex(3)
'0x3'
```

```
>>> a = 3
>>> id(3)
135072304
>>> id(a)
135072304
```

내장 함수

- filter(f, iterable)
  - '무엇인가를 걸러 낸다'는 뜻

```
filter(함수, 반복_가능한_데이터)
```

  - 시퀀스 데이터의 요소 값을 순서대로 함수에 전달하며 함수를 호출했을 때 리턴 값이 참인 것만 묶어서 (걸러 내서) 리턴
  - filter 함수를 사용해 간단하게 작성
  - lambda를 사용해 더욱 간단하게 작성

```
def positive(l):
    result = []                 # 양수만 걸러 내서 저장할 변수
    for i in l:
        if i > 0:
            result.append(i)    # 리스트에 i 추가
    return result

print(positive([1, -3, 2, 0, -5, 6]))
```

```
def positive(x):
    return x > 0

print(list(filter(positive, [1, -3, 2, 0, -5, 6])))
```

```
>>> list(filter(lambda x: x > 0, [1, -3, 2, 0, -5, 6]))
[1, 2, 6]
```

내장 함수

- map(f, iterable)
  - 함수(f)와 시퀀스 데이터를 입력 받음
  - 입력 받은 데이터의 각 요소를
    함수 f에 전달하고 결과를 리턴 받아
    결과 값들을 리스트로 반환
  - lambda 활용 가능

```
>>> def two_times(x):
...     return x * 2
...
>>> list(map(two_times, [1, 2, 3, 4]))
[2, 4, 6, 8]
```

```
>>> list(map(lambda a: a*2, [1, 2, 3, 4]))
[2, 4, 6, 8]
```

내장 함수

- int(x)
  - 문자열 형태의 숫자나 소수점이 있는 숫자 를 정수로 리턴
  - radix 진수로 표현된 문자열 x를 10진수로 변환하여 리턴
- str(object)
  - 문자열 형태로 객체를 변환하여 리턴 하는 함수
- list(iterable)
  - 시퀀스 데이터를 입력 받아 리스트로 만들어 리턴
- tuple(iterable)
  - 시퀀스 데이터를 튜플로 바꾸어 리턴 하는 함수
  - 입력이 튜플인 경우 그대로 리턴

```
>>> int('3')  <- 문자열 '3'
3
>>> int(3.4)  <- 소수점이 있는 숫자 3.4
3
```

```
>>> int('11', 2)
3
```

```
>>> int('1A', 16)
26
```

```
>>> str(3)
'3'
>>> str('hi')
'hi'
```

```
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
>>> list((1, 2, 3))
[1, 2, 3]
```

```
('a', 'b', 'c')
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple((1, 2, 3))
(1, 2, 3)
```

내장 함수

- max(iterable)
  - 시퀀스 데이터를 입력 받아 그 최대 값을 리턴
- min(iterable)
  - 시퀀스 데이터를 입력 받아 그 최소 값을 리턴
- sum(iterable)
  - 입력 데이터의 합을 리턴 하는 함수
- sorted(iterable)
  - 입력 데이터를 정렬한 후 그 결과를 리스트로 리턴 하는 함수

```
>>> max([1, 2, 3])
3
>>> max("python")  <- 문자열의 경우, 유니코드 값이 가장 큰 문자를 리턴
'y'
```

```
>>> min([1, 2, 3])
1
>>> min("python")
'h'
```

```
>>> sum([1, 2, 3])
6
>>> sum((4, 5, 6))
15
```

```
>>> sorted([3, 1, 2])
[1, 2, 3]
>>> sorted(['a', 'c', 'b'])
['a', 'b', 'c']
>>> sorted("zero")
['e', 'o', 'r', 'z']
>>> sorted((3, 2, 1))
[1, 2, 3]
```

내장 함수

- len(s)
  - 입력 값 s의 길이(요소의 전체 개수)를 리턴
- type(object)
  - 입력 값의 자료형이 무엇인지 알려 주는 함수
- round(number[, ndigits])
  - 숫자를 입력 받아 반올림해 리턴 하는 함수
  - ,ndigits는 반올림하여 표시하고 싶은 소수점의 자릿수를 의미

```
>>> len("python")
6
>>> len([1, 2, 3])
3
>>> len((1, 'a'))
2
```

```
>>> type("abc")
<class 'str'>  <-- "abc"는 문자열 자료형
>>> type([])
<class 'list'> <-- []는 리스트 자료형
>>> type(open("test", 'w'))
<class '_io.TextIOWrapper'> <-- 파일 자료형
```

```
>>> round(4.6)
5
>>> round(4.2)
4
```

first coding

파이썬의 입출력

사용자 입출력

파일 읽고 쓰기

프로그램의 입출력

사용자 입출력

- **사용자 입력 활용하기**

  - input 함수 사용하기
    - 사용자가 키보드로 입력한 데이터를 문자열로 반환
  - 프롬프트를 띄워 사용자 입력 받기
    - 사용자에게 입력 받을 때 안내 문구 보여 주기

```
>>> a = input()
Life is too short, you need python  <-- 사용자가 문장을 입력
>>> a
'Life is too short, you need python'
```

```
>>> number = input("숫자를 입력하세요: ")
숫자를 입력하세요: 3  <-- 3 입력
>>> print(number)
3
```

사용자 입출력

- print 함수
  - 데이터를 출력하는 데 사용
  - 큰따옴표로 둘러싸인 문자열은 + 연산과 동일하다
  - 문자열 띄어쓰기는 쉼표로 한다
  - 한 줄에 결과 값 출력하기

```
>>> a = 123
>>> print(a) <-- 숫자 출력하기
123
>>> a = "Python"
>>> print(a) <-- 문자열 출력하기
Python
>>> a = [1, 2, 3]
>>> print(a) <-- 리스트 출력하기
[1, 2, 3]
```

```
>>> print("life" "is" "too short") <-- ①
lifeistoo short
>>> print("life"+"is"+"too short") <-- ②
lifeistoo short
```

```
>>> print("life", "is", "too short")
life is too short
```

```
>>> for i in range(10):
...     print(i, end = ' ')
...
0 1 2 3 4 5 6 7 8 9 >>>
```

파일 읽고 쓰기

- **파일 생성하기**

  - 사용자가 직접 '입력'하고 모니터 화면에 결과 값을 '출력'하는 방법만 있는 것은 아님
  - 파일을 통한 입출력도 가능

```
f = open("새파일.txt", 'w')
f.close()
```

  - 소스코드를 실행하면 프로그램을 실행한 디렉터리에 새로운 파일이 하나 생성됨
  - **파일을 생성하기 위해 파이썬 내장 함수 open을 사용**

```
파일_객체 = open(파일_이름, 파일_열기_모드)
```

  - f.close( )는 열려 있는 파일 객체를 닫아 주는 역할

파일 읽고 쓰기

- **파일 생성하기**

  – 파일 열기 모드

파일 열기 모드	설명
r	읽기 모드: 파일을 읽기만 할 때 사용한다.
w	쓰기 모드: 파일에 내용을 쓸 때 사용한다.
a	추가 모드: 파일의 마지막에 새로운 내용을 추가할 때 사용한다.

  - 파일을 쓰기 모드(w)

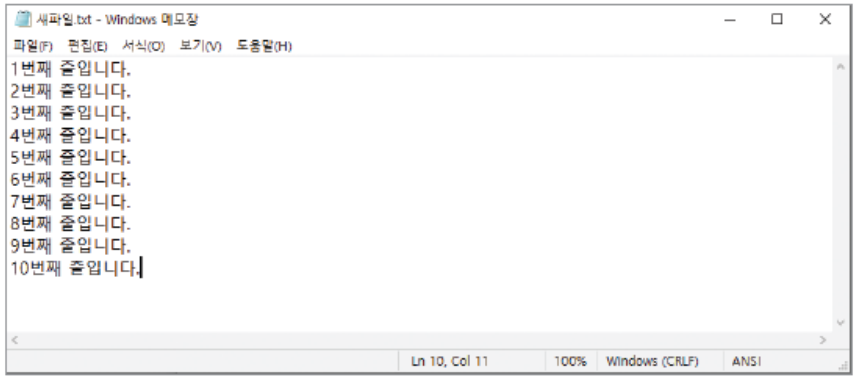
    → 기존 데이터를 덮어 씀

    → 해당 파일이 존재하지 않으면 새로운 파일이 생성됨

파일 읽고 쓰기

- **파일을 쓰기 모드로 열어 내용 쓰기**
  - 문자열 데이터를 파일에 직접 써서 출력
  - write 메서드 이용

```
f = open("새파일.txt", 'w')
for i in range(1, 11):
    data = "%d번째 줄입니다.\n" % i
    f.write(data)    # data를 파일 객체 f에 써라.
f.close()
```



파일 읽고 쓰기

- 파일을 읽기

  – readline 함수 사용하기

    - f = open("새파일.txt", 'r')

      → readline( )을 사용해서 파일의 첫 번째 라인 읽기

```
line = f.readline()
print(line)
f.close()
```

```
1번째 줄입니다.
```

    - 라인 단위로 읽어 출력하는 코드

      → 무한루프 안에서 f.readline( )을 사용

      → 더 이상 읽을 줄이 없으면 break 수행

      → readline( )은 더 이상 읽을 줄이 없을 경우 빈 문자열('')을 리턴

```
while True:
    line = f.readline()
    if not line: break
    print(line)
f.close()
```

파일 읽고 쓰기

- **파일을 읽는 여러 가지 방법**

  - readlines 메서드 사용
    - 파일의 모든 줄을 읽어서 각각의 줄을 요소로 가지는 리스트를 리턴
    - [“1번째 줄입니다.₩n”, “2번째 줄입니다.₩n”, …, “10번째 줄입니다.₩n”]를 리턴

```
lines = f.readlines()
for line in lines:
    print(line)
f.close()
```

파일 읽고 쓰기

- **파일을 읽는 여러 가지 방법**

  - read 함수 사용하기
    - f.read( )는 파일의 내용 전체를 문자열로 리턴
    - data는 파일의 전체 내용

```
data = f.read()
print(data)
f.close()
```

파일 객체를 for 문과 함께 사용하기

파일 객체(f)는 for 문과 함께 사용하여 파일을 줄 단위로 읽을 수 있음

```
for line in f:
    print(line)
f.close()
```

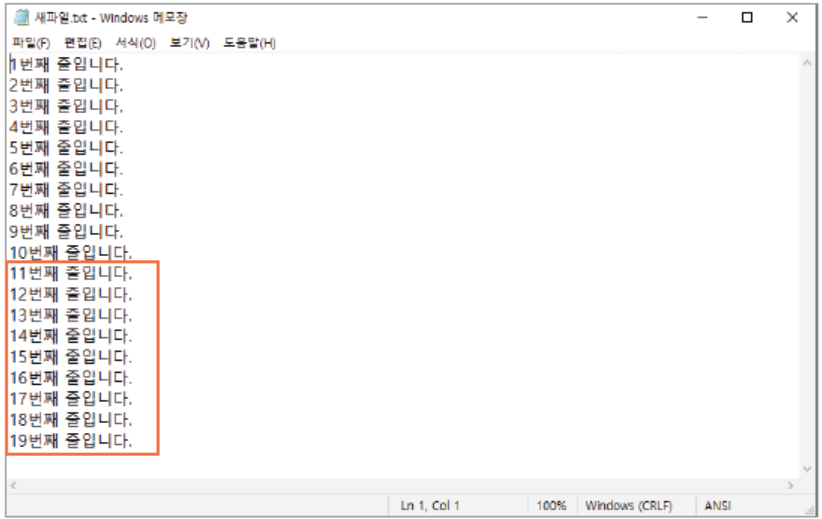
파일 읽고 쓰기

- **파일에 새로운 내용 추가하기**

  - 원래 있던 값을 유지하면서 단지 새로운 값만 추가해야 할 경우
  - 파일을 추가 모드('a')로 열기

    f = open("새파일.txt", 'a')

```
for i in range(11, 20):
    data = "%d번째 줄입니다.\n" % i
    f.write(data)
f.close()
```



파일 읽고 쓰기

- with 문과 함께 사용하기

  - 지금까지 파일을 열고 닫은 방법

```
f = open("foo.txt", 'w') <-- 파일 열기
f.write("Life is too short, you need python")
f.close() <-- 파일 닫기
```

    - f.close( )는 열려 있는 파일 객체를 닫아 주는 역할
    - 쓰기 모드로 열었던 파일을 닫지 않고 다시 사용하면 오류가 발생하기 때문에, close( )를 사용해서 열려 있는 파일을 직접 닫아 주어야 함.

파일 읽고 쓰기

- with 문과 함께 사용하기

  - with 문은 파일을 열고 닫는 것을 자동으로 처리해주는 문법
  - 앞선 예제를 with 문을 사용하여 수정한 코드

```
with open("foo.txt", "w") as f:
    f.write("Life is too short, you need python")
```

  - with 문을 사용하면 with 블록을 벗어나는 순간 열린 파일 객체 f가 자동으로 닫힘

first
coding

예외처리

예외 처리

- **오류 예외 처리 기법**

  - try-except 문

    1) try-except만 쓰는 방법

       - 오류 종류에 상관없이
         오류가 발생하면 except 블록 수행

```
try:
    ...
except:
    ...
```

    2) 발생 오류만 포함한 except 문

       - 오류가 발생했을 때 except 문에
         미리 정해 놓은 오류와 동일할 때만
         except 블록을 수행한다는 뜻

```
try:
    ...
except 발생_오류:
    ...
```

예외 처리

- **오류 예외 처리 기법**

  – try-except 문

    3) 발생 오류와 오류 변수까지 포함한 except 문

    → 오류가 발생했을 때

    except 문에 미리 정해 놓은 오류와 동일할 때만

    except 블록을 수행

```
try:
    ...
except 발생_오류 as 오류_변수:
    ...
```

```
try:
    4 / 0
except ZeroDivisionError as e:
    print(e)
```

예외 처리

- 오류 예외 처리 기법

  - try-finally 문
    - finally 절은 try 문 수행 도중
      예외 발생 여부에 상관없이 항상 수행됨
    - 보통 finally 절은 사용한 리소스를
      close해야 할 때 많이 사용

```
try:
    f = open('foo.txt', 'w')
    # 무언가를 수행

    (...생략...)

finally:
    f.close()    # 중간에 오류가 발생하더라도 무조건 실행
```

예외 처리

- 오류 예외 처리 기법

  - 여러 개의 오류 처리하기

    - try문 안에서
      여러 개의 오류를 처리하기 위한 방법

```
try:
    ...
except 발생_오류1:
    ...
except 발생_오류2:
    ...
```

```
try:
    a = [1, 2]
    print(a[3])
    4 / 0
except ZeroDivisionError:
    print("0으로 나눌 수 없습니다.")
except IndexError:
    print("인덱싱할 수 없습니다.")
```

예외 처리

- 오류 예외 처리 기법

  - 여러 개의 오류 처리하기
    - 오류 메시지 가져오기
    - 2개 이상의 오류를 동일하게 처리하기 위해 괄호를 사용하여 함께 묶어 처리

```
try:
    a = [1, 2]
    print(a[3])
    4 / 0
except ZeroDivisionError as e:
    print(e)
except IndexError as e:
    print(e)
```

```
try:
    a = [1, 2]
    print(a[3])
    4 / 0
except (ZeroDivisionError, IndexError) as e:
    print(e)
```

예외 처리

- 오류 예외 처리 기법
  - try-else 문
    - 오류 메시지 가져오기
    - try 문에 else 절을 사용한 예제

```
try:
    ...
except [발생_오류 [as 오류_변수]]:
    ...
else:    ← 오류가 없을 경우에만 수행
    ...
```

```
try:
    age = int(input('나이를 입력하세요: '))
except:
    print('입력이 정확하지 않습니다.')
else:
    if age <= 18:
        print('미성년자는 출입금지입니다.')
    else:
        print('환영합니다.')
```

예외 처리

- 오류 회피하기

  - 특정 오류가 발생할 경우 그냥 통과시키는 방법

```
try:
    f = open("나없는파일", 'r')
except FileNotFoundError:    # 파일이 없더라도 오류가 발생하지 않고 통과
    pass
```

    - try 문 안에서 FileNotFoundError가 발생할 경우, pass를 사용하여 오류를 그냥 회피

예외 처리

- 오류 강제 발생시키기

  - raise 명령어를 사용해 오류를 강제로 발생시킬 수 있음
    - raise MyError()
    - raise NotImplementedError()

예외 처리

- 사용자 정의 예외 만들기

  – 파이썬 내장 클래스인 Exception 클래스를 상속하여 생성 가능

```
class MyError(Exception):
    pass
```

  - 별명을 출력해주는 함수에서 MyError 적용

```
def say_nick(nick):
    if nick == '바보':
        raise MyError()
    print(nick)
```

```
say_nick("바보")
```

예외 처리

- 예외 만들기

  - 파이썬 내장 클래스인 Exception 클래스를 상속하여 생성 가능

```
class MyError(Exception):
    pass
```

  - MyError에서 __str__ 메서드 구현하여 오류 메시지 사용하기

```
try:
    say_nick("천사")
    say_nick("바보")
except MyError as e:
    print(e)
```

```
class MyError(Exception):
    def __str__(self):
        return "허용되지 않는 별명입니다."
```