❖ 학습 내용

- Python 함수
- Python 모듈과 패키지
- Python 예외처리

Python 함수

❖ 함 수

◆ 정의

- 여러 개의 Statement들을 하나로 묶은 단위
- 자주 사용되는 코드를 별도로 만들어 놓고 필요시 불러서 재사용하는 방법
- 기능의 최소 단위

◆ 장점

- 반복적인 수행이 가능 → 코드의 길이를 짧게 만들어준다.
- 코드의 유지 보수를 쉽게 만들어 준다.
- 코드를 논리적으로 이해하는 데 도움을 준다
- 코드의 일정 부분을 별도의 논리적 개념으로 독립화할 수 있음
- 수학에서 복잡한 개념을 하나의 단순한 기호로 대치하는 것과 비슷

➡ ex 1]

```
def add(a, b):
    return(a + b)
print(add(3, 4))
print(add([1,2,3], [4,5,6]))
```

```
7
```

➡ 설 명 1]

- 코드의 양이 많을 경우 함수를 사용하여 한번에 호출 가능
- 코드의 길이가 짧아지고 코드의 유지보수가 쉬움
- 함수의 인자(매개변수)도 타입형을 선언해 주지 않음
- 인자(매개변수)를 받을 때 타입이 결정되기 때문에 동적인 특징을 가짐

❖ 함 수

◆ 리턴값

- 리턴값 : 함수가 어떠한 기능을 수행하고 그 결과를 호출한 곳으로 돌려주는 값
- 함수가 어떠한 기능을 수행하고 돌려줄 값이 없으면 return 명령은 생략할 수 있다.

- return 의 의미
  - 실행중인 함수를 종료

◆ 함수 만드는 방법

- 함수를 만들때는 **def** 예약어를 사용합니다.
- **함수의 내용은 들여쓰기**를 합니다. 들여쓰기가 종료되면 함수 정의가 종료 됩니다.
- 함수의 이름은 변수명을 만들 듯이 사용자가 임의로 지정이 가능합니다.
- **필요에 따라 괄호안에 매개 변수를 지정**할 수 있습니다.
- [] (대괄호)는 생략가능하다는 표시 입니다.
- **매개 변수에 초기값을 지정**할 수 있습니다.
- **초기값을 지정하지 않으면 호출 시 반드시 인수를 지정**해야 합니다.

❖ 함 수

◆ 함수 만드는 방법

- 함수를 만들때는 **def** 예약어를 사용
- **함수의 내용은 들여쓰기**를 합니다. 들여쓰기가 종료되면 함수 정의가 종료
- 함수의 이름은 변수명을 만들 듯이 사용자가 임의로 지정이 가능
- **필요에 따라 괄호안에 매개 변수를 지정**할 수 있습니다.
- [] (대괄호)는 생략가능하다는 표시
- **매개 변수에 초기값을 지정**할 수 있습니다.
- **초기값을 지정하지 않으면 호출 시 반드시 인수를 지정**해야 합니다.

◆ 형식

함수 정의 시 사용하는 키워드: def

```
def 함수명([매개변수[=초기값], .....]):
    수행할 문장1
    수행할 문장2
    ....
    ....
    ....
    [return 돌려줄값]
```

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 반환값이 없는 함수

➡ ex 1]

```
# 매개변수(parameter)를 두개 지정
def line(count=80, char='-'):
    for i in range(0, count):
        print(char, sep='', end='')
    print()

print('재미있는 Python!!')
line(17)
# 인수(arguments)로 숫자 1개 지정. 첫번째 개변수(parameter)에 17이 대입

print('정말 재미있는 Python!!')
# 인수(arguments)로 숫자 1개와 문자 지정, 매개변수(parameter) 이름을 생략시 차례대로 대입
line(22, '~')

print('진짜 진짜 재미있는 Python!!')
line(char='^', count=27)  # 매개변수 이름을 지정하면 순서와 상관 없음

print('진짜 진짜 정말로 재미있는 Python!!')
line() # 인수(arguments) 생략 시 기본값 적용
```

```
재미있는 Python!!
----------
정말 재미있는 Python!!
~~~~~~~~~~~~~~~~~~~~~~
진짜 진짜 재미있는 Python!!
^^^^^^^^^^^^^^^^^^^^^^^^^^^
진짜 진짜 정말로 재미있는 Python!!
--------------------------------------------------------------------------------
```

❖ 함 수

◆ 반환값이 없는 함수

➡ 설 명 1]

- 함수를 정의할때 매개변수(parameter)를 두개 지정하였으며 기본 값도 지정하였습니다.
- 이렇게 함수를 지정하면 호출 할때 아주 다양하게 호출이 가능합니다.
- 함수를 호출할 때 인수를 생략하면 기본값이 적용됩니다. line()으로 호출하면 '-'를 80개 그립니다.
- 함수를 호출할 때 매개변수(parameter) 이름을 생략하면 인수값이 매개변수에 차례대로 전달됩니다.
- 함수를 호출할 때 매개변수(parameter) 이름을 지정하면 지정 이름에 값이 대입됩니다.

❖ 함 수

◆ 반환값이 있는 함수

➡ ex 1]

```
# n ~ m 사이의 합계 구하는 함수
def sum(n=1, m=100):
    if n > m:  # 앞의 값이 크면 교환합니다. (변수 스와핑)
        n, m = m, n
    s = 0
    for i in range(n, m+1):
        s += i
    return s

s = sum(1, 3) + sum(1, 5) + sum(1, 10)
print("합계 :", s)

print("{0}~{1}까지 합 : {2}".format(1, 100, sum()))
print("{0}~{1}까지 합 : {2}".format(1, 10, sum(1, 10)))
print("{0}~{1}까지 합 : {2}".format(1, 1000, sum(1, 1000)))

print("{0}~{1}까지 합 : {2}".format(10, 1, sum(10, 1)))
print("{0}~{1}까지 합 : {2}".format(10, 10, sum(10, 10)))
```

```
합계 : 76
1~100까지 합 : 5050
1~10까지 합 : 55
1~1000까지 합 : 500500
10~1까지 합 : 55
10~10까지 합 : 10
```

❖ 함 수

◆ 반환값이 있는 함수

- 리턴값 : 함수가 어떠한 기능을 수행하고 그 결과를 호출한 곳으로 돌려주는 값
- 함수가 어떠한 기능을 수행하고 돌려줄 값이 있으면 return 명령을 사용

➡ 설 명 1]

- 함수를 만들때는 def 예약어를 사용합니다.
- 함수의 내용은 들여쓰기를 합니다. 들여쓰기가 종료되면 함수 정의가 종료 됩니다.
- 함수의 이름은 변수명을 만들 듯이 사용자가 임의로 지정이 가능합니다.
- 필요에 따라 괄호안에 매개 변수를 지정할 수 있습니다.
- [] (대괄호)는 생략가능하다는 표시 입니다.
- 매개 변수에 초기값을 지정할 수 있습니다.
- 초기값을 지정하지 않으면 호출 시 반드시 인수를 지정해야 합니다.
- 값을 되돌려 줄때 return 명령어를 사용합니다.
- return 명령어를 만나면 함수가 종료됩니다.
- return 명령어는 하나의 함수에 여러번 나올 수 있습니다. 하지만 1번만 사용하는것이 좋습니다.

❖ 함 수

◆ 반환값이 있는 함수

- 두 개 이상의 값을 동시에 반환할 수 있다.

➡ ex 1]

```
def calc(x, y):
    return x + y, x - y, x * y, x / y

print(calc(10, 2))
```

(12, 8, 20, 5)

➡ 설 명 1]

- 인자는 2개인데, 총 4개(더하기, 빼기, 곱하기, 나누기한 값)을 리턴
- 여러 개의 값이 콤마의 형태로 묶여있는 자료: tuple(튜플)

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 함수 객체와 함수 호출

- 함수의 이름 자체는 함수 객체의 레퍼런스(Reference)를 지니고 있다.

➡ ex 1]

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

```
<function add at 0x108ceeb90>
```

- 파이썬에서 함수는 객체 → add에는 함수객체의 레퍼런스가 있음

➡ ex 2]

```
c = add(10, 30)
print(c)
```

```
40
```

❖ 함 수

◆ 함수 객체와 함수 호출

- 함수 이름에 저장된 레퍼런스를 다른 변수에 할당하여 그 변수를 이용한 함수 호출 가능

➡ ex 3]

```
f = add
print(f(4, 5))
```

```
9
```

➡ 설 명 3]

- f = add → add의 참조값이 f에 복사

❖ 함 수

◆ 함수 인수값 전달방법

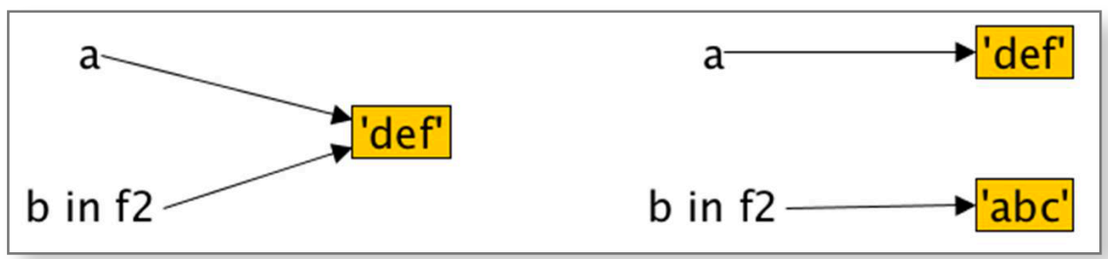
- 함수 인자에 변경불가능(Immutable) 객체인 문자열을 전달
  - 함수 내에서 다른 문자열로 치환 --> 의미 없는 인자 전달

➡ ex 1]

```
def f2(b):
    b = "abc"
a = "def"
f2(a)
print(a)
```

```
def
```

➡ 설 명 1]



- a와 b 모두 문자열을 가리키며 레퍼런스 값 가짐
- 문자열은 변경할 수 없는 자료형
- 튜플도 같은 맥락으로 확인 가능

❖ 함 수

◆ 함수 인수값 전달방법

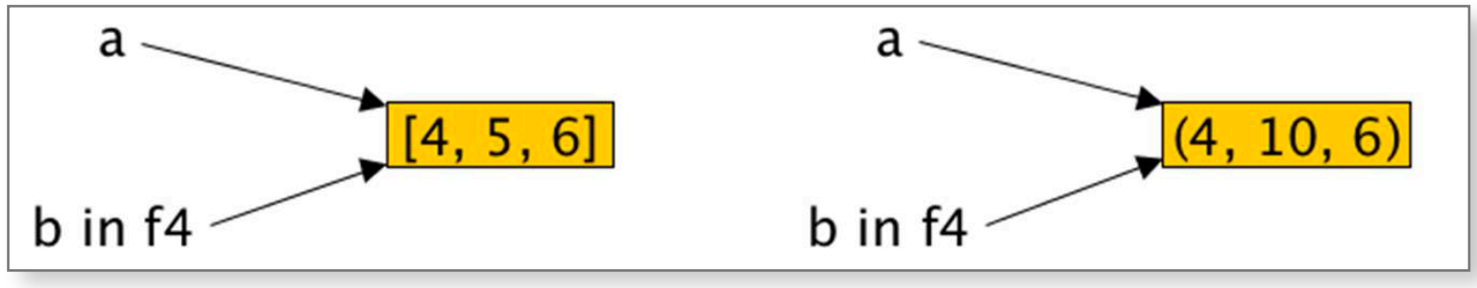
- 함수 인자에 변경가능한(Mutable)한 객체인 리스트 전달 및 내용 수정
  - 전형적인 함수 인자 전달법 및 활용법

➡ ex 2]

```
def f4(b):
    b[1] = 10
a = [4,5,6]
f4(a)
print(a)
```

[4, 10, 6]

➡ 설 명 2]



- 리스트는 변경 가능한 자료형
- 함수에 인자 전달 시 리스트로 전달하는 경우 많음

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 함수 인수값 전달방법

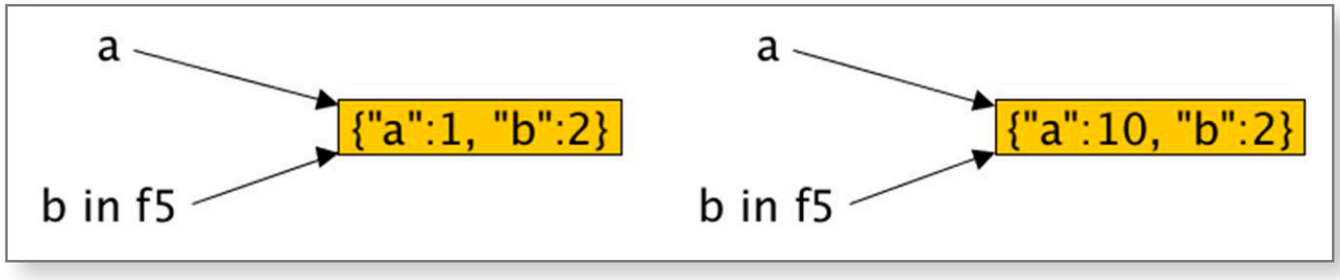
- 함수 인자에 변경가능한(Mutable)한 객체인 사전 전달 및 내용 수정
  - 전형적인 함수 인자 전달법 및 활용법

➡ ex 3]

```
def f5(b):
    b['a'] = 10
a = {"a":1, "b":2}
f5(a)
print(a)
```

```
{'a': 10, 'b': 2}
```

➡ 설 명 3]



훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 함수의 정의와 호출

- 파이썬에서는 모든 객체는 동적으로 (실행시간에) 그 타입이 결정된다.
- 함수 인자는 함수가 호출되는 순간 해당 인자에 전달되는 객체에 따라 그 타입이 결정된다.
  - 함수 몸체 내에서 사용되는 여러가지 연산자들은

➡ ex 1]

```
def add(a, b):
    return a + b
c = add(1, 3.4)
d = add('dynamic', 'typing')
e = add(['list'], ['and', 'list'])
print(c)
print(d)
print(e)
```

```
4.4
dynamictyping
['list', 'and', 'list']
```

➡ 설 명 1]

- 파이썬은 변수나 인자 상황에 type을 적지 않음
- 값이 실제 정의된 함수에 할당이 될 때 type 결정됨

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 기본 인수 값

- **기본 인수 값**
  - 함수를 호출할 때 인수를 넘겨주지 않아도 인수가 기본적으로 가지는 값

➡ ex 1]

```
def incr(a, step=1):
    return a + step
b = 1
b = incr(b) # 1 증가 # incr(a=b) 로 실행시켜도 된다.
print(b)
b = incr(b, 10) # 10 증가 (stop = 10) or b = incr(step=10, a=b) 로 호출해도 된다.
print(b)
```

```
2
12
```

➡ 설 명 1]

- step에는 기본적으로 인수값 1이 할당됨
- incr(b)에서 b는 a에 들어감
- incr(b, 10) → 10은 step에 들어감
- 기본인자 없으면 error 발생 → 정의한 함수의 인자가 2개이기 때문

❖ 함 수

◆ 기본 인수 값

- [주의] 함수 정의를 할 때 기본 값을 지닌 인수 뒤에 일반적인 인수가 올 수 없음

➡ ex 1]

```
def incr(step=1, a):
    return a + step
```

```
def incr(step=1, a):
             ^
SyntaxError: non-default argument follows default argument
```

➡ 설 명 1]

- step = 1 이라고 정의한 것은 앞 인자에 쓰지 못함
- 기본인자는 맨 마지막 뒤에 와야 함
- 인자가 여러 개일 경우 기본인자 값이 없는 것이 맨 앞 위치

❖ 함 수

◆ 기본 인수 값

- 함수 정의 시에 여러 개의 기본 인수 값 정의 가능

➡ ex 2]

```
def incr(a, step=1, step2=10):
    return a + step + step2
print(incr(10))
```

```
21
```

➡ 설 명 2]

- 일반인자가 앞으로 위치해야 함

★[주의] 함수 호출시에 키워드 인수 뒤에 일반 인수 값이 올 수 없다.
(키워드 인수가중간에 와도 에러)

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 가변 인수 리스트

- 함수 정의시에 일반적인 인수 선언 뒤에 *var 형식의 인수로 가변 인수를 선언할 수 있음
  - var에는 함수 호출시 넣어주는 인수 값들 중 일반 인수에 할당되는 값을 제외한 나머지 값들을 지닌 *** 튜플 객체 ***가 할당된다.

➡ ex 1]

```
def varg(a, *arg):
    print(a, arg)
varg(1)
varg(2,3)
varg(2,3,4,5,6)
```

```
1 ()
2 (3,)
2 (3, 4, 5, 6)
```

➡ 설 명 1]

- *arg → 가변 인수를 받겠다는 의미
- *arg → 튜플 형태로 반환함
- 콤마(,)가 있어야지 원소가 하나인 튜플을 반영
- arg[0] → 튜플의 첫 번째 원소

❖ 함 수

◆ 가변 인수 리스트

- **Java 언어의 printf문과 유사한 형태의 printf 함수 정의 방법**

➡ ex 2]

```
def printf(format, *args):
    print(format % args)

printf("I've spent %d days and %d night to do this", 6, 5)
```

I've spent 6 days and 5 night to do this

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 튜플 인수로 함수 호출하기

- 함수 호출에 사용될 인수값들이 튜플에 있다면 "*튜플변수"를 이용하여 함수 호출이 가능

➡ ex 1]

```
def h(a, b, c):
    print(a,b,c)

args = (1, 2, 3)
h(*args)
```

```
1 2 3
```

➡ 설 명 1]

- 함수 정의 시 * 사용하게 되면 가변인수
- 함수 호출 시 * 사용하고 뒤에 튜플을 넣으면 튜플 전체 호출 가능

❖ 함 수

◆ 사전 인수로 함수 호출하기

- 함수 호출에 사용될 인수값들이 사전에 있다면 "**사전변수"를 이용하여 함수 호출이 가능

➡ ex 1]

```
def h(a, b, c):
    print(a,b,c)

dargs = {'a':1, 'b':2, 'c':3}
h(**dargs)
```

```
1 2 3
```

➡ 설 명 1]

- 함수 호출 시 ** 사용하고 뒤에 사전을 넣으면 사전 전체 호출 가능
- 인자의 이름과 식별자가 동일한 키 값을 가져야 함

❖ 함 수

◆ 재귀 함수

- 재귀함수 : 어떤 함수에서 자신을 다시 호출하여 작업을 수행하는 방식의 함수를 의미
  - 다른 말로는 재귀호출, 되부름
  - 반복문을 사용하는 코드는 항상 재귀함수를 통해 구현하는 것이 가능하며 그 반대도 가능

- 재귀함수를 작성할 때는
  - 함수내에서 다시 자신을 호출한 후 그 함수가 끝날 때 까지 함수 호출 이후의 명령문이 수행되지 않는다
  - 종료조건이 꼭 포함 되어야한다는 부분을 인지하고 작성(무한루프를 방지)

❖ 함 수

◆ 재귀 함수

➡ ex 1]

```
# n ~ m 사이의 합계 구하는 함수
def total(n = 1, m = 100):
    # m이 n 보다 클경우 위치 변환
    if n > m:
        n, m = m, n

    # 함수의 종료 조건
    if m == n:
        return n

    # 함수의 반환값 - 이 함수를 다시 호출
    return m + total(n, m - 1)

print('total(10, 1) :', total(10, 1))
```

```
total(10, 1) : 55
```

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 이름 공간

- 이름 공간 또는 스코프 (Naming Space or Scope): 이름이 존재하는 장소
  - 파이썬은 실행 시간에 각 이름들을 적절한 이름 공간에 넣어 관리한다.
- 이름 공간(스코프)의 종류
  - 지역(Local): 각 함수 내부
  - 전역(Global): 모듈 (파일) 내부
  - 내장(Built-in): 파이썬 언어 자체에서 정의한 영역
- 변수가 정의되는 위치에 의해 변수의 스코프가 정해짐
  - 파이썬에서 변수의 정의
    - ○변수가 l-value로 사용될 때
- 변수가 r-value로 사용될 때 해당 변수의 값을 찾는 순서 규칙
  - L --> G --> B

➡ l-value와 r-value의 정의

- l-value : 반드시 명시적인 메모리 공간을 가져야 한다. 값이나 개체를 대입 받기 때문
- r-value : 잠깐 사용하고 사라지는 임시적인 값. 잠시만 유지해서 l-value에 값을 전해줄 수만 있으면 된다.
  - 임시 값'에 대한 통칭

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 지역변수 전역변수

- 변수의 스코프는 해당 변수가 l-value로서 정의되는 위치에 따라 달라짐
- 변수가 함수 내에서 정의되면 해당 함수의 지역 변수가 된다.

➡ ex 1]

```
# g, h는 전역 변수
g = 10
h = 5
def f(a): # a는 지역 변수
    h = a + 10 # h는 지역, 새로 l-value로 정의했음
    b = h + a + g
    # b도 지역, g는 r-value이므로 기존 값을 참조 - 전역 변수
    return b
print(f(h))
# 함수 호출시에 사용되는 변수는 해당 위치의 스코프에서 값을 찾음
# - 전역 변수
print(h) # 전역 변수 h는 변함 없음
```

```
30
5
```

➡ 설 명 1]

- 함수 호출 시 ** 사용하고 뒤에 사전을 넣으면 사전 전체 호출 가능
- 인자의 이름과 식별자가 동일한 키 값을 가져야 함

❖ 함 수

◆ 지역변수 전역변수

- 함수 내부에서 전역 변수를 직접 사용하고자 할 때
  - global 키워드 활용

➡ ex 1]

```
h = 5
def f(a): # a는 지역
    global h
    # h 변수를 전역이라고 미리 선언함

    h = a + 10
    # h는 l-velue로 정의되더라도 미리 선언된 내용 때문에 전역 변수

    return h
print(f(10))
print(h) # 전역 변수 h 값이 함수 내에서 변경되었음
```

```
20
20
```

➡ 설 명 1]

- f 함수 내 h는 지역변수
- global h 키워드를 적으면 h가 전역 변수 사용으로 선언됨
- h = 5 가 사라지고 새로운 값 20이 들어감

❖ 함 수

◆ 지역변수 전역변수

- [주의] 동일 함수 내에서 동일한 변수가 지역변수와 전역변수로 동시에 활용될 수 없음
  - 함수 내에서 정의되는 변수는 지역 변수로 간주
  - 지역 변수로 선언되기 이전에 해당 변수를 사용할 수 없음

➡ ex 1]

```
g = 10
def f():
    a = g # l-value로 사용되는 g는 전역 변수
    g = 20 # r-value로 정의되는 g는 지역 변수
    return a
print(f())
```

```
---------------------------------------------------------------------
                        생  략
---------------------------------------------------------------------
    a = g # l-value로 사용되는 g는 전역 변수
```

➡ 설 명 1]

- a라는 변수는 함수 내에서 새로 만들어지는 것
- g는 local 변수 내에 없으므로 전역변수 값 사용
- global g 와 같은 내용이 없기 때문에 l-value의 g는 새롭게 선언
- 그러면 g는 함수 안에 존재하는 지역변수가 됨
- g가 지역변수이면서 전역변수이기 때문에 error 발생

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 함 수

◆ 지역변수 전역변수

➡ ex 2]

```
g = 10
def f():
    global g # g는 전역 변수로 선언됨
    a = g # a는 지역 변수, g는 전역 변수
    g = 20 # g는 전역 변수
    return a
print(f())
```

```
10
```

➡ 설 명 2]

- a=g 가 g =20 아래에 쓰이면 결과는 20

❖ 함 수

◆ 함수의 중첩 영역(Nested Scopes) 지원

- Nested Scope: 함수 안에 정의되어 있는 함수 내부
  - 가장 안쪽의 스코프부터 바깥쪽의 스코프쪽으로 변수를 찾는다.

➡ ex 1]

```
x = 2

def F():
    x = 1
    def G():
        print(x)
    G()

F()
```

```
1
```

➡ 설 명 1]

- 아래 x는 위 x와 다른 F 함수의 지역변수
- G() → 중첩영역을 지원하는 함수
- 함수 안에 print x를 하면 가까운 x를 먼저 찾음

모듈과 패키지

❖ 모 듈

◆ 정의

- 모듈: 파이썬 프로그램 파일로서 파이썬 데이터와 함수등을 정의하고 있는 단위
  - 서로 연관된 작업을 하는 코드들을 묶어서 독립성을 유지하되 재사용 가능하게 만드는 단위
  - 모듈을 사용하는 측에서는 모듈에 정의된 함수나 변수 이름을 사용

◆ 종류

- 제공자에 따른 분류

- 표준 모듈 :
  - 파이썬 언어 패키지 안에 기본적으로 포함된 모듈(파이썬 설치 시 포함 되어있는 모듈)
  - 대표적인 표준 모듈 예 : math, string
- 사용자 생성 모듈 :
  - 프로그래머가 직접 정의(작성)한 모듈
- 써드 파티 (3rd Party) 모듈 : 파이썬 제단도 프로그래머도 아닌 다른 프로그래머, 또는 업체에서 제공하는 모듈

- 모듈이 정의되고 저장되는 곳은 파일
  - 파일 : 모듈 코드를 저장하는 물리적인 단위
  - 모듈 : 논리적으로 하나의 단위로 조직된 코드의 모임
- 파이썬 모듈이 정의되는 파일의 확장자: .py
- 다른 곳에서 모듈을 사용하게 되면 해당 모듈의 .py는 바이트 코드로 컴파일 되어 .pyc로 존재한다.
  - pyc가 만들어지면 py가 없더라도 pyc를 활용하여 import가능

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 모 듈

◆ 모듈을 왜 사용하는가?

- 함수와 모듈
  - 함수: 파일 내에서 일부 코드를 묶는 것
  - 모듈: 파일 단위로 코드들을 묶는 것○비슷하거나 관련된 일을 하는 함수나 상수값들을 모아서 하나의 파일에 저장하고 추후에 재사용하는 단위
- 모듈 사용의 이점
  - 코드의 재사용
  - 프로그램 개발시에 전체 코드들을 여러 모듈 단위로 분리하여 설계함으로써 작업의 효율을 높일 수 있음
  - 별도의 이름 공간(스코프)를 제공함으로써 동일한 이름의 여러 함수나 변수들이 각 모듈마다 독립적으로 정의될 수 있다.
- 모듈은 하나의 독립된 이름 공간을 확보하면서 정의된다

- 함수와 모듈의 공통점 : 관련된 코드를 한 곳으로 묶는 것
- 동일한 레벨에서 여러 개의 함수가 한 모듈 내 존재 가능
- 모듈 : 파일 하나가 모듈
- 여러 개의 모듈을 기능, 역할 단위로 나누어 관련 내용을 채워 넣는 것
- 분업도 가능
- a 모듈에 aaa --> b 모듈에 aaa
- #으로 시작됨 --> 주석
- 주석 안 내용은 코딩이라는 키 값의 값으로 utf-8 을 적어줌
- 코딩이 utf-8 방식으로 저장됨
- 파이썬과 개발자 간의 약속
- 주석에 마음대로 한글 작성 가능

❖ 모 듈

◆ 사용자 모듈 만들기와 호출하기

➡ ex 1]

```
#File: mymath.py
mypi = 3.14

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

def area(r):
    return mypi * r * r
```

➡ 설 명 1]

- mypi 변수, add 함수, area 함수 존재
- mymath --> 모듈객체

❖ 모 듈

◆ 사용자 모듈 만들기와 호출하기

- 모듈 이름은 해당 모듈을 정의한 파일 이름에서 .py를 제외한 것
  - 모듈을 불러오는 키워드: import
- 모듈에서 정의한 이름 사용하기

➡ ex 2]

```
import mymath
print(dir(mymath)) # mymath에 정의된 이름들 확인하기
print(mymath.mypi) # mymath 안에 정의된 mypi를 사용한다
print(mymath.area(5)) # mymath 안에 정의된 area를 사용한다
```

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'add', 'area', 'mypi']
3.14
78.5
```

➡ 설 명 2]

- dir(mymath) --> mymath안에 들어있는 모든 이름을 리스트로 출력
- '__'가 없는 식별자는 평범한 식별자
- 모듈명 + 모듈 안 존재하는 member

❖ 모 듈

◆ 모듈이 지닌 이름들 알아보기

- dir(모듈): 모듈이 지니고 있는 모든 이름들을 리스트로 반환

➡ ex 1]

```
import string
print(dir(string))
```

```
['Formatter', 'Template', '_ChainMap', '_TemplateMetaclass', '__all__', '__builtins__',
'__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', '_re', '_string', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase',
'capwords', 'digits', 'hexdigits', 'octdigits', 'printable', 'punctuation', 'whitespace']
```

➡ 설 명 1]

- string --> 파이썬에서 같이 설치되는 표준 모듈

❖ 모 듈

◆ 이름 공간을 제공하는 다른 예들

- 독립된 이름 공간(스코프)을 제공하는 것들
  - 모듈
  - 클래스
  - 객체
  - 함수

➡ ex 1]

string 모듈 이름 공간에 변수 a를 생성한다.
- 표준 모듈에 사용자가 정의하는 이름을 생성하는 것은 비추천
- 단치 모듈 자체가 독립적인 이름 공간을 제공한다는 것을 알려줌

```
import string
string.a = 1
print(string.a)
```

```
1
```

➡ 설 명 1]

- string 모듈 바깥에서 새로운 변수를 정의하여 삽입 가능
- 표준모듈에 새로운 변수를 정의하여 삽입하는 것은 추천 X

❖ 모 듈

◆ import

• 다른 모듈 내의 코드에 대한 접근을 가능하게 하는 명령
  - 변수, 함수, 클래스 등이 모두 포함

* 형식 1

```
import 모듈   # 모듈의 실제 파일 명은 "모듈.py"
```

➡ ex 1]

```
import math
print(3**2*math.pi)
```

```
28.274333882308138
```

* 형식 2

```
from 모듈 import 변수 또는 함수
```

➡ ex 2]

```
from math import pi
print(3**2*pi)
```

```
28.274333882308138
```

❖ 모 듈

◆ import

* 형식 3

```
from 모듈명 import *
```

- 해당 모듈에 존재하는 '__'로 시작되는 이름들을 제외한 모든 이름들을 현재 이름 공간으로 불러들인다.

➡ ex 3]

```
from math import *
print(3**2*pi)
```

```
28.274333882308138
```

* 형식 4

```
import 모듈명 as 새로운 모듈 이름
```

- 해당 모듈을 새로운 다른 이름으로 사용하고자 할 때 사용
- 기존 모듈 이름이 너무 길거나 현재 사용중인 다름 이름들과 충돌이 일어날 때 유용

➡ ex 4]

```
import math as mt
print(3**2*mt.pi)
```

```
28.274333882308138
```

❖ 모 듈

◆ import

* 형식 5

```
from 모듈명 import 이름 as 새로운 이름[, 이름 as 새로운 이름]
```

- 해당 모듈 내에 정의된 이름을 다른 새로운 이름으로 사용하고자 할 때 사용

➜ ex 5]

```
from math import pi as p7
print(3**2*p7)
```

```
28.274333882308138
```

❖ 모 듈

◆ import 명령의 위치

import 문은 보통의 문(statement)이 작성될 수 있는 곳이면 어디에서나 사용 가능
- 예를 들면 함수 정의 def 문 안이나 if 문 안에서 사용할 수 있음

➡ ex 1]

```
def str_test(s):
    import string
    t = string.split(s)
    return t
```

➡ 설 명 1]

- 함수 정의 내에서 import 바로 사용

❖ 모 듈

◆ import에 의한 모듈 코드 수행

- import는 코드를 가져오기만 하는 것이 아니라 가져온 코드를 수행한다.

➡ ex 1]

```
#FILE : mymath.py
mypi = 3.14
def add(a, b):
    return a + b

def area(r):
    return mypi * r * r

print(area(4.0))
```

```
import mymath
```

50.24

➡ 설 명 1]

- import mymath → mymath 수행값이 있으면 가져오기도 함

❖ 모 듈

◆ 컴파일과 적재시간

- import mymath를 수행할 때 발생하는 일
  1) 우선 mymath.pyc를 찾는다.
  2) mymath.pyc가 없다면 mymath.py를 찾아서 mymath.pyc를 생성한다.
  3) 생성된 mymath.pyc를 메모리로 읽어들여 수행한다.
- .pyc 파일
  - 바이트 코드 파일
    - ○ 기계나 플랫폼(OS)에 의존하지 않도록 만들어진 일종의 목적 코드 (Object Code)
    - ○ 파이썬은 컴파일 언어이면서 동시에 인터프리터 언어의 수행 방식을 취하고 있다.
- 새로운 .pyc 생성에 대한 판단
  - .py 수정 시간이 .pyc 수정 시간보다 더 최근일 때
- .py가 없이도 .pyc 파일만 있어도 import 가능
  - 코드를 숨기는 간단한 기법으로 활용 가능

- .pyc → 원래 mymath.py가 import 되는 순간 바이트 코드 같이 생성
- .pyc → 바이너리 파일도 아니고, 텍스트 파일도 아닌 중간 역할
- 바이트 코드는 문자들로 이루어짐
- pyc 파일은 처음 생성 후 다시 생성 가능
- py 없이 pyc만 존재해도 모듈로서 역할 가능
- pyc의 내용은 일반적으로 내용 확인 불가

❖ 패 키 지

◆ 정의

- 패키지(Package)
  - 여러모듈들을한데묶어서정리해놓은구조
  - 물리적으로여러모듈파일을모아놓은디렉토리에해당
    - ○ 최상위디렉토리이름이패키지이름이된다.
    - ○ 최상위디렉토리하위에여러서브디렉토리는해당최상위패키지의 하위패키지가된다.

❖ 예외처리

◆ 예외처리의 목적

- 프로그램의 정상적인 종료

◆ 예외처리 형식

```
try:
    (예외 발생 가능한) 일반적인 수행문들
[ except 예외클래스1:
    예외가 발생하였을 때 수행되는 문들
except 예외클래스2:
    예외가 발생하였을 때 수행되는 문들
    . . .
[ else:
    예외가 발생하지 않았을 때 수행되는 문들] ]
[finally:
    예외 발생 유무와 관계없이 무조건 수행되는 문들]
```

❖ 예외처리

◆ 예외처리 방법

• try: 구문을 수행하면 프로그램이 오류가 발생해도 비정상적인 종료가 되지 않고 정상적으로 수행

➡ ex 1]

```
try:
    print 1.0 / 0.0
except ZeroDivisionError:
    print('zero division error!!!')
```

```
zero division error!!!
```

➡ 설 명 1]

- try는 새로운 구문이 시작되어야 하기 때문에 콜론(:) 삽입
- 0으로 나누고 있기 때문에 ZeroDivisionError 발생
- try, except 절 사용 X → 빨간 Error 발생 (프로그램 비정상적 수행)
- try 절을 수행하면 프로그램이 정상적으로 수행되어 종료 X

❖ 예외처리

◆ 예외처리 방법

• 상황에 따라서는 에러와 함께 따라오는 정보를 함께 받을 수도 있다.

➡ ex 2]

```
try:
    spam()
except NameError, msg:
    print('Error -', msg)
```

Error - name 'spam' is not defined

➡ 설 명 2]

- msg → NameError의 메시지가 구현됨
- 콤마(,) 대신 as 사용하여 메시지 받을 수 있음

❖ 예외처리

◆ 예외처리 방법

- else: 구문은 except: 구문없이 사용 못한다.

➡ ex 3]

```
def division():
    for n in range(0, 5):
        try:
            print 10.0 / n
        else:
            print "Success"

division()
```

➡ 설 명 3]

- else는 except 없이는 사용 X
- else가 있는데 except이 없는건 구문적 error → 빨간 X, 밑줄 제시
- finally는 관계없이 적을 수 있음

❖ 예외처리

◆ 예외처리 방법

• except 뒤에 아무런 예외도 기술하지 않으면 모든 예외에 대해 처리된다.

➡ ex 4]

```
try:
    spam()
    print 1.0 / 0.0
except:
    print('Error')
```

Error

➡ 설 명 4]

- NameError, NameEroor 둘 다 발생이 되는 상황
- except 뒤에 아무런 내용 X → 모든 예외 catch
- 어떤 예외가 발생했는지는 알 수 없음

❖ 예외처리

◆ 예외처리 방법

• 여러 예외들 각각에 대해 except 절을 다중으로 삽입할 수 있다.

➡ ex 5]

```
b = 0.0
name = 'aaa.txt'
try:
    print(1.0 / b)
    spam()
    f = open(name, 'r')
    '2' + 2
except NameError:
    print('NameError !!!')
except ZeroDivisionError:
    print('ZeroDivisionError !!!')
except (TypeError, IOError):
    print('TypeError or IOError !!!')
else:
    print('No Exception !!!')
finally:
    print('Exit !!!')
```

```
ZeroDivisionError !!!
Exit !!!
```

➡ 설 명 5]

- ZeroDivisionError, NameError, IOError, TypeError 발생 가능
- 튜플 형태로 a or b로 묶어서 정의 가능
- 튜플은 가로가 없어도 사용 가능
- 각각의 경우에 따라서 제어를 해주고 싶으면 except 여러 번 사용 가능
- 예외가 발생하지 않아도 finally 이 부분은 항상 수행

❖ 예외처리

◆ 주요 예외 클래스의 범주와 의미 (발생 원인)

BaseException	모든 예외의 최상위 예외
├─ SystemExit	프로그램을 종료하는 명령이 실행되었을 때
├─ KeyboardInterrupt	Control-C 키가 입력되었을 때
└─ Exception	대부분의 예외의 상위 예외
├─ ArithmeticError	수의 연산과 관련된 문제
│ └─ ZeroDivisionError	수를 0으로 나누려 할 때
├─ AssertionError	assert 문에 의해 발생
├─ AttributeError	(모듈·클래스·인스턴스에서) 잘못된 속성을 가리킬 때
├─ EOFError	(파일에서) 읽어들일 데이터가 더이상 없을 때
├─ ImportError	모듈을 임포트할 수 없을 때
│ └─ ModuleNotFoundError	임포트할 모듈을 찾을 수 없을 때
├─ LookupError	잘못된 인덱스·키로 인덱싱할 때
│ ├─ IndexError	(시퀀스에서) 잘못된 인덱스로 인덱싱할 때
│ └─ KeyError	(매핑에서) 잘못된 키로 인덱싱할 때
├─ NameError	잘못된 이름(변수)을 가리킬 때
├─ OSError	운영 체제의 동작과 관련된 다양한 문제
│ ├─ ChildProcessError	하위 프로세스(프로그램이 실행한 외부 프로그램)에서 오류 발생
│ ├─ FileExistsError	이미 존재하는 파일·디렉터리를 새로 생성하려 할 때
│ ├─ FileNotFoundError	존재하지 않는 파일·디렉터리에 접근하려 할 때
│ ├─ IsADirectoryError	파일을 위한 명령을 디렉터리에 실행할 때
│ ├─ NotADirectoryError	디렉터리를 위한 명령을 파일에 실행할 때
│ ├─ PermissionError	명령을 실행할 권한이 없을 때
│ └─ TimeoutError	명령의 수행 시간이 기준을 초과했을 때
├─ RuntimeError	다른 분류에 속하지 않는 실행시간 오류
│ ├─ NotImplementedError	내용 없는 메서드가 호출되었을 때
│ └─ RecursionError	함수의 재귀 호출 단계가 허용한 한계를 초과했을 때
├─ SyntaxError	구문 오류
│ └─ IndentationError	들여쓰기가 잘못되었을 때
│ └─ TabError	들여쓰기에 탭과 스페이스를 번갈아가며 사용했을 때
├─ TypeError	연산·함수가 계산할 데이터의 유형이 잘못되었을 때
├─ ValueError	연산·함수가 계산할 데이터의 값이 잘못되었을 때
│ └─ UnicodeError	유니코드와 관련된 오류
└─ Warning	심각한 오류는 아니나 주의가 필요한 사항에 관한 경고

- 모든 예외 클래스는 `BaseException` 클래스의 하위 클래스이며,
  대부분의 예외 클래스는 `Exception` 클래스의 하위 클래스

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 예외처리

◆ 예외 강제 발생

* 형식

```
raise 예외클래스
```

➡ ex 1]

```
raise Exception("예외를 일으킵니다.")
```

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise Exception("예외를 일으킵니다.")
```

➡ ex 2]

```
text = input()
if text.isdigit() == False:
    rasie Exception(“입력받은 문자열이 숫자로 구성되어 있지 않습니다.”):
```

❖ 예외처리

◆ 사용자 정의 예외

- 파이썬이 제공하는 내장 예외 형식만으로 충분하지 않을 때 직접 예외 클래스를 정의할 수 있음.
- 사용자 정의 예외 클래스는 Exception 클래스를 상속하여 정의함.

* 형식

```
class 만들예외이름(Exception):
    처리내용
```

➡ ex 1]

```
class InvalidIntException(Exception):
    def __init__(self, arg):
        super().__init__('정수가 아닙니다.: {0}'.format(arg))

def convert_to_integer(text):
    if text.isdigit(): # 부호(+, -) 처리 못함.
        return int(text)
    else:
        raise InvalidIntException(text)

if __name__ == '__main__':
    try:
        print('숫자를 입력하세요:')
        text = input()
        number = convert_to_integer(text)
    except InvalidIntException as err:
        print('예외가 발생했습니다 ({0})'.format(err))
    else:
        print('정수 형식으로 변환되었습니다 : {0}({1}'.format(number, type(number)))
```

```
InvalidIntException.py
숫자를 입력하세요:
123
정수 형식으로 변환되었습니다 : 123(<class 'int'>

>InvalidIntException.py
숫자를 입력하세요:
abc
예외가 발생했습니다 (정수가 아닙니다.: abc)
```

훈련교사 : 전 은 석
email : euns_jun@naver.com

❖ 예외처리

◆ 참고] 가정 설정문 - assert

• assert는 뒤의 조건이 True가 아니면 AssertError를 발생한다.

* 형식

assert 조건 [, 조건이 거짓일때 실행문]

➡ ex 1]

```
lists = [1, 3, 6, 3, 8, 7, 13, 23, 13, 2, 3.14, 2, 3, 7]
def test(t):
    assert type(t) is int, '정수 아닌 값이 있네'

for i in lists:
    test(i)
```

AssertionError: 정수 아닌 값이 있네