

파이썬 프로그래밍



Index



- 1) 파이썬 개요
- 2) 변수와 데이터타입
- 3) 자료구조
- 4) 제어문
- 5) 함수
- 6) 파이썬 객체지향 프로그래밍
- 7) 모듈과 패키지
- 8) 예외처리
- 9) 입출력
- 10) 내장함수
- 11) 이터레이터와 데코레이터
- 12) 정규 표현식



파이썬 개요



파이썬

- 1991년 네덜란드 프로그래머 귀도 판 로섬(Guido van Rossum) 이 만든 프로그래밍 언어.
- 1989년 크리스마스 연휴 기간에 취미로 가질 만한 프로그래밍 언어로 만들기 시작
- 평이한 영어로 이해할 수 있는 코드와, 일상적인 업무에 적용할 수 있고 짧은 개발 시간을 목표로 개발
- 파이썬 2와 파이썬 3
 - 2008년 파이썬 3버전이 발표됨.
 - 그 이전의 파이썬 2.XX 버전과 하위 호환 없이 새로운 버전으로 발표됨.
 - 파이썬 2와 파이썬 3은 각각 버전업이 되고 있다.
 - 파이썬 2.XX 은 2020년까지 지원할 예정임.



파이썬

- 특징

- 인터프리터 방식
- 구문의 가독성이 좋다.
 - '코드는 작성하는 것 보다 읽는 경우가 많다' 는 명제 아래 코딩 스타일의 일관성을 유지하여 가독성을 높이는 것을 중시한다.
- 유지보수성이 좋다.
 - 파이썬은 배우기 쉽고, 코드를 읽기 쉽기 때문에 유지보수성이 좋다.
- 스크립트 언어
 - 동적 타이핑언어
 - 인터프리터 언어
 - 크로스 플랫폼
- 다른 언어와의 유연한 확장 구조
- 강력한 표준라이브러리 제공
- 잘 갖춰진 생태계
 - 풍부한 확장 패키지들을 다양한 집단에서 제공(서드파티 라이브러리) 하여 개발 생산성이 좋다.

파이썬

- 파이썬으로 할 수 있는 것

- 업무 자동화
 - 스크립트
 - 업무자동화
- 범용 어플리케이션 프로그램 개발
 - GUI 기반의 독립형 어플리케이션 개발
- 데이터 과학과 머신러닝
 - 최근에 파이썬이 각광 받는 이유
 - numpy, pandas, scikit-learn 라이브러리
- 웹서비스 및 Restful API
 - Django나 Flask 와 같은 프레임워크를 이용하면 웹 어플리케이션을 빠르고 쉽게 개발 할 수 있다.

- 할 수 없는 것

- 시스템 프로그래밍에는 적합하지 않다. (C/C++등을 이용)
- 모바일 프로그래밍

파이썬 코딩 관례 (Coding Convention)

- PEP8 – 파이썬 코딩 스타일 가이드 참고(<https://www.python.org/dev/peps/pep-0008/>)
 - PEP: Python Enhancement Proposal – 파이썬을 개선하기 위한 제안서
- 코드 블록은 들여 쓰기를 이용한다. (규칙)
 - 선언문 다음에 공백 4개 또는 Tab만큼 들여 쓴 뒤 작성한다.
 - 들여쓰기는 탭 대신 공백 4칸을 권장
- 구문 하나당 (실행문당) 한 줄
 - 세미콜론(;)을 이용해 한 줄에 여러 구문을 작성할 수 있지만 권장되지 않는다.
 - 하나의 실행문을 여러 줄로 나눠 작성할 경우 \ (backslash)로 연결한다.
- 명명 관례
 - 함수, 변수는 소문자로 주고 단어는 Underscore(_)로 연결한다.
 - age_sum, read_csv()
 - 클래스는 단어 첫 글자를 대문자로 하는 파스칼 표기법을 사용한다.
 - Member
 - 모듈이름은 한단어의 소문자로 하되 단어가 합쳐 져야 할 때는 는 Underscore(_)로 연결할 수 있다.

파이썬 코딩 관례 (Coding Convention)

- 한 줄은 최대 79자까지 작성을 권장.
- 최상위 함수와 클래스 정의는 2줄을 뚫는다.
- 클래스내 메소드는 1줄을 띄어 쓴다.
- 불필요한 공백을 주지 않는다.
 - []와 () 안에 공백을 주지 않는다.
 - (,) , (:), (;) 앞의 공백을 주지 않는다.
- 키워드 인자와 기본값이 있는 매개변수의 경우 = 은 붙여서 쓴다.
- 코드가 변경되면 주석의 내용도 갱신하고 불필요한 주석은 달지 않는다.



PyPI (Python Package Index)

- 파이썬 서드파티 패키지들의 저장소
 - 파이썬의 패키지 대부분이 관리되고 있는 패키지 저장소
 - <https://pypi.org/>

- 주요 명령어

| 명령어 | 설명 |
|-------------------------------------|-----------------------|
| pip install 패키지명 | 패키지 install |
| pip install --upgrade 패키지명 | 패키지 업그레이드 |
| pip uninstall 패키지명 | 패키지 제거 |
| pip list | 설치된 패키지 조회 |
| pip show 패키지명 | 특정 패키지의 정보 조회 |
| python -m pip install --upgrade pip | pip 툴 업그레이드 (windows) |
| pip install --upgrade pip | pip 툴 업그레이드 (리눅스) |

ANACONDA. Conda package repository

- 아나콘다에서 관리하는 패키지 저장소
- <https://anaconda.org/anaconda/repo>
- 주요 명령어

| 명령어 | 설명 |
|--------------------|---------------|
| conda install 패키지명 | 패키지 install |
| conda update 패키지명 | 패키지 업그레이드 |
| conda remove 패키지명 | 패키지 제거 |
| conda list | 설치된 모든 패키지 조회 |
| conda list 패키지명 | 특정 패키지의 정보 조회 |



변수와 데이터 타입



- 1) 변수
- 2) 데이터 타입
- 3) 자료구조
 - 리스트 (List)
 - 튜플 (Tuple)
 - 딕셔너리 (Dictionary)
 - 집합 (Set)

변수

- 변수란

- 데이터를 담는 메모리 공간
- 값의 의미를 나타내는 이름

- 변수 선언 및 대입(할당) 구문

- 변수명 = 값
 - name = "홍길동"
 - age = 20
- 사용
 - print(name)
 - age = age + 20

| 파이썬 예약어 | | | | |
|---------|----------|---------|----------|--------|
| False | await | else | import | pass |
| None | break | except | in | raise |
| True | class | finally | is | return |
| and | continue | for | lambda | try |
| as | def | from | nonlocal | while |
| assert | del | global | not | with |
| async | elif | if | or | yield |

- 변수명 주기

- 명명규칙
 - 일반 문자(영어 알파벳 뿐 만 아니라 한글 한자 등 모든 일반 문자를 사용할 수 있다.), 숫자, 특수 문자는 _(underscore) 만 가능.
 - 숫자는 두번째 글자부터 가능
 - 예약어는 사용할 수 없다.
 - 대소문자 구분한다.
- 변수 명명의 일반적 관례
 - 소문자로 주고 단어 구별은 _ 로 한다.

변수

- 변수 선언 및 대입(할당)의 여러가지
 - 여러 변수 동시 선언 및 값 대입
 - `a = b = 0`
 - `a, b, c = 10, 20 , 30`
 - 변수 삭제
 - **del** 변수명
- 동적 타입 언어(Dynamic Type Language)
 - 변수의 데이터 타입을 실행 시점에 결정한다.
 - 변수 선언 시 데이터 타입을 지정하지 않는다.
 - **type**(값)
 - 변수나 값의 타입을 체크하는 함수
 - `type("abcde")`
 - `type(30)`

숫자형

- 정수(int)와 실수(float) 형이 있다.
 - 정수 : 10, 20, -1, -20, 0 등
 - 실수 : 20.1, 0.123411, 15.2321598
- 산술연산자

| 연산자 | 설명 | 비고 |
|-----|-------|-----------------------------|
| + | 더하기 | |
| - | 빼기 | |
| * | 곱하기 | $2 * 10 \Rightarrow 20$ |
| ** | 제곱 | $2 ** 10 \Rightarrow 1024$ |
| / | 나누기 | $10/3 \Rightarrow 3.333333$ |
| // | 나누기 몫 | $10//3 \Rightarrow 3$ |
| % | 나머지 | $10 \% 3 \Rightarrow 1$ |

논리형 (bool)

- 논리형
 - 참 거짓을 표현 하는 값
 - 값(Value)
 - **True** : 참
 - **False** : 거짓
 - 주로 조건문에서 많이 사용된다.
 - bool(값) 함수.
 - 다른 타입을 논리형 값으로 변환하는 함수
 - 빈 문자열, 숫자 0, None은 **False** 나머지는 True로 변환된다.

논리형(bool) 관련 연산자

■ 비교 연산자

- 두 값(모든 타입 비교가능)을 비교 후 그 결과를 논리형으로 리턴 한다.

| 연산자 | 설명 | 비고 |
|-----|---------|---------------------------------|
| == | 같다. | |
| != | 같지 않다 | |
| > | 크다. | 문자열일 경우 사전식 비교로 나중에 나오는 글자가 크다. |
| >= | 크거나 같다 | |
| < | 작다 | |
| <= | 작거나 같다. | |

논리형(bool) 관련 연산자

■ 논리 연산자

- 논리형 값을 연산해서 결과를 논리형 값으로 리턴 한다.

| 연산자 | 설명 |
|---------|--|
| & (and) | 두 값이 True 이면 결과 True 나머지 모두 False |
| (or) | 두 값이 모두 False 이면 False 나머지 모두 True |
| ^ | 두 값이 다를 경우 True 같으면 False |
| not | 피연산자를 반대로 부정한다. (True->False, False->True) |

■ 삼항 연산자(조건연산자)

- 조건이 True이거나 False이냐에 따라 결과값을 반환하는 연산자

| |
|---|
| Value1 if 조건식 else Value2 |
| 조건식이 True이면 Value1을 False이면 Value2를 반환 |
| str = '양수' if num >= 0 else '음수' |

None

- None
 - 아무 값도 없음을 나타내는 값
 - 다른 언어에서는 주로 null 을 사용한다.

문자열형(string)

- 문자열 만들기

- 작은 따옴표나 큰 따옴표로 감싼다.

```
name = '홍길동'  
address = "서울시 종로구"  
value1 = "I'm a student"  
value2 = '명수가 말했습니다."안녕 친구들" '
```

- 여러 줄 문자열 : ''' 또는 """ 으로 감싼다. (따옴표 세 개) 엔터를 알아서 처리한다.

```
desc = '''파이썬은 컴퓨터 언어입니다.  
파이썬은 귀도 반 로섬이라는 네덜란드 출신 프로그래머가  
1991년에 만들었습니다.'''
```

- 다른 타입의 데이터를 문자열로 바꾸기

- str(값)

```
value = str(200)  
value = "값 : "+str(2000)
```

문자열형(string)

▪ 문자형 연산

- 문자열 + 문자열

- 문자열을 합친다.
- 문자열 + 다른 타입 은 에러 발생한다. (str() 내장함수로 다른 타입을 string으로 변환 해야함)

```
name = "이영희"  
value = "이름 : "+name
```

```
value = "20" + 10 #에러  
value = str("20") + 10
```

- 문자열 * n

- 문자열은 n회 반복.

```
line = "-" * 20 => '-----'  
print('*' * 50)
```

- 문자열의 글자수 세기

- 내장 함수의 **len(문자열)** 사용

```
name = "홍길동"  
len(name)  
len("Hello World")
```

- 문자열 내에 특정 문자열이 있는지 확인

- * **in, not in** 연산자 사용

```
'he' in 'hello'  
'he' not in 'hello'  
hello 안에 he가 있는지/없는지 bool값으로 리턴
```

문자열 인덱싱(Indexing) 과 슬라이싱(Slicing)

- 문자열의 각 문자는 index 번호를 가진다.

- index 는 0부터 시작한다.

| | | | | | | | | | | | | |
|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|
| 안 | 녕 | 하 | 세 | 요 | . | | 반 | 갑 | 습 | 니 | 다 | . |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

- 인덱싱 : 문자열의 index를 이용해 문자를 조회한다.
 - 슬라이싱 : 문자열의 index의 **범위**로 문자열을 조회 한다.

- 인덱싱

- 문자열 [index]

- index 의 글자를 조회
 - index를 음수로 주면 뒤에서부터 조회
 - 변경은 안된다.
 - 문자열은 불변(Immutable) 이다.

```
value = '안녕하세요 반갑습니다'
print(value[0]) => 안
print(value[7]) => 반
print(value[-1]) => .
print(value[-4]) => 습
value[0] = '가' => 에러 발생
```

문자열 인덱싱(Indexing) 과 슬라이싱(Slicing)

- 슬라이싱(Slicing)

- 문자열 [시작 index : 종료 index : 간격]
 - 시작 index ~ (종료 index - 1)
 - 간격을 지정하면 간격만큼 index를 증/감한다. (생략 시 **1**이 기본 간격)
- 0번 index 부터 조회시 시작 index는 생략가능
 - str_value [: 5] => 0 ~ 4 까지 조회
- 마지막 index까지 (끝까지) 조회시 종료 index는 생략 가능
 - str_value[2 :] => 2번 index 에서 끝까지
- 명시적으로 간격을 줄 경우
 - str_value[: : **3**] => 0, 3, 6, 9.. index의 값 조회
 - str_value[1 : 9 : **2**] => 1, 3, 5, 7, 9 index의 값 조회

문자열 formatting

- 문자열에 문장 형태를 미리 만들어 놓고 값은 나중에 대입하는 방식으로 문자열을 만드는 것
 - 이름 : XXX 나이 : XXX 성별 : XXX
 - 기본 형식은 같은데 XXX에 들어갈 값들은 그때 그때 다를 경우 사용
 - format() 함수 이용
 - 문자열을 만들 때 나중에 넣은 곳을 {} 로 표시하고 format() 메소드에서 {}에 들어갈 값을 순서대로 넣는다.

```
myformat = '이름 : {}, 나이 : {}, 성별 : {}'  
value1 = myformat.format('홍길동', 20, '남성')  
value2 = myformat.format('이영희', 23, '여성')
```

- {} 을 변수 처리

```
myformat = '이름 : {name}, 나이 : {age}'  
value = myformat.format(name="김영수", age=20)
```

문자열 formatting

- f' '

- 파이썬 3.6에서 추가된 형식

```
height = '180.5 cm'  
weight = '75kg'  
value = f'키는 {height}이며, 몸무게는 {weight}입니다'
```

- 형식 문자를 이용한 formatting

"형식문자를 이용한 서식" % (형식 문자에 넣을 값)

```
>>> "이름 : %s, 나이 : %d" % ('홍길동', 32)  
이름 : 홍길동, 나이 32
```

| 형식 문자 (format 문자) | 비고 |
|----------------------|----|
|----------------------|----|

모든 형식문자는 %로 시작한다.

| | |
|----|-----|
| %s | 문자열 |
|----|-----|

| | |
|----|----|
| %d | 정수 |
|----|----|

| | |
|----|----|
| %f | 실수 |
|----|----|

| | |
|----|---|
| %% | % |
|----|---|

문자열 주요 메소드

▪ String 주요 메소드

| 메소드 | 설명 | 비고 |
|--------------------------------------|---|---|
| split(구분문자열) | 구분 문자열을 기준으로 나눈다. | 구분 문자열 생략 시 공백이 기본값 "사과,배,귤".split(",") |
| strip(), lstrip(), rstrip() | 앞뒤(strip) 앞(lstrip) 뒤(rstrip) 공백 제거. | ' abc '.strip() |
| replace('바꿀 문자열', '새문자열') | 바꿀 문자열을 새문자열로 바꾼다. | |
| in, not in | 문자열 안에 특정 문자열이 있는지(in) 없는지 (not in) 확인한다. 결과는 boolean으로 알려준다. | '사과' in '귤 복숭아 수박' '사과' not in '귤 복숭아 수박' |
| count('세려는 문자열') | 세려는 문자열이 몇 번 쓰였는지 확인 | 'helloworld'.count('l') |
| index(문자열) find(문자열) | 문자열이 몇 번째 index에 있는지 확인 | 없으면 index()는 Error발생 find()는 -1 리턴 여러 개일 경우 첫번째 것의 index만 리턴. |
| upper(), lower() | 대문자(upper) 소문자(lower)로 변환 | |
| startswith("문자열") endswith("문자열") | 문자열로 시작/끝 나는지 확인 | |

데이터 타입 변환 함수

▪ 정수로 변환

- **int(값)**

- 실수의 경우 소수점 이하를 버린다.
- int(5.32), int('230')
- int('50.123') => Error 발생. 정수형태의 문자열일 경우만 가능

▪ 실수로 변환

- **float(값)**

- float(30) => 30.0
- float('50.123')
- float('40.7') => 40.7

▪ 문자열로 변환

- **str(값)**

- 모든 타입의 값을 문자열로 변환한다.
- str(10), str(20.5), str(True)

▪ 논리값으로 변환

- **bool(값)**

- bool(1) => True
- bool('a') => True

- **변환 기준**

- 숫자 : 0 - False, 음수, 양수 - True
- 문자열
 - 0글자 - False, 한 글자 이상 - True
- None : False



자료 구조



- 1) 리스트
- 2) 튜플
- 3) 딕셔너리(사전)
- 4) Set(집합)

리스트(List)

- 값을 순서대로 모아서 관리하는 구조
- 특징
 - 원소, 요소(Element), 항목(item) : 자료구조에 모아 관리되는 값(value) 하나 하나
 - len(리스트) : 리스트의 원소 개수 조회
 - 각 원소들은 index를 가지며 index로 관리(조회나 변경)된다.
 - 다른 타입의 값들을 모을 수 있다.
 - 리스트의 원소들은 다른 값으로 바꿀 수 있다.
- 리스트 만들기
 - 구문
 - [값, 값, 값, ..]
 - 값들은 , 로 구분해서 넣는다.
 - 값들의 타입이 같지 않아도 된다.
 - Index는 0부터 시작한다.

```
list1 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
list2 = ['a','b','c','d','e','f','g']
list3 = [20.7, 20.8, 30.7, 28.5, 92.6]
list4 = [10, "abc", False, None, 20.7]
```

리스트(List)

- 리스트 인덱싱(Indexing) 슬라이싱(Slicing)

- Indexing과 slicing을 이용한 원소 조회

```
num_list = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

num_list[0] => 0번 index 원소 조회

num_list[100] => 100번 index 원소 조회 => **없는 index 조회 시 IndexError 발생**

num_list[1 : 10] => 1~9 index 원소들 조회 => **끝 index는 포함되지 않는다.**

num_list[: 5] => 시작이 0번 index일 경우

num_list[5 :] => 마지막 index까지 조회일 경우

num_list[: : 3] => 세 개 index씩 건너 뛰어 조회

- Indexing과 slicing을 이용한 원소 변경

```
num_list = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

num_list [0] = 20

num_list[1 : 5] = [100,200,300,400]

num_list[0 : 3] = [0,0,0,0,0,0,0,0,0,0]

num_list[: 5] = [] => 0~4 index의 원소들을 삭제

del num_list[3] => 3번 index의 원소 삭제

리스트(List)

- + 연산자를 이용해 리스트 합치기
 - 리스트 + 리스트 하면 두 리스트를 합친다.

```
num_list1 = [1, 2, 3]
num_list2 = [10, 20, 30]
result = num_list1 + num_list2
#result => [1, 2, 3, 10, 20, 30]
```

- * 연산자를 이용해 리스트 반복하기
 - 리스트 * n
 - 리스트의 원소를 n번 반복한 리스트를 반환한다.

```
a = [1, 2, 3]
b = a * 3 => [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- in, not in 연산자
 - 어떤 값이 리스트(자료구조)의 원소에 있는지 여부를 반환

```
a = [1, 2, 3]
print(3 in a)      # True
print(3 not in a)  # False
```

리스트(List)

- 중첩 리스트

- 리스트의 원소로 리스트가 추가된 것

```
my_list = [[1, 2, 3], [10,20,30,40], ['가', '나', '다']]  
my_list[0] => [1,2,3]  
my_list[1][0] => 10  
my_list[2][1] => '나'
```

- 리스트 대입

```
a, b, c = [10, 20 , 30]  
# a= 10, b = 20, c = 30
```

- 각 변수에 리스트의 원소들이 순서대로 들어간다.

리스트(List) – 주요 메소드

▪ List 주요 메소드

| 메소드 | 설명 | 비고 |
|----------------------|--|---|
| append(값) | 값을 마지막 원소로 추가 | num_list.append(1) |
| extend(리스트) | 인수로 받은 리스트의 원소들을 추가(+ 연산 효과) | num_list.extend([1,2,3]) num_list + [1, 2, 3] 와 동일 |
| sort([reverse=True]) | 리스트 내 원소를 오름차순 정렬 sort(reverse = True): 내림차순 정렬 | 리스트내 원소들의 타입이 같아야 한다. |
| insert(index, 값) | 값을 index에 삽입한다. | |
| remove(값) | 리스트에서 값과 일치하는 원소를 삭제한다. | 일치하는 것 중 첫번째 원소만 삭제한다. |
| index(값 [, 시작idx]) | 값의 index번호를 반환한다. | 찾기 시작할 index 지정가능. |
| pop([index]) | index의 값을 삭제하면서 반환한다. | index생략 시 마지막 원소를 반환 삭제한다. |
| count(값) | 매개변수에 전달한 값의 개수를 반환한다 | |
| clear() | 모든 원소들을 한번에 제거한다. | |

튜플(Tuple)

- 튜플은 리스트와 같이 순서대로 원소들을 저장하는 자료구조이다.
- 리스트와 다른 점은 **원소를 변경할 수 없다**.
- 튜플은 각 위치(Index)마다 정해진 의미가 있고 그 값이 한번 설정되면 바뀌지 않는 경우에 많이 사용한다.
 - 튜플은 값의 변경되지 않으므로 안전하다.
- len(튜플)
 - 튜플의 원소 개수 조회
- in, not in 연산자
 - 어떤 값이 튜플(자료구조)의 원소에 있는지 여부를 반환

튜플(Tuple)

- 튜플 만들기
 - (값, 값, 값)
 - () 생략 가능
 - 값, 값, 값
 - 원소가 1개인 튜플
 - 원소 뒤에 , 를 붙인다.

```
a = (10, 20, 30, 40, 50)
```

```
b = 10, 20, 30, 40, 50
```

```
c = ('가', '나', '다')
```

```
d = '가', '나', '다'
```

```
e = (100, )
```

```
f = 200,
```

```
g = (200) => int
```

- 리스트를 튜플로 변환
 - **tuple(리스트)** 함수 이용

```
nums = [1,2,3,4,5]
```

```
nums_tuple = tuple(nums)
```

튜플(Tuple)

- 튜플 인덱싱과 슬라이싱

| | |
|---|--|
| <pre>nums = (1, 2, 3, 4, 5, 6, 7, 8, 9) nums[0] nums[5]</pre> | <pre>nums[3 : 6] nums[: 5] nums[3 :] nums[0 : 6 : 2]</pre> |
|---|--|

- 튜플 합치기와 곱하기

- 리스트와 같이 + 로 합치고 * 로 반복할 수 있다.

| | |
|--------------------------------------|--|
| <pre>(1, 2, 3) + (10, 20, 30)</pre> | <pre>#=> (1, 2, 3, 10, 20, 30)</pre> |
| <pre>(10, 20, 30) * 3</pre> | <pre>#=> (10, 20, 30, 10, 20, 30, 10, 20, 30)</pre> |

- 주요 메소드

- 튜플은 값을 변경하는 메소드는 없다.
- index(값)
 - 값과 동일한 첫번째 원소가 몇 번째 index에 있는지 반환한다.
- count(값)
 - 튜플 안에 매개변수로 전달한 값이 몇 개 있는지 반환한다.

딕셔너리(Dictionary, 사전)

- 딕셔너리는 값을 키(key)-값(value) 쌍의 형태로 저장하는 자료구조이다.
 - 리스트나 튜플의 index의 역할을 하는 key를 직접 지정한다.
- 딕셔너리 만들기
 - { 키 : 값, 키 : 값, 키 : 값 }
 - 키(key)는 불변의 값들만 사용 가능하다. (숫자, 문자열, 튜플)
- 딕셔너리의 원소 조회 및 변경
 - 딕셔너리[키]
 - 없는 키로 조회 시 **KeyError** 발생
 - 딕셔너리[키] = 값
 - 키가 있으면 변경이고 없으면 추가

```
fruit_cnt = {'사과' : 20, '귤' : 30, '참외' : 13}
fruit_cnt['사과']          #=> 20
fruit_cnt['수박'] = 50      #=> 수박은 없는 키이므로 추가
fruit_cnt['사과'] = 35      #=> 사과는 있는 키이므로 변경

person = {'이름':'홍길동', '나이':20, '직업':'학생'}
```

딕셔너리(Dictionary, 사전) – 주요 메소드

| 메소드 | 설명 |
|---------------|--|
| pop(Key) | Key 와 연결된 값을 반환하면서 삭제한다. Key가 없으면 KeyError 발생 |
| clear() | 모든 원소 삭제 |
| del dict[Key] | 특정 Key의 값을 삭제 |
| get(Key, 기본값) | 키와 연결된 값을 반환한다. 단 찾는 키의 값이 없을 경우 기본값을 반환한다. 기본값 생략시 Key의 값이 없으면 None 반환 |
| items() | Key Value 쌍의 튜플로 묶어 리턴 |
| keys() | Key값들만 모아서 리턴 |
| values() | Value들만 모아서 리턴 |

len(사전)

사전의 원소(key-value 쌍) 개수 조회

in, not in 연산자

어떤 값이 **사전의 key**로 있는지 여부를 반환

집합 (Set)

- Set은 중복되는 값을 허용하지 않고 순서를 신경 쓰지 않는다.
- Set은 indexing과 slicing을 지원하지 않는다
- Set 만들기
 - {값, 값, 값 }
- set() 를 이용해 다른 자료구조를 set으로 변환
 - set() 에 인수로 다른 자료구조 객체를 넣어 만든다.
 - 다른 자료구조의 원소 중 중복을 빼고 조회할 때 set()를 이용해 Set으로 변환 한다.

```
set1 = { 1, 2, 3, 4, 5}
set2 = {1, 2, 2, 2, 3}
set3 = set([1, 2, 3, 3, 4, 4, 4, 5, 5, 5])
set4 = set((1, 2, 3, 3, 4, 5, 5))
set5 = set({'a':1, 'b':2, 'c':3})
```

집합(Set) – 주요 메소드

- add(값)
 - 집합에 요소 추가
- update(자료구조)
 - 한번에 여러 개의 요소를 추가
- 삭제
 - pop()
 - 값을 하나씩 반환하면서 제거한다.
 - remove(값)
 - 값을 찾아서 삭제한다.
 - 값이 set내에 없으면 **KeyError** 발생
- len(튜플): Set의 원소 개수 조회
- in, not in 연산자:
 - 어떤 값이 Set(자료구조)의 원소에 있는지 여부를 반환

집합(Set) – 연산

- 합집합

- 집합 | 집합
- 집합.union(집합)

- 교집합

- 집합 & 집합
- 집합.intersection(집합)

- 차 집합

- 집합 - 집합
- 집합.difference(집합)

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
```

```
set1 | set2      #=> {1, 2, 3, 4, 5}
```

```
set1 & set2      #=> {3}
```

```
set1 - set2      #=> {1, 2}
```




제어문



1) 조건문

- if else

2) 반복문

- while
- for in

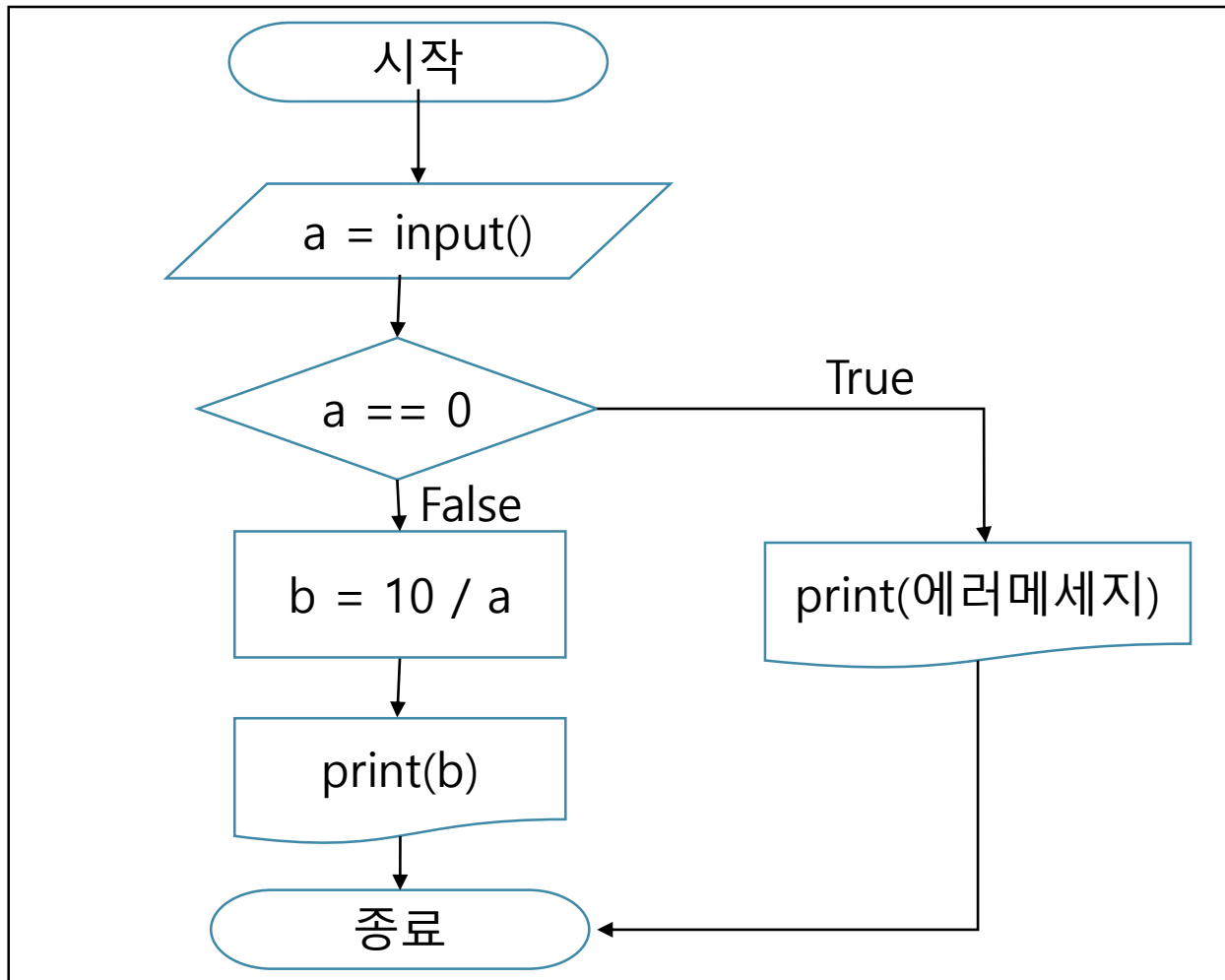
3) 컴프리헨션 (Comprehension)

논리형(bool, boolean) 다시 보기

- True와 False를 나타내는 자료형
- 비교 연산자
 - 두 값을 비교하며 결과가 논리형으로 나온다.
- 논리 연산자
 - 피연산자가 논리형 값인 연산자
 - & - and, | - or, ^, not
- 논리형 데이터를 사용하는 곳에 다른 타입의 데이터를 넣을 경우 다음은 False로 처리한다.
 - 숫자 : **0, 0.0**
 - 문자 : **빈 문자열**
 - **length가 0인 자료 구조(튜플, 리스트, 딕셔너리)**
 - **None**
- **bool(값)** : 인수인 '값'이 True인지 False인지 알려준다.

조건문(분기문)

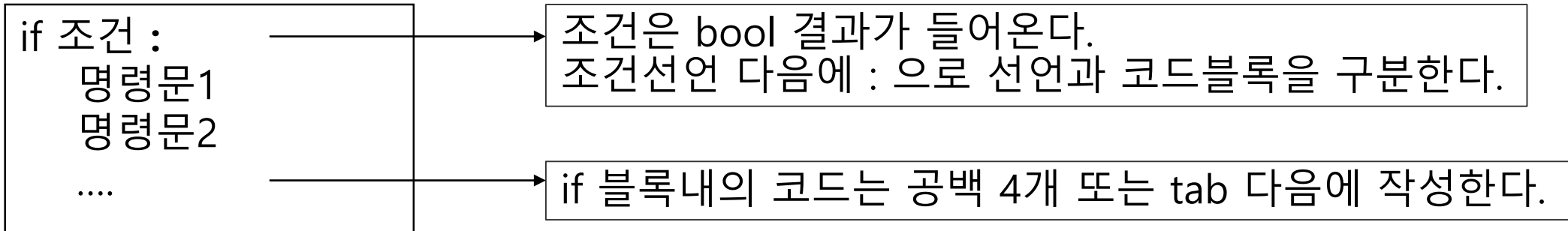
- 프로그램이 명령문들을 실행하는 도중 특정 순서에서 흐름의 나눠져야 하는 경우 사용한다.



입력 받은 a 의 값이 0인지 여부에 따라 두가지 흐름으로 분기된다.

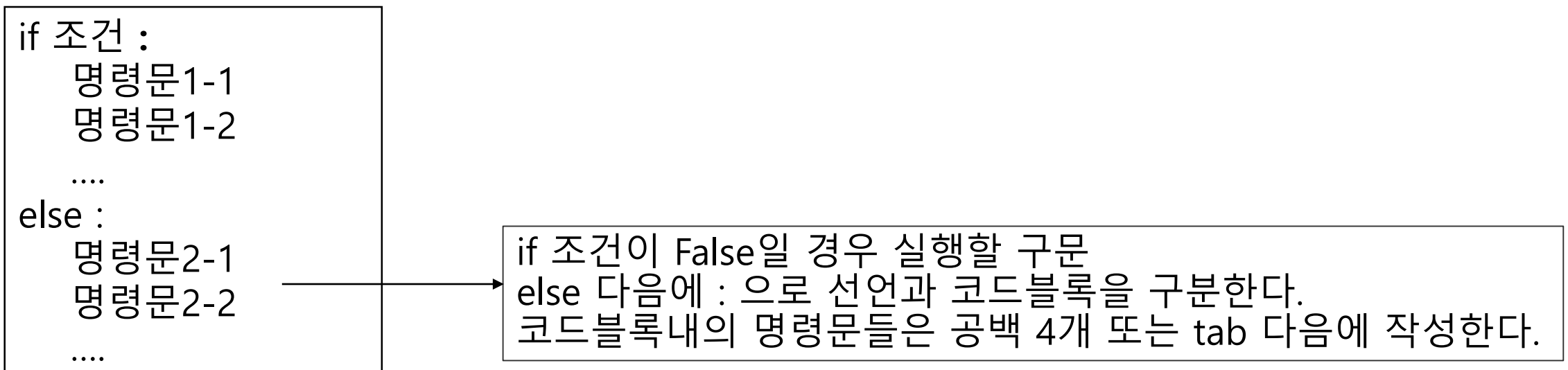
조건문(분기문) – if

▪ if 문



▪ if else 문

- 조건이 True일 경우와 False일 경우 실행할 구문이 분리된 경우



조건문(분기문) - if

▪ if elif

- 조건이 여러 개일 경우 사용한다.
- 마지막에 else가 올 수 있다.
- 조건은 처음 선언한 조건부터 순서대로 체크하여 True인 코드블록을 실행하고 빠져 나온다.

```
if 조건1:  
    명령문1-1  
    명령문1-2  
    ....  
elif 조건2:  
    명령문2-1  
    명령문2-2  
    ....  
elif 조건3 :  
    명령문3-1  
    명령문3-2  
else:  
    명령문4
```

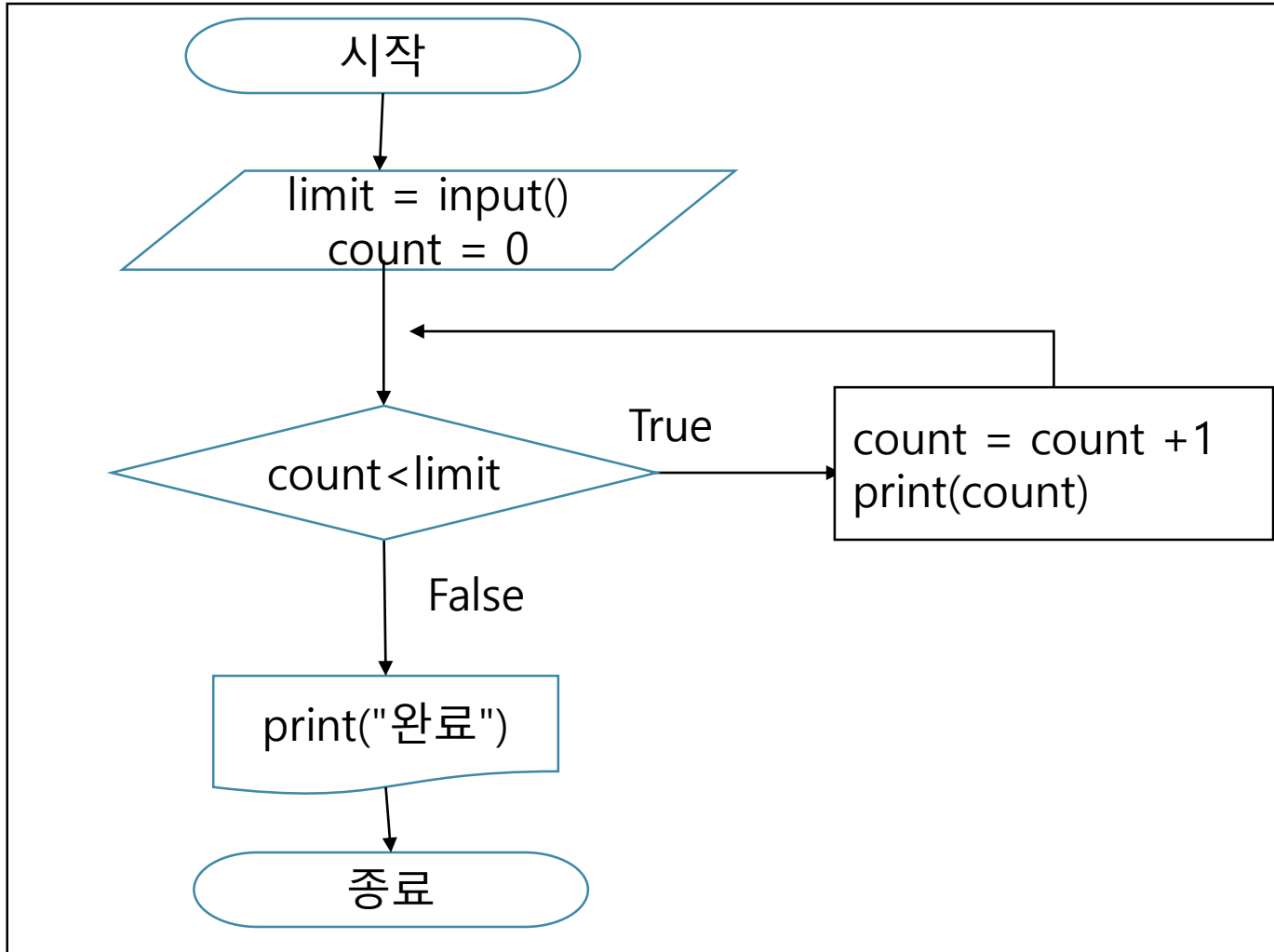
elif 마지막에 : 으로 선언과 코드블록을 구분한다.
코드블록내의 명령문들은 공백 4개 또는 tab 다음에 작성한다.

반복문

- 특정 조건이 True인 동안 명령문을 반복해서 실행한다.
 - while문
- Iterable한 객체가 값이 없을 때 반복 조회한다.
 - for in
 - **Iterable한 객체란**
 - 반복가능한 객체를 말한다.
 - 자료구조들, 문자열이 대표적인 Iterable이다.

반복문 - while

- 조건이 True인 동안 구문을 반복한다.



while 조건 :
반복할 구문1
반복할 구문2
....

- while 선언 마지막에 : 을 넣는다.
- 반복구문은 공백 4칸이나 tab을 입력 후 작성한다.

반복문 – for in

- for 문은 Iterable 타입의 객체가 가지고 있는 값들을 순회하며 조회할 때 사용.
 - Iterable 객체는 여러 개의 값들로 구성되어 있으며 반복 조회가 가능한 타입을 통칭한다.
 - 리스트, 튜플, 딕셔너리, set, 문자열 등이 있다.

```
for 변수 in Iterable :  
    반복구문  
    반복구문  
....
```

```
for num in (1, 2, 3, 4, 5) :  
    print(num)  
  
print('반복문완료')
```

- for 선언 후 : 으로 선언부와 구현부를 나눈다.
- 변수는 반복할 때 조회되는 원소를 반복구문에서 사용하도록 저장하는 임시변수이다.
- Iterable 안의 모든 값들을 다 조회하면 반복구문은 멈춘다.
- 반복구문은 공백4개 또는 tab을 입력 후 작성한다.
- 반복구문이 끝나면 공백없이 코드를 작성하면 된다.

continue와 break로 반복문 제어

- continue
 - 나머지 부분을 실행하지 않고 다음 반복을 시작한다.
- break
 - 반복문을 멈춘다.

```
for num in range(1, 20):  
    if(num % 5 != 0):  
        continue  
    print(num)  
print("for문 종료")
```

```
for num in range(1, 20):  
    if(num % 5 != 0):  
        break  
    print(num)  
print("for문 종료")
```

for in 연관 내장 함수 - range() 함수

- 연속된 정수들을 만들 때 사용한다.

- 구문

- range([시작값], 멈춤값, [증감값])
 - 시작 값은 포함하고 멈춤 값은 포함하지 않는다.
 - 증감 값 만큼 증감한 연속된 정수들을 만든다.
 - 시작 값 생략 시 0이 기본값
 - 증감 값 생략 시 1이 기본값
 - 멈춤값은 생략할 수 없다.

```
range(1, 20, 5) => [ 1, 6, 11, 16 ]  
range(1, 5)     => [ 1, 2, 3, 4 ]  
range(7)        => [ 0, 1, 2, 3, 4, 5, 6 ]
```

```
list(range(1,20))  
tuple(range(1,20))  
set(range(1,20))
```

```
for num in range(1,20) :  
    print(num)
```

for in 관련 내장 함수 - enumerate()

- 반복 조회 시 현재 원소의 index와 원소를 튜플로 묶어서 반환한다.

```
strs = ['A', 'B', 'C']  
for idx, str in enumerate(strs):  
    print(idx, str)
```

```
0 'A'  
1 'B'  
2 'C'
```

- 전달인자

- start : 시작할 index값을 지정한다. 기본값은 0
 - enumerate(strs, start = 1) : 1부터 시작한다.

for in 관련 내장 함수 - zip() 함수

- 여러 개의 자료구조 객체를 받아 같은 index의 값끼리 튜플로 묶어 준다.

```
a = [1, 2, 3]
b = [10, 20, 30]
for z in zip(a, b):
    print(z)
```

```
(1, 10)
(2, 20)
(3, 30)
```

- 묶는 자료구조 객체의 개수는 상관없다.
- 각 자료구조 객체의 크기가 다를 경우 작은 것의 개수에 맞춘다.

컴프리헨션(Comprehension)

- 기존 자료구조가 가진 원소들을 이용해 새로운 자료구조를 만드는 구문
 - 주로 기존 자료구조의 원소들을 처리한 결과를 새로운 자료구조에 넣을 때 사용한다.
 - 리스트 컴프리헨션
 - 딕셔너리 컴프리헨션
 - 집합(Set) 컴프리헨션
 - 튜플 컴프리헨션은 없다.
 - 딕셔너리/집합 컴프리헨션은 파이썬 3 부터 지원

| | |
|---|--|
| [out for out in list] [out for out in list if 조건식] | [num+10 for num in num_list] [num+10 for num in num_list if num%2 == 0] |
|---|--|

| | |
|---|--|
| { k:v for k in dictionary } { k:v for k in dictionary if 조건식 } | {i:v for i, v in enumerate(str_list)} {i:v for i, v in enumerate(str_list) if i%2==0} |
|---|--|

| | |
|---|--|
| { out for out in list } { out for out in list if 조건식 } | {num-10 for num in num_list} {num-20 for num in num_list if num%2 == 0} |
|---|--|



함수(Function)



- 1) 함수란
- 2) 함수 구문
- 3) 매개변수 관련 문법
 - 매개변수
 - 가변인자
 - 키워드 인수
- 4) 함수 호출

함수란

- 하나의 작업, 기능, 동작을 처리하기 위한 명령문들의 묶음.
 - 만들어진 함수는 동일한 작업이 필요할 때 마다 재사용될 수 있다.
 - 함수를 만드는 것을 **함수 정의**라고 한다.
 - 정의된 함수를 사용하는 것을 **함수 호출**이라고 한다.
 - 파이썬에서 함수는 일급 시민 객체(First Class Citizen Object)이다.
 - 일급 객체란 변수에 할당할 수 있고, 인수로 전달할 수 있고, 반환 값으로 반환할 수 있는 객체를 말한다.
- 함수의 호출

함수이름([인수, 인수,])

```
>>> int('100')  
100
```

- int 는 호출하는 함수의 이름이다.
- 괄호안의 '100' 은 함수의 **인수(Argument)** 라고 하며 함수에게 전달하는 값이다.
- 이 함수는 처리결과를 반환하는데 그 값을 **반환 값(return value)** 라고 한다.
반환 값이 없는 함수는 None이 반환된다.

함수 만들기

▪ 함수의 정의

- 새로운 함수를 만드는 것을 함수의 정의라고 한다.
- 함수의 선언부와 구현부로 나누어진다
 - 함수의 선언 부(Header) : 함수의 이름과 인수를 받을 변수를 지정한다.
 - 함수의 구현 부(Body) : 함수가 호출 되었을 때 실행할 실행문들을 지정한다.

```
def 함수이름( [인자, 인자, ..] ) :  
    실행구문1  
    실행구문2  
    실행구문3  
    ...  
    [return [결과값]]
```

선언 부(Header)

구현 부(body)

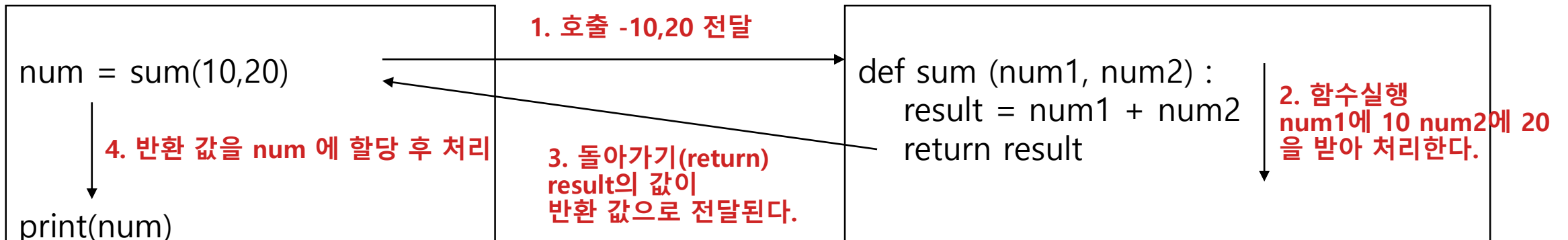
공백 4개

- 함수 선언 마지막에는 : 을 넣어 구현부와 구분한다.
- 매개변수(인자, Parameter) 는 인수를 받기 위한 변수로 0개 이상 선언할 수 있다.
- 함수의 실행구문은 반드시 공백4개 또는 탭 이후에 작성한다.
- 결과값이 있을 경우 return 구문을 넣고 없을 경우 return은 생략할 수 있다.
 - return 값이 없는 함수는 **None**을 반환 한다.

함수의 매개변수(인자, Parameter)

- 매개변수는 선언부에 선언하는 변수로 호출할 때 전달된 인수를 저장하는 변수.
- 매개변수는 0 ~ N 개 선언이 가능하다.
 - 2개 이상 선언 시 , 를 구분자를 사용한다.
- 매개변수는 로컬 변수로 함수 내에서만 사용할 수 있다.
 - 로컬변수 : 함수 안에 선언된 변수로 함수 안에서만 사용할 수 있고 외부에서 사용은 안된다.

흐름



함수의 매개변수(인자, Parameter)

- 기본값이 있는 매개변수

- 매개변수 선언 시 값을 할당하면 기본값이 된다.
- 호출 시 인수가 전달되지 않으면 할당한 기본값을 사용한다.
- 기본값을 할당한 매개변수 뒤에 기본값을 할당하지 않은 매개변수는 선언 할 수 없다.

```
def sum(num = 10):  
    return num + 100
```

```
sum()      ➔ 110  
sum(200)   ➔ 300
```

```
def printInfo(name, age=20, job) :  
    ...
```

SyntaxError 발생

- 키워드 인수(Keyword Argument)

- 호출 할 때 매개변수의 변수명을 명시하면 순서와 상관없이 값을 전달 할 수 있다.

```
def printInfo(name, job, age=20, address='서울') :  
    ....
```

```
printInfo('홍길동', '회사원', address='인천')  
printInfo(name='김영수', job='학생')
```

함수의 매개변수 - 가변인자

- 인수의 개수를 정하지 않고 받을 경우 사용한다.
- 매개변수 앞에 * 를 붙인다.
 - 가변인자는 튜플로 처리된다.
- 매개변수 앞에 ** 를 붙인다.
 - 가변인자는 딕셔너리(사전)으로 처리된다.
 - 키=값 형식으로 인수를 전달해야 한다.

```
def print( *num ) :  
    for i in num:  
        print(i)  
  
print(10, 20, 30, 40, 50)
```

```
def printInfo( **info) :  
    for a in info.keys():  
        print(info[a])  
  
printInfo(name='홍길동', age=20)
```

- 가변인자는 하나만 선언가능 하며 마지막 변수로 선언되어야 한다.
- * 와 ** 가변인자는 하나씩 같이 선언할 수 있다.

함수 호출 - 인수 전달, 반환 값

- 함수 호출 구문

- 기본 구문 : 변수 = 함수명(인수, 인수)

- 인수 전달

- 선언된 매개변수 순서대로 전달

```
def test(num1, num2, num3) :
```

```
test(10, 20, 30)
```

- 호출 시 변수명을 지정해서 전달

- 일반적으로 기본값이 있는 매개변수에는 값을 전달 하지 않을 경우 사용.

```
def test(num1, num2, num3) :
```

```
test(num1 = 10, num2 = 20, num3 = 30)
```

```
def test(num1=10, num2=20, num3=30) :
```

```
test(num2 = 1000)
```

- 반환 값이 없는 함수 호출 하면 **None**이 반환 된다.



파이썬 객체지향 프로그래밍

클래스와 객체



- 1) 객체지향 프로그래밍
- 2) 클래스 정의, 객체 생성
- 3) 객체의 속성과 메소드
- 4) 특수 메소드 (Special Method)

객체지향 프로그래밍

- 객체
 - 연관성 있는 데이터와 기능을 가지고 있는 프로그램 모듈
 - 속성(Attribute)
 - 객체의 데이터
 - 객체의 상태
 - 메소드(Method)
 - 객체가 제공하는 기능
- 클래스 (Class)
 - 객체가 가지는 속성과 메소드를 정의한 객체의 설계도
- 클래스로부터 객체를 생성해 사용한다.
 - 인스턴스화 (Instantiate)
 - 객체를 생성하는 작업
 - Instance
 - 클래스로부터 생성된 객체

클래스 정의

■ 클래스 정의

```
class 클래스이름 :
```

클래스 코드 블록

```
class Person:
```

```
def __init__(self, name, age):  
    self.name = name  
    self.age = age
```

```
def go(self):  
    print("간다")
```

```
def printInfo(self):  
    print("이름 {}, 나이 {}".format(self.name, self.age))
```

instance 생성 및 사용

```
변수 = 클래스이름() #instance 생성
```

```
변수.속성          #attribute 사용  
변수.메소드        #메소드 호출
```

```
p1 = Person('이순신', 20)  
p1.age = 30  
p1.go()  
p1.printInfo()  
p1.address = "서울시 종로구 "
```

```
p2 = Person('홍길동', 30)  
print(p2.name)  
p2.go()  
p2.printInfo()
```

객체의 속성(Instance 변수)

- 객체의 속성(Attribute) 는 가변이다.
 - 객체의 데이터
 - 클래스 선언 시 instance 변수를 선언하지 않는다.
 - 생성자(Initializer)나 함수에서 동적으로 instance 변수를 만들어 넣는다.
 - 객체를 이용해 직접 변수를 만들어 값을 대입할 수 있다.
 - 같은 클래스에서 생성된 객체들 이라도 서로 다른 instance 변수를 가질 수 있다.
 - 객체.__dict__ 를 이용해 객체가 가지고 있는 instance 변수들을 조회할 수 있다.

객체의 메소드(Instance Method)

- 메소드는 객체가 제공하는 기능이다.
 - 클래스에 정의한 객체의 함수를 메소드라고 한다.
 - 메소드는 첫번째 매개변수로 반드시 객체 자신(메소드 소유객체)을 받는 변수를 선언한다.
 - 첫번째 인자로 선언한다.
 - 변수명은 보통 self를 지정한다.
 - 이 인자를 선언하지 않으면 호출할 수 없다.

```
p.set_info("홍길동", 20, "서울")
```

```
def set_info(self, name, age, address):  
    ....
```

정보 은닉

- 객체가 가진 instance 변수나 메소드를 다른 곳에서 호출하지 못하도록 하는 것
 - 데이터 보호가 주 목적
 - C++, Java 등은 접근 제한자를 이용해 처리하는데 파이썬은 접근제한자가 없다.
 - 파이썬은 원칙적으로 접근 제한을 막는 방법이 없다.
- instance 변수명이나 메소드 이름에 `_` (더블언더스코어) 를 붙이면 private 처리
 - 이름을 `_클래스명_원래이름` 형식으로 변환한다.
 - 이름이 `_`로 시작해도 `_`로 끝나면 public으로 간주한다.(이름을 안바꾼다)
 - Person의 `_name` => `_Person_name` 으로 이름을 바꾼다.
 - 같은 클래스 내에서는 원래 이름으로 접근 하되 외부에서는 바뀐 이름으로 호출 해야 한다.
 - 값을 변경하는 메소드, 조회하는 메소드를 제공한다.

상속

- 기존 클래스를 확장하여 instance 변수나 메소드를 추가하는 방식
 - 기반(Base) 클래스, 상위(Super) 클래스, 부모(Parent) 클래스
 - 물려 주는 클래스. 좀더 추상적
 - 파생(Derived) 클래스, 하위(Sub) 클래스, 자식(Child) 클래스
 - 상속하는 클래스. 좀더 구체적

class 클래스이름 (**Super클래스명**[,**Super클래스명**,...]):

클래스 코드 블록

```
class Person:  
    pass
```

```
class Student(Person):  
    pass
```

```
class UniversityStudent(Person):  
    pass
```

상속

- 메소드 재정의 (Method Overriding)
 - 기반 클래스의 메소드의 구현부를 파생클래스에서 재 구현하는 것.
 - 메소드 구현을 좀더 구체화한다.
- super() 내장함수
 - 파생 클래스에서 기반 클래스의 instance를 반환(return)해주는 함수
 - 파생클래스에서 기반클래스의 메소드를 호출하려면 반드시 호출해야 한다.
 - super().메소드명()
- 기반 클래스의 Instance 메소드를 호출할 때 - super().메소드()
- 같은 클래스의 Instance 메소드를 호출할 때 - self.메소드()

객체 관련 유용한 내장 함수, 특수 변수

- `isinstance(객체, 클래스이름) : bool`
 - 객체가 두번째 매개변수로 지정한 클래스의 타입이면 True, 아니면 False 반환
- `객체.__dict__`
 - 객체가 가지고 있는 instance 변수들과 대입된 값을 사전(dictionary)에 넣어 반환
- `객체.__class__`
 - 객체의 타입을 반환

특수 메소드(Special Method)

- 특수 메소드란

- 클래스에 정의 하는 약속된 메소드로 객체가 특정한 상황에서 사용될 때 자동으로 호출되는 메소드들이다.
- 메소드 명이 더블 언더스코어로 시작하고 끝난다.
 - `__init__()`, `__str__`
- 매직 메소드(Magic Method), 던더(DUNDER) 메소드라고도 한다.
- <https://docs.python.org/ko/3/reference/datamodel.html#special-method-names>

주요 특수 메소드 - 객체 생성/소멸

- `__init__(self [, ...])`
 - 생성자 (Initializer)
 - `self`는 새롭게 생성되는 instance가 전달된다.
 - 객체 생성시 instance 변수 초기화에 사용
- `__del__(self)`
 - 소멸자(finalizer)
 - 객체가 소멸되기 직전에 호출된다.
 - 객체는 참조 카운트가 0일 때 Garbage collection에 의해 소멸된다.
 - 인터프리터 종료 시 아직 남아있는 객체들의 소멸자 메소드 호출은 **보장되지 않는다**.

주요 특수 메소드 - 문자열 표현

- `__repr__(self)`
 - Instance를 문자열로 바꿀 때 사용할 문자열 값을 만들어 반환한다.
 - 내장함수 `repr()`에 전달되면 반환될 문자열로 다시 `eval()`에 전달하면 원래 Instance로 변환될 수 있는 문자열로 반환한다.
 - 대화형 IDE에서 변수를 값을 출력할 때 호출하는 메소드.
- `__str__(self)`
 - `__repr()`과 비슷하게 Instance를 문자열로 바꿀 때 사용할 문자열 값을 반환(return)한다.
 - 내장 함수 `str()` 나 출력 함수에 의해 호출된다.
 - 출력 시 객체에 `__str__`이 없으면 `__repr__`이 호출된다.
 - 주로 instance의 속성값들을 하나의 문자열로 합쳐 리턴 하도록 구현한다.

주요 특수 메소드 - 비교 연산자 관련 표현

- `__eq__(self, other) : self == other`
 - `==` 로 객체의 내용을 비교할 때 정의해야 한다.
- `__lt__(self, other) : self < other, __gt__(self, other) : self > other`
 - `min()`이나 `max()`에서 인수로 사용할 경우 정의해야 한다.
- 그 외 비교 연산자 관련 메소드
 - `__le__(self, other), __ge__(self, other), __ne__(self, other)`
- 산술 연산자 메소드
 - `__add().__`

클래스 메소드와 변수

- 객체가 아닌 클래스 자체의 메소드와 변수
 - 객체 별로 생성되는 것이 아니라 한 클래스에 속하게 된다.
 - 클래스 메소드는 클래스 변수와 관련된 기능을 제공하는 메소드를 만들 때 사용
- 클래스 변수 선언
 - 클래스 블록에 선언한 변수로 Class이름.변수명 으로 호출.
- 클래스 메소드
 - 메소드 선언부에 @classmethod 데코레이터를 붙인다.
 - 반드시 한 개의 매개변수를 선언해야 한다.
 - 첫번째 매개변수로 클래스 자신을 받는 변수를 선언해 다른 클래스 멤버들을 호출 할 수 있다.
- 클래스 변수/메소드 호출
 - 클래스이름을 이용해 호출한다.
 - 객체를 이용해서 호출 할 수 있다.

정적 메소드(Static Method)

- 클래스에 선언된 메소드로 객체와 상관없이 클래스의 기능을 제공한다.
 - 객체와 상관없는 클래스 만의 단순 기능을 제공하는 메소드를 만들 때 사용.
- 구현
 - 메소드 선언부에 @staticmethod 데코레이터를 붙인다.
- 클래스 메소드와 다르게 class를 받는 매개변수를 선언하지 않는다.
 - 클래스 변수에 직접 접근하지 못한다.
 - Class이름.변수명 으로 사용한다.
- 호출
 - 클래스이름.메소드명() 으로 호출한다.



모듈과 패키지



- 1) 모듈
- 2) import
- 3) 패키지

모듈이란

- 독립적인 기능을 가지고 재사용가능한 프로그램 단위를 모듈이라고 한다.
- 파이썬에서 모듈은 각각의 소스 파일을 말한다.
 - .py 로 저장한 하나의 파일이 하나의 모듈이 된다.
 - 소스파일에 저장된 함수나 클래스 들을 다른 python 프로그램에서 사용할 수 있다.
 - 그러므로 모듈은 라이브러리로 볼 수 있다.
- 모듈의 종류
 - **표준 모듈**
 - 파이썬에 내장된 모듈
 - **3rd Party 모듈**
 - 특정 개발업체나 개발자들이 만들어 배포하는 모듈
 - **사용자 작성 모듈**
 - 개발자가 재사용을 위해 직접 만든 모듈

import

- 내장함수이외의 모듈을 사용하기 위해서 해야 불러들이는 작업
 - 사용하려는 외부 library를 메모리에 올리는 작업
 - 모듈내의 함수나 변수를 사용하기 위해서는 **반드시** import를 해야 한다.
- 기본구문

```
import 모듈명 as 별칭
```

```
from 사용하려는것이 있는 경로 import 사용하려는 것 as 별칭
```

import

| 구문 | 예 | 비고 |
|--------------------------------------|---|---|
| import 모듈명 | my_lib.py 를 사용할 때 import my_lib my_lib.test1() | 확장자인 .py는 넣지 않는다. |
| from 모듈명 import 모듈내요소 | my_lib.py 내의 test() 함수 import from my_lib import test1 test() | 모듈내 요소로는 함수, 클래스, 전역변수 모두 가능하다. |
| from 모듈명 import 요소, 요소 | from my_lib import test1, test2 test1() test2() | 여러 요소를 import할 때 from import를 반복해도 되지만 , 를 구분자로 등록한다. |
| from 모듈명 import * | from my_lib import * test1() test2() test3() | * 를 from 에 사용하면 모듈내 모든 요소를 import 한다. |

import

- import 한 요소에 별칭 주기
 - import 한 모듈이나 함수등의 이름이 길 경우 별칭을 주어 편하게 사용 할 수 있다.

| 구문 | 예 | 비고 |
|--------------------------|--|------------------------|
| import 모듈명 as 별칭 | import my_lib as ml ml.test1() | my_lib 모듈을 별칭 ml로 사용 |
| from 모듈명 import 요소 as 별칭 | from my_lib import test1 as t t() | test1 함수의 이름을 별칭 t로 사용 |

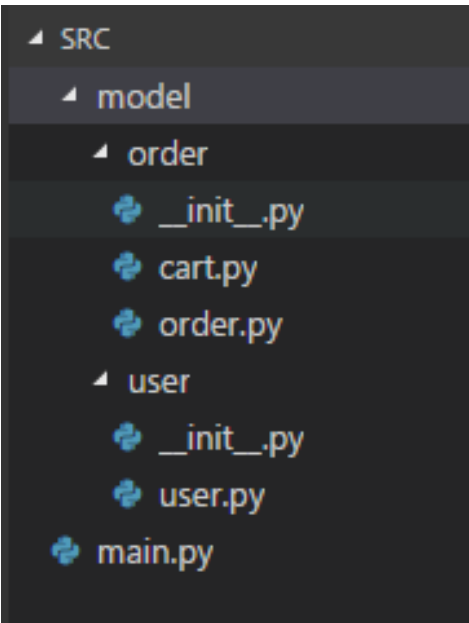
메인 모듈(Main Module)과 하위 모듈(Sub Module)

- 메인 모듈
 - 현재 실행하고 있는 모듈
- 하위 모듈
 - 메인 모듈에서 import 한 모듈
- `__name__` 내장 전역변수
 - 실행 중이 모듈명을 저장하는 내장 전역변수
 - 메인 모듈은 `'__main__'` 을 하위 모듈은 모듈명(파일명) 을 가진다.
- 모듈이 **메인 모듈로 시작하는지** 여부 확인 구문
 - 실행구문을 다음 구문의 if문 안에 넣지 않으면 서브 모듈로 실행시에도 실행된다.

```
if __name__ == '__main__':  
    실행 구문
```

패키지 (Package)

- 모듈들을 모아 놓은 디렉토리
 - 모듈의 수가 많아질 경우 역할이나 의미에 따라 디렉토리를 만들어 관리한다.
- `__init__.py`
 - 디렉토리가 패키지가 되기 위해서는 `__init__.py` 파일이 그 디렉토리에 있어야 한다.
 - 파이썬 3.3 부터는 없어도 패키지로 인식된다.
 - `__all__` 속성을 이용해 외부에서 `import *` 했을 때 임포트 될 모듈들을 선언 할 수 있다.



PYTHONPATH = SRC 인경우

| | |
|----------------------------|---|
| cart.py 임포트 하기 | <code>from model.order import cart</code> |
| order 패키지내 모든 모듈 임포트 | <code>from model.order import *</code> |
| user.py의 add_user() 함수 임포트 | <code>from model.user.user import add_user</code> |

site-packages 디렉토리

- 추가 패키지를 넣기 위해 제공되는 디렉토리
- 이 경로에 추가된 패키지는 환경설정 없이 import해서 사용할 수 있다.
- import 시 탐색되는 디렉토리 확인

```
import sys

for path in sys.path:
    print(path)
```

```
n\Anaconda3\python36.zip
n\Anaconda3\DLLs
n\Anaconda3\lib
n\Anaconda3
n\Anaconda3\lib\site-packages
n\Anaconda3\lib\site-packages\win32
n\Anaconda3\lib\site-packages\win32\lib
n\Anaconda3\lib\site-packages\Pythonwin
```



예외와 예외처리



- 1) 예외 처리개요
- 2) 예외 처리하기
- 3) 예외 구현 및 예외 발생시키기

예외 처리 개요

- 예외(Exception)란
 - 함수나 메소드가 처리 도중 다음 명령문을 실행하지 못하는 상황
- 예외 처리란
 - 발생한 예외를 해결하여 프로그램을 정상화 시키는 것

친구를 만난다()

1. 약속시간 1시간 전에 집에서 나온다.
2. 버스 정류장에 도착한다.
3. XXX번 버스를 타고 XX 정류장에 내린다.
4. 약속장소를 찾아 간다.
5. 친구를 만난다.

예외 상황

- 3에서 버스가 오지 않는 경우 (예외)
처리 : 택시를 타고 간다. (예외 처리)
- 4에서 약속장소가 쉬는 날인 경우 (예외)
처리 : 친구에게 전화해서 장소를 변경 (예외 처리)

예외 (Exception) 이란

■ 예외의 종류

- 파이썬 문법이나 구문 규칙을 어겨서 때문에 발생하는 오류 (System Exception)
 - 대부분 코드를 수정해야하는 오류
- 프로그램 업무 규칙상 발생하는 오류 (Application Exception)
 - 프로그램이 정한 업무규칙을 어기는 상황에서 발생하는 오류.
 - 코드의 수정이 아니라 처리를 해서 정상화 해야 한다.

■ 파이썬 오류 메시지

| 파이썬 shell | jupyter notebook |
|---|---|
| <pre>>>> 10/0 Traceback (most recent call last): File "<stdin>", line 1, in <module> ZeroDivisionError: division by zero</pre> | <pre>10/0 ----- ZeroDivisionError Traceback (most recent call last) <ipython-input-1-e574edb36883> in <module>() ----> 1 10/0 ZeroDivisionError: division by zero</pre> |

예외 (Exception) 이란

- 파이썬 실행 시 발생하는 주요 예외
 - 아래 예외는 코드를 대부분 수정해야 하는 오류들이다.
 - 예외 이름은 보통 Error로 끝난다.

| 예외이름 | 설명 |
|--------------------|-------------------------------|
| SyntaxError | 파이썬 문법에 어긋난 코드 작성시 발생한다. |
| NameError | 정의되지 않은 변수나 함수를 호출 한 경우 발생한다. |
| TypeError | 잘못된 타입의 값을 전달할 경우 발생한다. |
| ValueError | 타입은 맞는데 값이 잘못된 경우 발생한다. |
| IndexError | 없는 index로 리스트나 튜플 값 조회 시 발생 |
| KeyError | 없는 key로 딕셔너리의 값 조회 시 발생 |

예외 처리하기

▪ try, except 구문

```
try :  
    예외 발생 가능성 있는 코드블록 작성  
  
except :  
    예외가 발생했을 경우 실행될 코드 블록 작성  
    => 예외를 처리하는 코드가 여기 들어간다.
```

```
def divide(num1, num2):  
    try:  
        return num1 / num2  
    except:  
        print("나눗셈 도중 예외발생")
```

div(10, 0) # div 호출

▪ 특정한 예외만 처리하기

```
try :  
    예외 발생 가능성 있는 코드블록 작성  
  
except 예외이름:  
    예외가 발생했을 경우 실행될 코드 블록 작성  
    => 예외를 처리하는 코드가 여기 들어간다.
```

```
def divide(num1, num2):  
    try:  
        return num1 / num2  
    except ZeroDivisionError:  
        print("나눗셈 도중 예외발생")
```

div(10, 0) # div 호출

예외 처리하기

■ 여러 오류를 따로 따로 처리하기

```
try :  
    예외 발생 가능성 있는 코드블록 작성  
  
except 발생예외이름1:  
    처리 코드 블록  
except 발생예외이름2:  
    처리 코드 블록  
except:  
    처리 코드 블록
```

- 마지막 except:는 상위에서 처리한 것 이외의 예외가 발생시 실행될 처리코드를 넣는다.
- except: 코드블록은 마지막으로 와야 한다.

■ else

- 예외가 발생하지 않았을 경우 실행할 코드 블록을 작성한다.
- **except** 다음에 와야 한다.

```
try :  
    예외 발생 가능성 있는 코드블록 작성  
except :  
    예외 처리 코드  
else:  
    try에서 예외가 발생하지 않았을 경우 실행할 코드블록
```

예외 처리하기

▪ finally

- 예외 발생여부, 처리 여부와 관계없이 무조건 실행되는 코드블록
- finally 는 except 와 else 보다 먼저 올 수 없다.

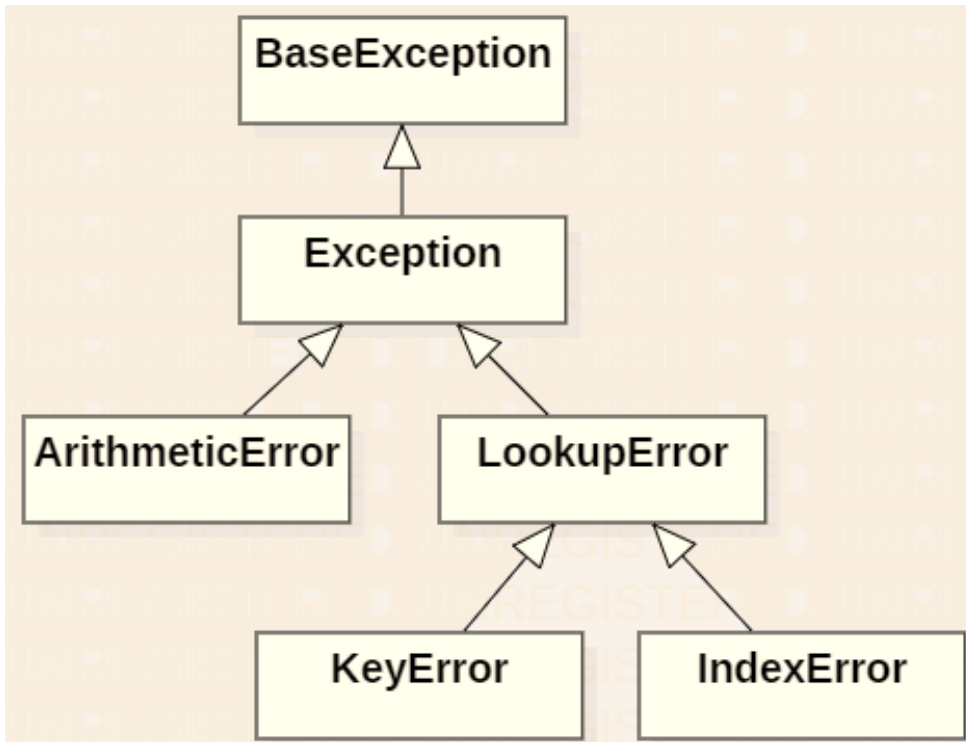
```
try :  
    예외 발생 가능성 있는 코드블록 작성  
  
except :  
    처리 코드 블록  
  
else:  
    예외가 발생하지 않을 경우 실행될 코드블록  
  
finally:  
    무조건 실행되는 코드블록
```

```
try :  
    예외 발생 가능성 있는 코드블록 작성  
  
finally:  
    무조건 실행되는 코드블록
```

```
try :  
    예외 발생 가능성 있는 코드블록 작성  
except:  
    처리 코드 블록  
  
finally:  
    무조건 실행되는 코드블록
```

예외 상속 구조

- 파이썬은 예외를 모두 클래스로 정의해 사용한다.
- 파이썬 예외의 상속 구조



- 프로그램 로직 흐름상 예외가 발생해야 하는 경우 예외 클래스를 정의할 수 있다.
- 사용자 정의 예외는 **Exception**을 상속받아서 만든다.

예외 발생시키기

- **raise** 예외객체
 - 예외를 발생시킨다.
 - 대부분 if 문에 작성한다.
 - 예외가 발생해야 하는 조건이 True일 경우

예외 클래스 구현 및 발생시키기

- 예외클래스 작성

- Exception을 상속받아 만든다.

```
class MyError(Exception):  
    #구현
```

- 예외 발생시키기

```
if something_problem == True:  
    raise MyError()
```



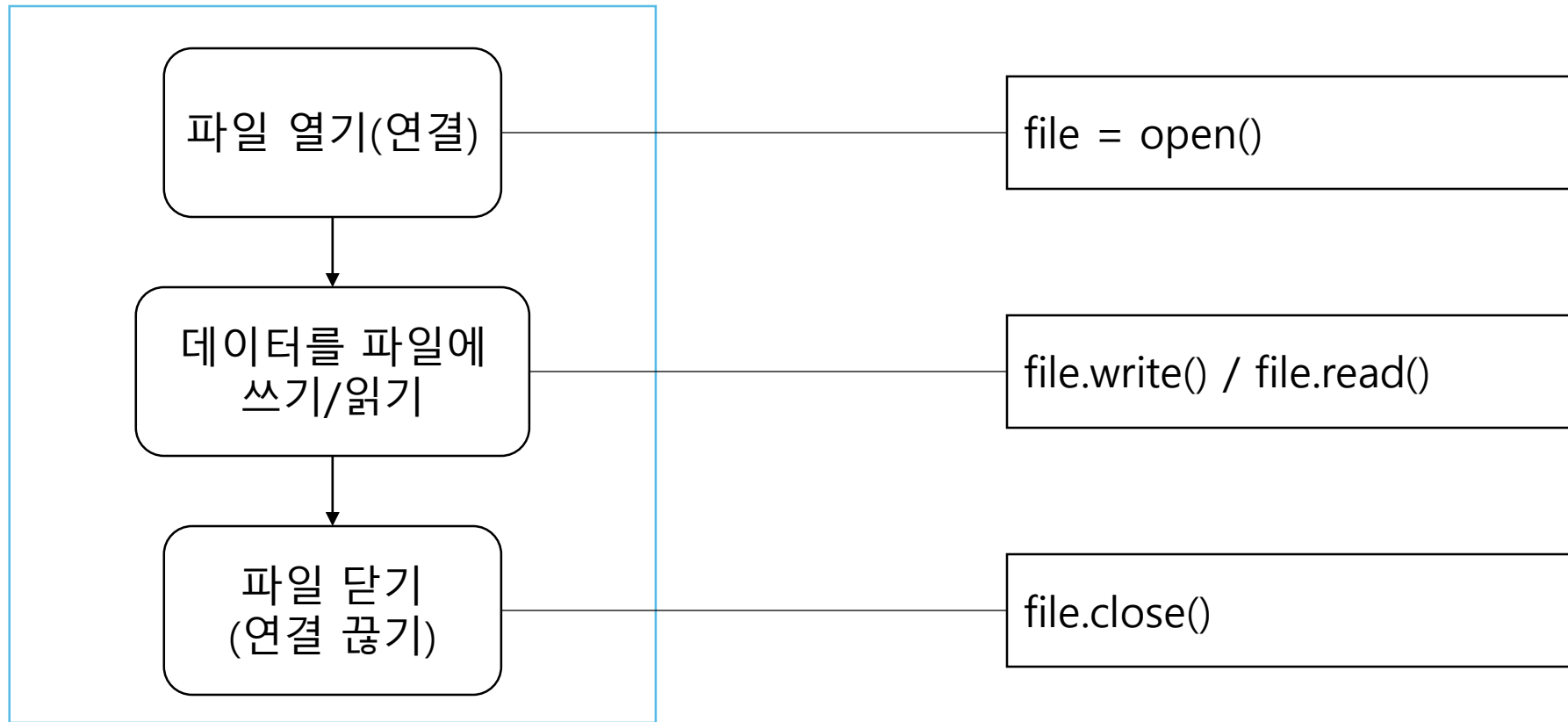
파일 입출력



- 1) 입출력 개요
- 2) Text 파일 입출력
- 3) with 문

개요

- 프로그램이 외부 파일에 데이터를 쓰거나 읽는 작업을 입출력(IO) 라고 한다.
- 패턴



파일 연결

- open() 함수 사용
 - 연결된 파일과 입출력 메소드를 제공하는 객체를 리턴 (TextIOWrapper)
- open() 함수 주요 매개변수
 - file : 연결할 파일 경로
 - mode : 열기 모드
 - 모드는 조합할 수 있다. (ex: wb)
 - encoding : 텍스트 파일일 경우 인코딩 방식

| 모드문자 | 의미 |
|------|--|
| 'r' | 읽기모드 (기본값) |
| 'w' | 쓰기모드. 존재하는 파일과 연결시 그 파일을 지운다. (새로 쓰기) |
| 'x' | 배타적 생성모드. 파일이 존재하면 IOError 발생 |
| 'a' | 이어쓰기. 존재하는 파일과 연결 시 그 파일의 기존 내용은 두고 이어 쓴다. |
| 'b' | 바이너리 모드 |
| 't' | 텍스트 모드(기본값) |
| '+' | 읽기/쓰기 모드 |

텍스트 파일 입출력

- 쓰기(출력) 메소드

- write(문자열)
- writelines(문자열을 가진 컬렉션)
 - 리스트, 튜플, 집합이 원소로 가진 **문자열**들을 한번에 출력한다.
 - 원소에 문자열 이외의 타입의 값이 있을 경우 TypeError 발생

텍스트 파일 입출력

- 읽기(입력) 메소드

1. read() : 문자열
 - 한번에 모두 읽어 들인다.
2. readline() : 문자열
 - 한 줄만 읽는다.
 - 만약 읽은 라인이 없으면 None을 리턴한다.
3. readlines() : 리스트
 - 한번에 다 읽은 뒤 각각의 라인을 리스트에 원소로 담아 반환한다.
4. for문을 이용한 라인단위 읽기

```
file = open('a.txt', 'r')  
  
for line in file:  
    print(line)
```

with 문

- 파일과 입출력 작업이 다 끝나면 반드시 **연결을 닫아야 한다**. 매번 닫는 작업을 하는 것이 번거롭고 실수로 안 닫을 경우 문제가 생길 수있다. 그래서 작업이 끝난 뒤 자동으로 닫도록 해주는 구문이 **with**문이다.

```
with open(파일명, ..) as 변수:  
    변수를 이용한 입출력 작업
```

with 블록이 끝나면 자동으로 파일과의 연결을 닫는다. (close() 할 필요 없다.)

```
with open('a.txt', 'w') as file:  
    file.write("hello")
```

```
with open('b.txt', 'r') as file:  
    for line in file:  
        print(line)
```

pickle 로 객체저장 및 읽기

- 파이썬 객체 자체를 binary 파일로 저장하는 파이썬 모듈
- open() 시 binary 모드로 연다.
 - 파일 확장자는 보통 pickle을 준다.
 - open('data.pickle', 'wb'), open('data.pickle', 'rb')
- 저장
 - pickle.dump(저장할객체, fw)
- 읽기
 - pickle.load(fi)



내장함수 (Built-In Function)



- 1) 타입 변환 함수들
- 2) Iterable 관련 함수들
- 3) 수학 연산관련 함수들
- 4) 기타

내장함수(Built-In Function)란

- 특정 모듈을 import 하지 않고 사용할 수 있는 함수들.
- <https://docs.python.org/ko/3/library/functions.html>

주요 함수 - 타입 변환 함수

- `type(object)`
 - `object`의 타입 조회.
- `int(x)`
 - 숫자 또는 문자열 `x`를 정수로 변환
- `float(x)`
 - 숫자 또는 문자열 `x`를 실수로 변환
- `bool(x)`
 - `x`를 논리값 `True`, `False`로 변환해 반환
- `str(x)`
 - `x`를 문자열로 변환

주요 함수 – iterable 관련 함수

- `list(iterable)` : iterable를 리스트로 반환
- `tuple(iterable)` : iterable를 튜플로 반환
- `set(iterable)` : iterable를 Set으로 반환
- `dict(**kwarg)` : **kwarg에 전달된 이름=값 을 이용해 사전(Dictionary) 객체를 생성
- `len(s)` : 자료구조객체들, 문자열등의 항목 수를 조회해 반환.
- `min(iterable), max(iterable)` : iterable의 원소중 최소값/최대값을 반환
- `all(iterable)` iterable의 모든 값이 True거나 비어 있으면 True반환
- `any(iterable)`
 - iterable의 값들 중 하나라도 True면 True반환. 비어 있으면 False 반환

`list()`, `tuple()`, `set()`에 사전(dictionary)를 전달하면 Key값만 각각의 자료구조에 넣어 반환한다.

주요 함수 – iterable 관련 함수

- `range([start,] stop [,step])`
 - 입력받은 범위의 숫자를 제공하는 iterable객체를 반환. stop만 지정 시 0부터 시작.
- `enumerate(iterable, start = 0)`
 - iterable의 원소와 그 원소의 index 반환하는 enumerate객체 반환. start는 시작 index 값.
- `zip(*iterable)` : 동일한 크기의 iterable들을 받아 같은 index의 값들을 묶어 리턴.
- `map(함수, iterable)`
 - iterable의 각 원소를 함수의 매개변수로 전달하여 그 반환 값들을 담은 iterable 객체를 반환.
- `filter(함수, iterable)`
 - iterable의 각 원소를 함수의 매개변수로 전달하여 True인 것만 걸러낸다.
- `sorted(iterable, reverse=False)`
 - iterable의 요소들을 오름차순 정렬한 리스트(List)를 반환, reverse를 True 설정 시 내림차순 정렬

주요 함수 - 수학연산관련 함수

- `abs(x)` : x 의 절대값을 반환
- `divmod(a, b)` : a 를 b 로 나눈 몫과 나머지를 튜플로 반환. ($a // b, a \% b$)
- `pow(x, y)` : x 의 y 제곱의 결과를 반환
- `round(number [, ndigits])`
 - 반올림한 정수를 반환.
 - `ndigits`는 반올림 정밀도 지정.
 - `ndigits`가 양수이면 소수점 이하 자릿수, 양수이면 정수부.

주요 함수 - 기타

- `dir([object])`
 - `object`에 있는 속성, 메소드들의 이름을 리스트에 담아 리턴 한다
 - `object` 생략하면 현재 `scope`에서 사용할 수 있는 속성, 함수등의 이름을 리턴 한다.



Iterator(반복자), Decorator(장식자)



- 1) Iterator
- 2) Local 함수와 클로저(Closure)
- 3) 함수 Decorator
- 4) 클래스 Decorator

Iterator(반복자)

- Iterable
 - for in 문을 이용해 요소를 순서대로 조회할 수 있는 것을 Iterable(반복할 수 있는) 타입이라 함
 - 리스트, 문자열, range 가 Iterable 이다.
- Iterator(반복자)
 - Iterable한 타입의 객체를 Iterator(반복자)라고 한다.
 - for in 문을 이용해 요소들을 조회할 수 있는 기능을 제공하는 객체.

Iterator(반복자)

- for in 문이 반복자 객체를 순환 조회하는 과정
 1. 반복 조회할 객체의 **__iter__()** 를 호출 하여 Iterator 객체를 Iterator를 구한다.
 - 일반적으로 반복자는 그 객체 자체를 리턴 한다.
 2. 매 반복마다 Iterator의 **__next__()** 를 호출하여 다음 원소를 조회하고 다음 위치로 이동.
 - 모든 요소를 다 읽을 때 까지 반복
 3. 모든 요소를 다 읽으면 Iterator는 StopIteration 예외를 발생시키고 for in문은 이 예외를 받으면 반복을 종료.

```
it = nums.__iter__()
while True:
    try:
        num = it.__next__()
        print(num)
    except StopIteration:
        break
```

Iterable 클래스 구현

- Iterable 타입 클래스를 구현
 - Iterator를 제공하는 클래스를 말한다.
- 구현
 - `__iter__()` 메소드 구현
 - `__next__()` 메소드를 가지고 원소를 제공하는 객체를 반환한다.
 - 일반적으로 자신(self)을 반환한다.
 - `__next__()` 메소드 구현
 - 위치(index)를 관리하고 원소를 반환
 - 모든 원소를 반환했을 때는 `StopIteration` 예외를 발생시킨다.

```
class MyIterator:
    def __init__(self, str):
        self.index = 0
        self.str = str

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.str):
            raise StopIteration
        s = self.str[self.index]
        self.index += 1
        return s

for s in MyIterator("ABCDEFGH"):
    print(s)
```

Generator(제너레이터)

- Iterator 의 역할을 하는 함수.
 - Iterator는 구현 시 클래스를 만들고 생성자, `__iter__()`, `__next__()`를 구현해야 한다. 번거롭다.
 - Generator 함수로 구하는 Iterator로 간단하게 구현할 수 있다.
- yield 명령어
 - `return` 과 같이 값을 반환하는데 그 반환한 변수의 마지막 값과 상태를 관리한다.
 - Generator는 일반함수의 형태로 구현하되 반환을 `yield`를 이용해 원소를 반환하도록 한다.

```
def mygenerator(str):  
    for idx in range(0, len(str)):  
        yield str[idx]
```

`yield`가 `str`의 마지막 값과 그 다음 값에 대한 상태를 관리한다.

Generator(제너레이터) – Generator Comprehension

- Generator Comprehension

- 리스트 Comprehension 동일한 구문인데 괄호를 () 사용한다.
- 리스트 Comprehension 은 미리 리스트를 만들어 놓는다.
Generator Comprehension 은 반복 가능한 객체만 만들고 실제 원소에 대한 요청이 왔을 때 값을 생성한다.
- 메모리 효율이 리스트 Comprehension 보다 좋다.

```
list = (num + 1 for num in range(10, 100))
```

```
for n in (num + 1 for num in range(10, 100)):  
    print(n)
```


Local(지역) 함수

- 파이썬의 함수는 **일급 시민 객체(First Class Citizen Object)** 이다.
 - 변수에 저장할 수 있고, 매개변수에 전달할 수 있고 반환할 수 있는 객체
 - 함수형 언어가 가지는 특징
- Local 함수
 - 함수 내에 정의 한 함수
 - 함수 내부에서만 호출 할 수 있다. 단 함수 자체를 반환하면 외부함수를 호출한 곳에서 사용가능.


```
def outer() :  
    num = 10  
  
    def inner(num2) :  
        return num + num2  
  
    return inner(20) #호출 결과 리턴  
  
print(outer())
```

Local(지역) 함수

- 클로저

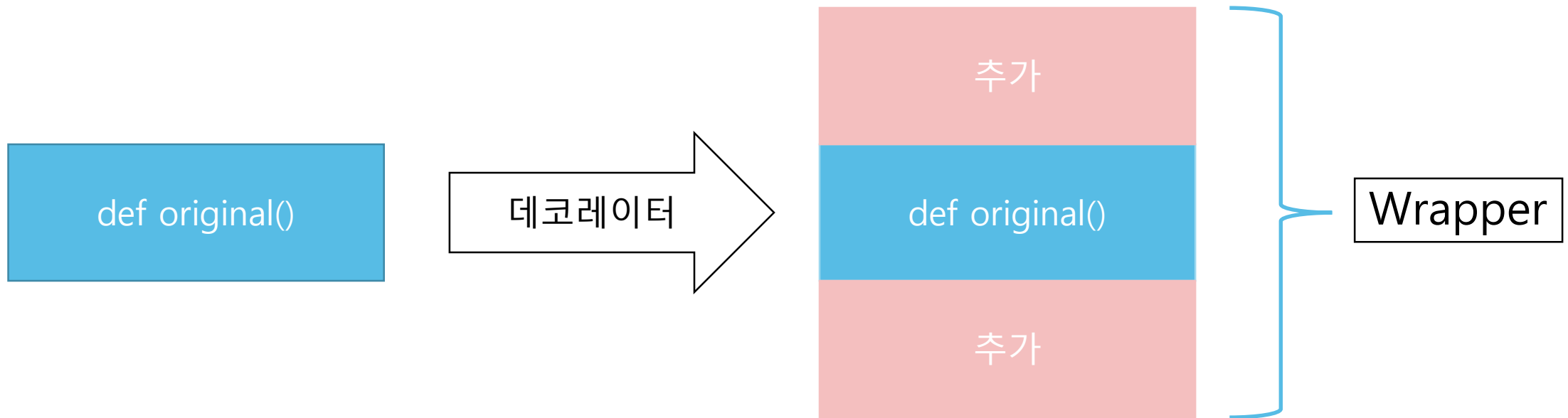
- 지역 함수가 사용하는 외부 함수의 지역변수는 외부함수가 종료 되어도 지역함수가 종료 될 때까지 메모리에 유지 되도록 하는 구조.

```
def outer() :  
    num = 10  
    def inner(num2) :  
        return num + num2  
    return inner #함수자체 리턴  
  
fun = outer()  
fun()
```



함수 데코레이터(Decorator)

- 기존 함수를 매개변수로 새롭게 변형된 함수로 바꾸어 반환하는 함수
 - 기존 함수코드를 고치지 않고 기능을 추가하는 것이 목적



함수 데코레이터(Decorator)

```
def decorator(func):  
    def wrapper():  
        print(func.__name__+"함수 호출 전")  
        func()  
        print(func.__name__+"함수 호출 후")  
    return wrapper
```

@decorator

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

- @decorator의 의미
 - hello_word = decorator(hello_world)

함수 데코레이터(Decorator)

```
def decorator(func):  
    def wrapper():  
        print(func.__name__+"함수 호출 전")  
        func()  
        print(func.__name__+"함수 호출 후")  
    return wrapper  
  
@decorator  
def hello_world():  
    print("안녕하세요")
```

```
# 호출  
hello_word()
```

- @decorator의 의미
 - `hello_word = decorator(hello_world)`

함수 데코레이터(Decorator)

- 매개변수가 있는 경우
 - wrapper 함수에 원본함수와 동일한 개수의 매개변수를 선언한다.
- 반환값이 있는 경우
 - wrapper 에서 반환한 값이 호출 한 곳에 반환된다.

```
def decorator(func):  
    def wrapper(name):  
        print(func.__name__+"함수 호출 전")  
        value = func()  
        print(func.__name__+"함수 호출 후")  
        return value  
    return wrapper
```

```
@decorator  
def hello_world(name):  
    return "{ } 님 안녕하세요".format(name)
```

정규표현식

Regular Expression, regexp

- 1) 정규표현식 개요
- 2) 정규 표현식 메타 문자
- 3) 파이썬 정규표현식 라이브러리 re
 - 코딩패턴
 - re의 함수
 - Grouping
 - Greedy와 Non-Greedy

정규표현식 개요

정규 표현식이란

- 텍스트에서 특정한 형태나 규칙을 가지는 문자열을 찾기 위해 그 형태나 규칙을 정의하는 것.
- 파이썬 뿐만 아니라 문자열을 다루는 모든 곳에서 사용된다.
- 정규식, **Regex**, **Regexp**이라고도 한다.

기본 용어

- 패턴

- 정규 표현식이라고 한다.
- 문장내에서 찾기 위한 문구의 형태에 대한 표현식.

- 메타문자

- 패턴을 기술하기 위해 사용되는 특별한 의미를 가지는 문자
- 예) a^* : a 가 0회 이상 반복을 의미. a , aa , $aaaa$

- 정규문자(리터럴)

- 표현식이 값 자체를 의미하는 것
- 예) a 는 a 자체를 의미한다.

정규표현식 메타문자

문자 클래스 : []

- [] 사이의 문자들과 매칭
 - [abc] : a, b, c 중 하나의 문자와 매치
- ` ` 를 이용해 범위로 설정할 수 있다.
 - [a-z] : 알파벳소문자중 하나의 문자와 매치
 - [a-zA-Z0-9] : 알파벳대소문자와 숫자 중 하나의 문자와 매치
- [^ 패턴] : ^ 으로 시작하는 경우 반대의 의미
 - [^abc] : a, b, c를 제외한 나머지 문자들 중 하나와 매치.
 - [^a-z] : 알파벳 소문자를 제외한 나머지 문자들 중 하나와 매치

미리 정의된 문자 클래스

- 자주 사용되는 문자 클래스를 미리 정의된 별도 표기법으로 제공

| 표기법 | 설명 |
|-----|---|
| \d | 숫자와 매치. [0-9]와 동일 |
| \D | \d의 반대. 숫자가 아닌 문자와 매치. [^0-9]와 동일 |
| \w | 문자와 숫자, _(underscore)와 매치. [a-zA-Z0-9_]와 동일 |
| \W | \w의 반대. 문자와 숫자와 _가 아닌 문자와 매치. [^a-zA-Z0-9_]와 동일 |
| \s | 공백문자와 매치. tab, 줄바꿈, 공백문자와 일치 |
| \S | \s와 반대. 공백을 제외한 문자열과 매치. |
| \b | 단어 경계(word boundary) 표시. |
| \B | \b의 반대. 단어 경계로 구분된 단어가 아닌 경우 |

글자수와 관련된 메타문자

| 표기법 | 설명 |
|--------|---|
| . | $\forall n$ -줄바꿈을 제외한 한개의 모든 문자. (a.b) |
| * | 앞의 문자(패턴)과 일치하는 문자가 0개 이상인 경우. (a*b) |
| + | 앞의 문자(패턴)과 일치하는 문자가 1개이상인 경우. (a+b) |
| ? | 앞의 문자(패턴)과 일치하는 문자가 한개 있거나 없는 경우. (a?b) |
| {m} | 앞의 문자(패턴)가 m개. (a{3}b) |
| {m,} | 앞의 문자(패턴)이 m개 이상. (a{3,}b) |
| {m, n} | 앞의 문자(패턴)이 m개이상 n개 이하. (a{2,5}b) |

기타

| 표기법 | 설명 |
|-----|---|
| ^ | 문자열의 시작. (^abc). 문자 클래스([])의 ^와는 의미가 다르다 |
| \$ | 문자열의 끝 (abc\$) |
| | 둘 중 하나. (010 011 016 019) |
| () | 패턴내 하위그룹을 만들때 사용 |

re 모듈

- 파이썬 정규표현식 지원 내장 모듈

프로그램 작성 패턴

- 모듈 import
 - import re
- 패턴객체 생성
 - 패턴 컴파일
 - 패턴을 가지고 있는 객체
- 텍스트에서 패턴 문자열 검색 또는 변경 작업

```
: 1 import re                #1. re 패키지 import
  2 p = re.compile('\d+')    # 2. 패턴 생성
  3 m = p.search('aaaaa99aa') # 3. 검색
  4 if m:
  5     print("숫자가 포함되어있습니다.", m.group())
  6 else:
  7     print("숫자가 포함되지 않습니다.")
```

executed in 5ms, finished 19:20:23 2019-05-15

함수 - 검색 함수

- `match(대상 문자열 [, pos=0])`
 - 대상 문자열의 시작 부터 정규식과 일치하는 것이 있는지 조회
 - `pos` : 시작 index 지정
 - 반환값
 - `Match` 객체: 일치하는 문자열이 있는 경우, 일치하는 문자열이 없는 경우 `None` 반환
- `search(대상문자열 [, pos=0])`
 - 대상문자열 전체 안에서 정규식과 일치하는 것이 있는지 조회
 - `pos`: 찾기 시작하는 index 지정
 - 반환값
 - `Match` 객체: 일치하는 문자열이 있는 경우, 일치하는 문자열이 없는 경우 `None` 반환

함수 - 검색 함수

- findall(대상문자열)
 - 대상문자열에서 정규식과 매칭되는 문자열들을 리스트로 반환
 - 반환값
 - 리스트(List) : 일치하는 문자열들을 가진 리스트를 반환
 - 일치하는 문자열이 없는 경우 빈 리스트 반환
- finditer(대상문자열)
 - find() 함수와 동일한데 반환값이 Match를 담고 있는 Iterator 객체 이다.

함수 - 문자열 변경

- `sub(바꿀문자열, 대상문자열 [, count=양수])`
 - 대상문자열에서 패턴과 일치하는 것을 바꿀문자열로 변경한다.
 - `count`: 변경할 개수를 지정. 기본: 매칭되는 문자열은 다 변경
 - 반환값: 변경된 문자열
- `subn(바꿀문자열, 대상문자열 [, count=양수])`
 - `sub()`와 동일한 역할.
 - 반환값 : (변경된 문자열, 변경된문자열개수) 를 tuple로 반환

compile() 함수를 이용해 패턴 객체 생성

- 정규식을 가지는 Pattern객체를 생성
- 구문
 - compile(정규표현식, [옵션])
- 옵션
 - re.DOTALL, re.S : 메타문자 '.'사용시 \n이 매칭 문자에 포함되도록 한다.
 - re.IGNORECASE, re.I : 매칭시 대소문자를 상관하지 않는다.
 - re.MULTILINE, re.M
 - 대상문자열이 여러줄일 경우 문장의 시작(^)과 끝(\$)을 라인별로 적용한다.
 - 전체 문장의 시작과 끝은 \nA와 \nZ를 사용한다.

그룹핑 (Grouping)

- 패턴 내에서 재 사용가능한 하위패턴을 만든다.
- 구문 : () 로 묶어 준다.
 - (\d{2,3})-(\d{3,4})-(\d{4})

패턴내에서 특정 패턴의 문장만 조회

```
1 # 전화번호에서 국번만 조회하려는 경우
2 import re
3 p = re.compile(r'(\d{2,3})-(\d{3,4})-(\d{4})')
4 m = p.search('010-1111-2345')
5
6 print(m.group(0)) # 매칭된 전체 문자열
7 print(m.group(1)) # 1번 그룹
8 print(m.group(2)) # 2번 그룹
9 print(m.group(3)) # 3번 그룹
```

executed in 6ms, finished 15:23:30 2019-06-01

010-1111-2345

010

1111

2345

패턴 내에서 group 참조

- \번호
- 지정한 '번호' 번째 패턴으로 매칭된 문자열과 같은 문자열을 의미

```
1 import re
2 p = re.compile(r'(010|011|016|019)-(\d{4})-\2')
3 print(p.match('010-1111-2222'))
4 print(p.match('011-3333-3333'))
5 print(p.match('016-5555-5555'))
```

executed in 5ms, finished 10:08:13 2019-05-16

None

<re.Match object; span=(0, 13), match='011-3333-3333'>

<re.Match object; span=(0, 13), match='016-5555-5555'>

그룹핑 - 확장표기법

- 그룹에 기능 추가하는 표기법
- 기본 구문 : (?extension ...)
- 그룹에 이름 주기
 - 구문
 - (?P<이름>...) : 이름 주기
 - (?P=이름) : 이름으로 하위그룹 참조

- 상황별 지정한 패턴을 참조하기
 - 패턴 내에서 참조
 - (?P=이름)
 - \W그룹번호
 - Match객체에서 참조
 - m.group("이름")
 - m.group(번호)
 - sub() 함수에서 참조
 - \Wg<이름>
 - \Wg<번호>

```
1 import re
2 p = re.compile(r'(?P<name>\w+)-(?P=name)-\1')
3 print(p.match('홍길동-이순신-홍길동'))
4 m = p.match('홍길동-홍길동-홍길동')
5 print(m.group('name'), m.group(1))
```

executed in 4ms, finished 10:46:33 2019-05-16

None

홍길동 홍길동