

# Personal Theorem


---

## 1. 둘러보기

- 사칙연산
- 변수에 숫자 대입하고 계산하기
- 변수에 문자 대입하고 출력하기
- 조건문 if
- 반복문 for
- 반복문 while
- 함수

## 2. 자료형

- 숫자형
  - 8진수(0o), 16진수(0x)
  - \*\*
  - //
- 문자열 자료형 [ ("x"), 수정불가능(immutable) ]
  - 이스케이프 코드
    - \n : 개행
    - \t : 수평 탭
    - \\ : 문자 "\"
    - ' : 단일 인용부호(')
    - " : 이중 인용부호(")
  - '''
  - """
  - 문자열 곱하기 응용
  - 문자열 인덱싱(ex. a[0])
  - 문자열 슬라이싱(ex. a[1:3], a[1:-7])
  - 문자열 포매팅
    - %
      - %s : 문자열(String)
      - %c : 문자 1개(Character)
      - %d : 정수(Integer)
      - %f : 부동소수(Floating-point)
      - %o : 8진수
      - %x : 16진수
      - %% : Literal %(문자 % 자체)
    - format 함수(ex. .format(num))
  - 문자열 관련 함수들
    - count(x)
    - find(x)
    - index(x) (find보다 이걸 더 많이 사용하는 듯..?)
    - join(x)
    - upper()
    - lower()
    - lstrip()

- `rstrip()`
  - `strip()`
  - `replace(x, y)`
  - `split()` or `split(x)`
- 리스트 자료형 [ `([x])`, 수정가능(`mutable`) ]
  - 인덱싱
  - 슬라이싱
  - 리스트(`str` 변경) + 문자열
  - 리스트에서 연속된 범위의 값 수정
    - `a[1:2] = ['a', 'b', 'c']`과 `a[1] = ['a', 'b', 'c']`
  - 리스트 삭제
    - `[]`을 이용
    - `del` 함수
  - 리스트 관련 함수들
    - `append(x)`
      - `append([x,y])`로 하면 `[x,y]`가 하나의 요소가 되어 들어간다.
    - `sort()`
    - `reverse()`
    - `index(x)`
    - `insert(x, y)`
    - `remove(x)`
    - `pop()` or `pop(x)`
    - `count(x)`
    - `extend([...])`
    - **리스트의 요소를 제거하는 3가지 방법 (그때그때 사용법이 다르다. 그래도 웬만하면 `del`을 많이 사용하는 듯하다..!)**
      - `remove` (`a.remove(x)` - `x`는 `a`리스트의 요소값)
      - `pop` (`a.pop(x)` - `x`는 `a`리스트의 인덱스)
      -  - `x`는 `a`리스트의 인덱스)
    - **함수 쓸 땐 대부분 `()`, `print`에서 변수에 있는 요소를 뺏을 때 대부분 `[]`**
- 튜플 자료형 [ `((x))`, 수정불가능(`immutable`) ]
  - 인덱싱
  - 슬라이싱
  - 더하기
  - 곱하기
- 딕셔너리 자료형 [ `{ a:b }`, 수정가능(`mutable`) ]
  - `Key`에는 변하지 않는 값을 사용하고, `Value`에는 변하는 값과 변하지 않는 값 모두 사용할 수 있다.
  - 인덱싱, 슬라이싱 불가능
  - 쌍 추가
  - 요소 삭제 `del` (ex. `del a[1]`, **여기서 1은 key값**)
  - 딕셔너리 만들 때 주의사항
    - 동일한 `key` 추가
    - `Key`에 리스트는 사용불가능
  - 딕셔너리 관련 함수들
    - `keys()`
    - `values()`

- item()
  - clear()
  - get()
    - print(a[x])
    - print(a.get(x))
      - 위와 동일한 결과이다.
      - 하지만 a['nokey']처럼 a 딕셔너리에 없는 키로 값을 가져오려고 할 경우 **a['nokey']는 Key 오류를 발생시키고 a.get('nokey')는 None을 리턴한다는 차이가 있다.**
    - a.get(x, y)
      - 딕셔너리 안에 찾으려는 key 값이 없을 경우 미리 정해 둔 디폴트 값을 대신 가져 오게 하고 싶을 때에는 get(x, '디폴트 값')을 사용하면 편리하다.
  - x in a
    - bool 형으로 나온다.
- 집합 자료형 [ set([x]) ]
  - set에는 다음과 같은 2가지 큰 특징이 있다.
    - 중복을 허용하지 않는다.
    - 순서가 없다.(Unordered) -> **인덱싱, 슬라이싱 불가능 (출력 형태가 {x}로서 중괄호 형태로 나오기 때문에)**
      - 만약 set 자료형에 저장된 값을 인덱싱으로 접근하려면 **리스트나 튜플로 변환한 후 해야 한다.**
  - 교집합(& or intersection(x))
  - 합집합(| or union(x))
  - 차집합(- or difference(x))
  - 집합 자료형 관련 함수들
    - add(x)
    - update([x])
    - remove(x)
    - difference\_update([x])
- 불 자료형
  - 참 vs 거짓
    - 참
      - True
      - "x"
      - [x]
      - 1
    - 거짓
      - ""
      - [], {}, {}
      - 0
      - None
- 자료형의 값을 저장하는 공간, 변수
  - C 언어나 Java처럼 변수의 자료형을 함께 쓸 필요는 없다. 파이썬은 변수에 저장된 값을 스스로 판단하여 자료형을 알아낸다.
  - **변수가 생성되는 과정(메모리)**
    - print(id(a)) 하면 메모리 주소 확인

- 리스트를 변수에 넣고 복사
  - 왜 복사하면 변수들이 다 바뀌는지? (ex.  $a = b$ 로 하면 같은 메모리를 가리키기 때문에)
  - 그렇다면 다른 주소값을 가지도록 하려면?
    - `[:]`
      - `b = a[:]`
    - `copy`
      - `from copy import copy`
      - `b = copy(a)`
- 변수를 만드는 여러 가지 방법
  - `a, b = ('python', 'list')`
  - `(c, d) = 'python', 'list'`
  - `[e, f] = 'python', 'list'`
  - `a = b = 'python'`
- 간단한 변수 바꾸기
  - `a, b = b, a`

### 3. 제어문 (들어쓰기 조심!)

- if문
  - 조건문
    - 비교연산자
    - `and`, `or`, `not`
    - `x in s`, `x not in s`
      - `print(1 in [1,2,3]) -> True`
      - `print(1 not in [1,2,3]) -> False`
    - `elif`
    - 조건부 표현식
      - `if score >= 60: num = 0 else: num = 1 < 원래버전 >`
      - `num = 0 if score >= 60 else 1`
- while문
  - `break` 사용
  - `continue` 사용
  - 무한 루프
- for문
  - 기본적인 사용
    - `for i in a:`
  - 다양한 for문 사용
    - `for (first, last) in a:`
  - for문과 `continue`
  - `range` 함수
    - `range(10)`
      - 0부터 9까지의 범위
    - `range(1, 11)`
      - 1부터 10까지의 범위
    - `len(a)`
  - **리스트 안에 for문 포함하기**
    - (`a = [1,2,3,4]`와 `result = []`를 선언하고 난 뒤에) `for i in a: / result.append(i*3) < 원래버전 >`
    - `result = [num * 3 for num in a]`

- result = [num \* 3 for num in a if num % 2 == 0]
- result = [x\*y for x in range(2, 10) for y in range(1, 10)]

#### 4. 입력과 출력

##### ○ 함수

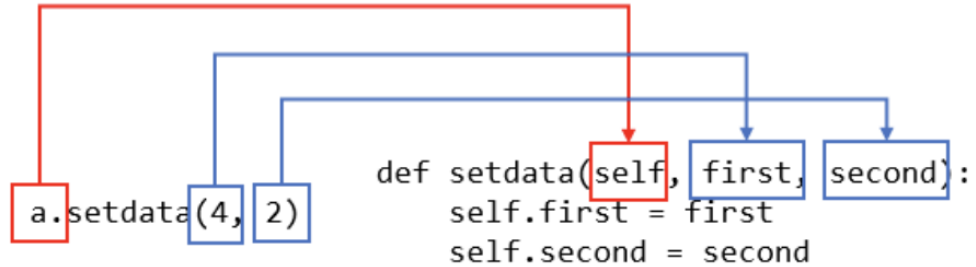
- 매개변수와 인수
  - 매개변수는 받는 값, 인수는 입력 값
- 입력값과 결과값에 따른 함수의 형태
  - 일반적인 함수 (return 있음)
    - a = sum(3,4) print(a)
    - print(sum(3,4))
  - 입력값이 없는 함수 (return 있음)
    - a = say() print(a)
    - print(say())
  - 결과값이 없는 함수 (return 없음)
    - sum(3,4)
    - a = sum(3,4) print(a) 하면 None
    - print(sum(3,4)) 하면 출력값 나오고 None
  - 입력값도 결과값도 없는 함수 (return 없음)
    - say()
    - a = say() print(a) 하면 None
- 입력값이 몇 개가 될지 모를 때
  - \*args (return 있음)
    - def sum\_many (\*args):
    - def sum\_mul (choice, \*args): 처럼 일반적인 매개변수와 같이 쓸 수도 있다.
  - \*\*kwargs (return 없음)
    - func 함수의 인수로 key=value 형태를 주었을 때 입력 값 전체가 kwargs라는 딕셔너리에 저장된다는 것을 알 수 있다. 즉, \* \* **kwargs**는 모든 key=value 형태의 입력인수가 저장되는 딕셔너리 변수이다.
    - def func(\*\*kwargs): / print(kwargs)
      - func(a=1) 을 입력하면 {'a':1} 출력
  - \*args + \*\*kwargs (return 없음)
    - def func(\*args, \*\*kwargs):
      - print(args)
      - print(kwargs)
      - func(1,2,3, name='foo', age=3) 을 입력하면 (1,2,3) {'name':'foo', 'age':3} 출력
- 함수의 결과값은 언제나 하나
  - return a+b, a\*b print(sum\_and\_mul(3,4)) 그러면 튜플형태로 나온다.
  - 만약에 따로 받고 싶다면?
    - sum, mul = sum\_and\_mul(3,4) print(sum, mul)
- 단독 return (함수 바로 빠져나가기)
- 매개변수 초깃값 미리 설정
  - def say\_myself(name, old, man=True):
  - **항상 맨 뒤쪽에 위치시키기!!**
- 함수 안에서 선언된 변수
  - return 이용

- 함수 사용 시, 마지막에 return a 로 입력한 뒤에, 따로 a = varstest(a)라고 해줘야한다
  - global 명령어 이용
    - global a 선언
  - lambda
    - def와 동일한 역할
    - sum(함수명) = lambda a, b(매개변수): a+b(return값)
    - myList = [lambda a,b:a+b, lambda a,b:a\*b] 이렇게도 가능
      - print(myList[0] (3,4)) or print(myList[1] (3,4)) 이런 식으로 사용 가능
- 사용자의 입력과 출력
  - input 사용
  - print 사용
    - 큰따옴표(")로 둘러싸인 문자열은 + 연산과 동일
      - print("life" "is" "too short")
      - print("life"+"is"+"too short")
    - 문자열 띄어쓰기는 콤마!
      - print("life", "is", "too short")
    - 한 줄에 결과값 출력
      - print(i, end=' ')
- 파일 읽고 쓰기
  - 파일 생성
    - f = open("new file.txt", 'w')
  - 파일을 쓰기 모드로 열어 출력값 적기
    - **f.write 사용! print X**
  - 저장된 파일 읽는 방법들 (print 사용)
    - 한 줄 읽기 : readline()
    - 모든 줄 읽기 : readlines()
    - 줄이 아니라 파일의 내용 전체를 문자열로 읽기 : read()
  - 파일에 새로운 내용 추가 (f.wirte 사용)
    - f = open("new file.txt", 'a')
  - with문 (close 없어도 됨)
    - with open("foo.txt", 'w') as f: / f.write("Life is too short, you need python")
  - (개인적인 경험) **format함수보다 문자열 포매팅이 더 깔끔하게 나온다.**

## 5. 클래스, 모듈, 예외 처리

- 클래스
  - 그런데 만약 한 프로그램에서 2개의 계산기가 필요한 상황이 발생하면 어떻게 해야 할까? (중간 생략)  
계산기 1의 결과값이 계산기 2에 아무런 영향을 끼치지 않음을 확인할 수 있다. 하지만 계산기가 3개, 5개, 10개로 점점 더 많이 필요해진다면 어떻게 해야 할 것인가? 그때마다 전역변수와 함수를 추가할 것인가? 여기에 빼기나 곱하기등의 기능을 추가해야 한다면 상황은 점점 더 어려워 질 것이다.
  - 클래스와 객체
    - 과자틀 -> 클래스 (class)
    - 과자틀에 의해서 만들어진 과자들 -> 객체 (object)
  - 객체와 인스턴스의 차이
    - a = Cookie()vx
    - a 입장에서
      - "a는 객체이다."
      - "a는 Cookie의 인스턴스이다."

- Cookie 입장에서
  - "Cookie"의 인스턴스는 a이다."
  - "Cookie에 대해서 a가 객체가 되기 위해서는 Cookie의 인스턴스가 되어야한다."
- 클래스를 만들 때는 반드시 어떻게 만들지를 먼저 구상하자!
- 클래스를 만들 때는 반드시 어떠한 동작을 하기 전에 그 동작을 할 수 있는 **재료** 들을 만들어주어야 한다. 쉽게 말해 사칙연산 클래스를 만들고 싶다면 재료들을 만들어놔야 사칙연산을 할 수 있다는 것이다.



- 파이썬 메서드의 첫번째 매개변수명은 관례적으로 **self** 라는 이름을 사용한다. 호출 시 호출한 객체 자신이 전달되기 때문에 self("self"는 자기자신이라는 뜻을 가진 영어단어이다.)라는 이름을 사용하게 된 것이다. 물론 self 말고 다른 이름을 사용해도 상관은 없다.
- self.first = first / self.second = second
  - = self.first = 4 / self.second = 2
  - = a.first = 4 / a.second = 2
  - = 객체.객체변수 = 값
  - 여기서 객체변수(instance variable)는 객체에 정의된 변수를 의미하며 객체간 서로 공유되지 않는 특징을 갖는다. 객체변수는 속성, 멤버변수 또는 인스턴스 변수라고도 표현한다.
- 생성자(Constructor) (*init*)
  - a = FourCal() / a.sum() 바로 쓰고 싶은데 안된다.. 언제까지 a = FourCal() / a.setdata(4,2) / a.sum()으로 해야하는가?
  - 파이썬 메서드명으로 *init* 을 사용하면 이 메서드는 생성자가 된다. *init* 메서드는 setdata 메서드와 이름만 다르고 모든게 동일하다. 단, 메서드 이름을 *init*으로 했기 때문에 생성자로 인식되어 객체가 생성되는 시점에 자동으로 호출되는 차이가 있다. *init* 메서드도 다른 메서드와 마찬가지로 첫번째 매개변수 self에 생성되는 객체가 자동으로 전달된다는 점을 기억하도록 하자.
    - 다시 말해, *init* 없이 기본적인 사용을 하면, a = FourCal() / a.setdata(4,2) / print(a.sum())이었지만, *init* 사용하면, a = FourCal(4,2) / print(a.sum())로 하면 끝난다. 왜냐하면 *init* 메서드를 사용함으로써 객체변수가 생성되기 때문이다.
  - 그래서 다음과 같이 출력하면 된다. a = FourCal(4,2) / print(a.first) print(a.second) / print(a.sum()) print(a.div())
- 상속
  - 상속(Inheritance)이란 "물려받다"라는 뜻으로, "재산을 상속받다"라고 할 때의 상속과 같은 의미이다. 클래스에도 이런 개념을 적용할 수가 있다. 어떤 클래스를 만들 때 다른 클래스의 기능을 물려받을 수 있게 만드는 것이다.
  - class MoreFourCal(FourCal): 곳
    - MoreFourCal : 자식클래스
    - FourCal : 부모클래스
  - 보통 상속은 기존 클래스를 변경하지 않고 기능을 추가하거나 기존 기능을 변경하려고 할 때 사용한다. 클래스에 기능을 추가하고 싶으면 기존 클래스를 수정하면 되는데 왜 굳이 상속을 받아서 처리해야 하지? 라는 의문이 들 수도 있다. 하지만 기존 클래스가 라이브러리 형태로 제공되거나 수정이 허용되지 않는 상황이라면 상속을 이용해야만 할 것이다.

- 여기서 더 나아가서 pow란 함수를 만들어서 기존의 부모클래스에 더하여 제공 계산도 할 수 있다.
- **메서드 오버라이딩**
  - class SafeFourCal(FourCal): 상속과 작성법은 똑같다.
  - SafeFourCal 클래스는 FourCal클래스에 있는 div라는 메서드를 동일한 이름으로 다시 작성하였다. **이렇게 부모 클래스(상속한 클래스)에 있는 메서드를 동일한 이름으로 다시 만드는 것을 메서드 오버라이딩(Overriding, 덮어쓰기)이라고 한다.** 이렇게 메서드를 오버라이딩하면 부모 클래스의 메서드 대신 오버라이딩한 메서드가 호출된다.
- 클래스 변수
  - class Family: / lastname = "이"
  - 객체를 생성했을 때 클래스 변수를 따로 Family.lastname = "박" 이런 식으로 바꿔주면 관련 객체에서도 동시에 클래스 변수가 바뀐다.
- 아마 웬만한건 class와 init만 있어도 더 편하게 사용할 수 있다.
  - 홍길동|42|A 같은 경우, split해서 따로 init에 리스트 형태를 index로 하여 자료들을 만들어두고, 객체를 data = Data("홍길동|42|A") 선언하면, print(data.age) print(data.name) print(data.grade) 으로 편하게 나타낼 수 있다.
- 모듈
  - 상속은 같은 파일 내에서 다른 클래스를 사용하고 싶거나(상속) 다른 클래스를 변형하고 싶을 때(오버라이딩), **모듈은 다른 파일을 쓰고 싶을 때**
  - 불러오는 방법은 import mod1(모듈이름)
    - **import는 현재 디렉터리에 있는 파일이나 파이썬 라이브러리가 저장된 디렉터리에 있는 모듈만 불러올 수 있다. 주의하자!**
  - 또다른 방법 from mod1 import sum 또는 from mod1 import \*
  - \$if name == "main" 😞
    - **이 파일에서 실행했을 때만 아래의 것들을 사용하겠다. 다른 파일에서 이 모듈을 실행시켰을 때는 실행하지 않도록 하는 것이다.**
  - 모듈에 포함된 변수, 클래스, 함수 사용
    - import mod2 / print(mod2.PI)
    - from mod2 import \* / a = Math() / print(a.solve(2))
    - from mod2 import Math, PI, sum 도 가능하다.
- 패키지
  - **패키지(Packages)는 도트(.)를 이용하여 파이썬 모듈을 계층적(디렉터리 구조)으로 관리할 수 있게 해준다.** 예를 들어 모듈명이 A.B인 경우 A는 패키지명이 되고 B는 A 패키지의 B 모듈이 된다.
  - 패키지 안의 함수 실행
    - import game / game.sound.echo.echo\_test() 이걸 좀 불편하다.
    - import game.sound.echo / game.sound.echo.echo\_test() 이걸 더 불편하다.
    - from game.sound import echo / echo.echo\_test() 위에게 보다는 편하다.
    - **from game.sound.echo import echo\_test / echo\_test() 이게 가장 편하다.**
    - **import game.sound.echo.echo\_test X!!!**
      - **도트 연산자(.)를 사용해서 import a.b.c처럼 import할 때 가장 마지막 항목인 c는 반드시 모듈 또는 패키지여야만 한다. 하지만 echo\_test는 함수이므로 불가능하다.**
  - *init.py*의 용도
    - *init.py* 파일은 해당 디렉터리(폴더)가 패키지(game 패키지)의 일부임을 알려주는 역할을 한다. 만약 game, sound, graphic등 패키지에 포함된 디렉터리에 *init.py* 파일이 없다면 패키지로 인식되지 않는다.
  - relative 패키지



- from game.graphic.render import render\_test를 from ..sound.echo import echo\_test 이  
렇게 바꿀 수 있다.
- relative한 접근자에는 다음과 같은 것들이 있다.
  - .. – 부모 디렉터리
  - . – 현재 디렉터리
- 예외 처리
  - 오류가 발생했을 때 이러한 오류를 무시하고 싶을 때도 있고 별도로 처리하고 싶을 때도 있다. 이에 파이썬은 try, except를 이용해서 오류를 처리할 수 있게 해준다.
  - 오류 예외 처리 기법
    - **try, except문**
      - try, except만 쓰는 방법
        - try: ... / except: ...
        - 잘 안쓰는 것 같다.
      - 발생 오류만 포함한 except문
        - try: ... / except 발생 오류: ...
        - try: a = [1,2] print(a[3]) / except ZeroDivisionError: print("블라블라")
        - 에러가 나면 print로 대체한다. 에러가 많이 났을 때 알아들을 수 있도록 바꾸는  
작업이라고 생각하면 편하다.
      - 발생 오류와 오류 메시지 변수까지 포함한 except문
        - try: ... / except 발생 오류 as 오류 메시지 변수: ...
        - try: a = [1,2] print(a[3]) / except ZeroDivisionError as e: print(e)
        - 에러가 나면 오류 메시지 변수를 통해서 좀 더 깔끔하게 출력할 수 있다.
    - try .. else문
      - try문은 else절을 지원한다. else절은 예외가 발생하지 않은 경우에 실행되며 반드시  
except절 바로 다음에 위치해야 한다. 만약 foo.txt라는 파일이 없다면 except절이 수행  
되고 foo.txt 파일이 있다면 else절이 수행될 것이다.
    - try .. finally
      - try문에는 finally절을 사용할 수 있다. finally절은 try문 수행 도중 예외 발생 여부에 상관  
없이 항상 수행된다. 보통 finally절은 사용한 리소스를 close해야 할 경우에 많이 사용된  
다.
      - finally: f.close()
      - 잘 안쓸 듯
    - 여러개의 오류처리하기
      - try: ... / except 발생 오류1: ... / except 발생 오류2: ...
      - 인덱싱 오류가 먼저 발생했으므로 4/0으로 발생되는 ZeroDivisionError는 발생하지 않  
았다. 다시 말해 먼저 발생하게 되면 뒤의 발생하지 않는다.
      - **except (ZeroDivisionError, IndexError) as e: print(e)** 이렇게 2개 이상의 오류  
를 동시에 처리하기 위해서는 위와같이 괄호를 이용하여 함께 묶어주어 처리하면 편하다.
  - 오류 회피
    - try .. except문에서 except문에 pass를 하면 오류가 통과된다.
  - **오류 일부러 발생(class를 통해) (일부러 오버라이딩하도록 만드는 것)**
    - 이상하게 들리겠지만 프로그래밍을 하다 보면 종종 오류를 일부러 발생시켜야 할 경우도 생긴다.  
파이썬은 raise라는 명령어를 이용해 오류를 강제로 발생시킬 수 있다.
    - class Bird: / def fly(self): / raise NotImplementedError 이런식으로 코드를 작성했을 때 상속받  
아서 다른 class에서 작동하고 싶더라도 오류가 발생하여 사용하지를 못한다.

- Eagle 클래스는 Bird 클래스를 상속받는다. 그런데 Eagle 클래스에서 fly 함수를 구현하지 않았기 때문에 Bird 클래스의 fly 함수가 호출된다. 그리고 raise문에 의해 다음과 같은 NotImplementedError가 발생할 것이다.
- 만약에 오버라이딩을 해서 바꾼다면 오류 메시지가 나타나도록 바꿀 수 있다.
- 다시말해 오버라이딩을 해야 오류 메시지가 뜨지 않는다.
- **예외 만들기 (class통해 직접 예외를 만들 수 있다.)**
  - 프로그램 수행 도중 특수한 경우에만 예외처리를 하기 위해서 종종 예외를 만들어서 사용하게 된다. 직접 예외를 만들어 보자. 예외는 다음과 같이 파이썬 내장 클래스인 Exception클래스를 상속하여 만들 수 있다.
  - class MyError(Exception): pass 를 통해 예외를 만들 수 있다. 그리고 raise MyError() 를 사용하여 예외를 사용할 수 있다.
  - 하지만 실행 해 보면 print(e)로 출력한 오류메시지가 아무것도 출력되지 않는 것을 확인 할 수 있다. 오류 메시지를 출력했을 때 오류 메시지가 보이게 하기 위해서는 오류 클래스에 다음과 같은 *str* 메서드를 구현해야 한다. *str* 메서드는 print(e) 처럼 오류메시지를 print문으로 출력할 경우에 호출되는 메서드이다.

## 6. 파이썬 라이브러리

### ○ 내장 함수

- 숫자
  - abs \*
  - abs(x)는 어떤 숫자를 입력으로 받았을 때, 그 숫자의 절대값을 돌려주는 함수이다.
  - pow \*
  - pow(x, y)는 x의 y 제곱한 결과값을 리턴하는 함수이다.
  - divmod
  - divmod(a, b)는 2개의 숫자를 입력으로 받는다. 그리고 a를 b로 나눈 몫과 나머지를 튜플 형태로 리턴하는 함수이다.
  - int
    - int(x)
    - int(x, radix)
      - int(x, radix)는 radix 진수로 표현된 문자열 x를 10진수로 변환하여 리턴한다.
      - print(int('11', 2)) / 3 (10진수 값)
      - print(int('1A', 16)) / 26 (10진수 값)
  - len \*
  - max
  - min
  - round \*
  - round(x)
    - 반올림
  - round(x, y)
    - round 함수의 두번째 파라미터는 반올림하여 표시하고 싶은 소수점의 자리수 (ndigits)이다.
- 문자
  - str \*
  - str(object)은 문자열 형태로 객체를 변환하여 리턴하는 함수이다.
  - eval \*
  - eval(expression)은 실행 가능한 문자열(1+2, 'hi' + 'a')을 입력으로 받아 문자열을 실행한 결과값을 리턴하는 함수이다.

- 보통 eval은 입력받은 문자열로 파이썬 함수나 클래스를 동적으로 실행하고 싶은 경우에 사용된다.
- chr
  - chr(i)는 아스키(ASCII) 코드값을 입력으로 받아 그 코드에 해당하는 문자를 출력하는 함수이다.
- ord
  - ord(c)는 문자의 아스키 코드값을 리턴하는 함수이다. ord 함수는 chr 함수와 반대이다.
- oct
  - oct(x)는 정수 형태의 숫자를 8진수 문자열로 바꾸어 리턴하는 함수이다.
- hex
  - hex(x)는 정수값을 입력받아 16진수(hexadecimal)로 변환하여 리턴하는 함수이다.
- 묶는 것
  - list \*
    - list(s)는 반복 가능한 자료형 s를 입력받아 리스트로 만들어 리턴하는 함수이다.
  - tuple
    - tuple(iterable)은 반복 가능한 자료형을 입력받아 튜플 형태로 바꾸어 리턴하는 함수이다. 만약 튜플이 입력으로 들어오면 그대로 리턴한다.
  - range \*
    - 인수가 하나
    - 인수가 2개
    - 인수가 3개
  - enumerate \*
    - enumerate는 "열거하다"라는 뜻이다. 이 함수는 순서가 있는 자료형(리스트, 튜플, 문자열)을 입력으로 받아 인덱스 값을 포함하는 enumerate 객체를 리턴한다.
    - for문처럼 반복되는 구간에서 객체가 현재 어느 위치에 있는지 알려주는 인덱스 값이 필요할때 enumerate 함수를 사용하면 매우 유용하다.
  - filter
    - filter란 무엇인가를 걸러낸다는 뜻으로, filter 함수도 동일한 의미를 가진다. filter 함수는 첫 번째 인수로 함수 이름을, 두 번째 인수로 그 함수에 차례로 들어갈 반복 가능한 자료형을 받는다. 그리고 두 번째 인수인 반복 가능한 자료형 요소들이 첫 번째 인수인 함수에 입력되었을 때 리턴값이 참인 것만 묶어서(걸러내서) 돌려준다.
    - lambda를 이용하면 더욱 간편하게 코드를 작성할 수 있다.
    - filter 함수는 거르는 것이 아니면 아래와 같이 효과가 없다.
  - map
    - map(f, iterable)은 함수(f)와 반복 가능한(iterable) 자료형을 입력으로 받는다. map은 입력받은 자료형의 각 요소가 함수 f에 의해 수행된 결과를 묶어서 리턴하는 함수이다.
    - lambda를 사용하면 다음처럼 간략하게 만들 수 있다.
    - map을 filter처럼 사용하면 다음과 같이 bool형으로 나온다.
  - zip \*
    - zip(iterable\*)은 동일한 개수로 이루어진 자료형을 묶어 주는 역할을 하는 함수이다.
  - sorted \*
    - sorted(iterable) 함수는 입력값을 정렬한 후 그 결과를 리스트로 리턴하는 함수이다.
    - 추가적으로 리스트 자료형에도 sort라는 함수가 있다. 하지만 리스트 자료형의 sort 함수는 리스트 객체 그 자체를 정렬만 할 뿐 정렬된 결과를 리턴하지는 않는다.
    - 평범한 소트
      - sorted(a) vs a.sort() 차이점

- key를 이용하여 소트하기
  - general ver. `print(sorted(students, key=lambda student: student[1]))`
  - class ver. `result = sorted(student_objects, key=lambda student: student.age) / print(result)`
  - key파라미터에는 함수가 와야 한다. key 파라미터에 함수가 설정되면 소트해야 할 리스트들의 항목들이 하나씩 key 함수에 전달되어 key 함수가 실행되게 된다. 이 때 수행된 key 함수의 리턴값을 기준으로 소트가 진행된다.
  - 위 예에서는 key함수에 students의 요소인 튜플데이터가 key함수의 입력으로 순차적으로 전달될 것이다. key함수는 입력된 튜플 데이터의 "나이"를 의미하는 2 번째 항목을 리턴하는 lambda함수이다. 따라서 sorted수행 후 나이순으로 소트 된 리스트가 리턴된다.
- operator 모듈
  - itemgetter
    - `result = sorted(students, key=itemgetter(1))`
    - itemgetter(1)과 같이 사용하면 students의 item인 튜플의 2번째 요소로 소트를 하겠다는 의미이다.
  - attrgetter
    - `result = sorted(student_objects, key=attrgetter('age'))`
    - attrgetter('age')와 같이 사용하면 students\_objects의 요소인 Student 객체의 객체변수 age로 소트를 하겠다는 의미이다.
- 순차 정렬과 역순 정렬
  - sort 또는 sorted함수에 reverse라는 파라미터를 이용하면 역순 정렬이 가능하다.
  - `result1 = sorted(student_objects, key=attrgetter('age'), reverse=True)`
- 중첩 소트
  - 이렇게 순서를 바꾸어 소트를 해 주어야 제대로 된 결과를 얻을 수 있게 된다. 이것이 가능한 이유는 소트할 때 기존의 순서를 그대로 유지하는(stable) 특성이 있기 때문이다. 위 예에서 보면 sally와 dave의 "성적"은 동일하지만 "나이"로 먼저 소트를 해 놓았기 때문에 그 순서가 유지되어 있는 것이다. 쉽게 말해 먼저 있는 열들을 소트해줘야 한다는 것이다.
- bool
  - all
    - all(x)은 반복 가능한(iterable) 자료형 x를 입력 인수로 받으며, 이 x가 모두 참이면 True, 거짓이 하나라도 있 으면 False를 리턴한다.
  - any
    - any(x)는 x 중 하나라도 참이 있을 경우 True를 리턴하고, x가 모두 거짓일 경우에만 False를 리턴한다. all(x)의 반대 경우라고 할 수 있다.
  - isinstance
    - isinstance(object, class)는 첫 번째 인수로 인스턴스, 두 번째 인수로 클래스 이름을 받는다. 입력으로 받은 인스턴스가 그 클래스의 인스턴스인지를 판단하여 참이면 True, 거짓이면 False를 리턴한다.
- 나머지
  - id
    - id(object)는 객체를 입력받아 객체의 고유 주소값(레퍼런스)을 리턴하는 함수이다.
  - type
    - type(object)은 입력값의 자료형이 무엇인지 알려주는 함수이다.

- input \*
  - input([prompt])은 사용자 입력을 받는 함수이다. 입력 인수로 문자열을 주면 아래의 세 번째 예에서 볼 수 있듯이 그 문자열은 프롬프트가 된다.
- 외장 함수
  - sys
    - 명령 행에서 인수 전달하기 - sys.argv
      - import sys / print(sys.argv) 하고 C:/doit/mymod>python argv\_test.py you need python 이라고 명령 프롬프트 창에 입력하면 ['argv\_test.py', 'you', 'need', 'python'] 으로 나온다.
      - 쉽게 말해 명령 프롬프트 창에서 입력을 주고 싶을 때 사용하는 것 같다. 어디에 쓰는지는 아직 잘 모르겠다.
    - 강제로 스크립트 종료하기 - sys.exit
    - 자신이 만든 모듈 불러와 사용하기 - sys.path
  - pickle \*
    - pickle은 객체의 형태를 그대로 유지하면서 파일에 저장하고 불러올 수 있게 하는 모듈이다.
      - pickle 모듈의 dump 함수를 이용하여 딕셔너리 객체인 data를 그대로 파일에 저장하는 방법
      - pickle.dump에 의해 저장된 파일을 pickle.load를 이용해서 원래 있던 딕셔너리 객체 (data) 상태 그대로 불러옴
    - 파일 읽고 쓰기와 같이 파일에 데이터를 저장하고, 다시 불러오는 것을 외장 함수를 통해서도 할 수 있다는 것을 보여준다.
  - os
    - 내 시스템의 환경 변수값을 알고 싶을 때 - os.environ
    - 디렉터리 위치 변경하기 - os.chdir
    - 디렉터리 위치 리턴받기 - os.getcwd
    - 시스템 명령어 호출하기 - os.system
    - 실행한 시스템 명령어의 결과값 리턴받기 - os.popen
  - shutil
    - 파일 복사하기 - shutil.copy(src, dst)
      - src라는 이름의 파일을 dst로 복사한다. 만약 dst가 디렉터리 이름이라면 src라는 파일 이름으로 dst라는 디렉터리에 복사하고 동일한 파일 이름이 있을 경우에는 덮어쓴다.
  - glob
    - 디렉터리에 있는 파일들을 리스트로 만들기 - glob(pathname)
  - tempfile
  - time
    - time.time
      - time.time()은 UTC(Universal Time Coordinated 협정 세계 표준시)를 이용하여 현재 시간을 실수 형태로 리턴하는 함수이다. 1970년 1월 1일 0시 0분 0초를 기준으로 지난 시간을 초 단위로 리턴한다.
      - ex. 1532695788.549929
    - time.localtime
      - time.localtime은 time.time()에 의해서 반환된 실수값을 이용해서 연도, 월, 일, 시, 분, 초,... 의 형태로 바꾸어 주는 함수이다.
    - time.asctime
      - 위의 time.localtime에 의해서 반환된 튜플 형태의 값을 인수로 받아서 날짜와 시간을 알아보기 쉬운 형태로 리턴하는 함수이다.

- ex. 'Fri Jul 27 21:49:48 2018'
- time.ctime
  - time.asctime(time.localtime(time.time()))은 time.ctime()을 이용해 간편하게 표시할 수 있다. asctime과 다른점은 ctime은 항상 현재 시간만을 리턴한다는 점이다.
  - 'Fri Jul 27 21:49:48 2018'
- time.sleep \*
  - time.sleep 함수는 주로 루프 안에서 많이 사용된다. 이 함수를 사용하면 일정한 시간 간격을 두고 루프를 실행할 수 있다.
  - import time / for i in range(10): / print(i) / time.sleep(1)
- calendar
  - calendar.weekday
  - calendar.monthrange
- random \*
  - random은 난수(규칙이 없는 임의의 수)를 발생시키는 모듈이다. random과 randint에 대해서 알아보자.
  - 0.0에서 1.0 사이의 실수 중에서 난수값을 리턴
    - import random / random.random()
  - 1에서 10 사이의 정수 중에서 난수값을 리턴
    - random.randint(1, 10)
  - random 모듈의 choice 함수를 사용하여 다음과 같이 좀 더 직관적으로 만들 수도 있다. random.choice 함수는 입력으로 받은 리스트에서 무작위로 하나를 선택하여 리턴한다.
    - random.choice(data)
  - 리스트의 항목을 무작위로 섞고 싶을 때는 random.shuffle 함수를 이용하면 된다.
    - random.shuffle(data)
- webbrowser
- namedtuple
- defaultdict
- threading
  - 컴퓨터에서 동작하고 있는 프로그램을 프로세스(Process)라고 한다. 보통 1개의 프로세스는 1가지 일만 하지만, 스레드를 이용하면 한 프로세스 내에서 2가지 또는 그 이상의 일을 동시에 수행하게 할 수 있다.
  - import threading / for msg in ['you', 'need', 'python']: / t = threading.Thread(target=say, args=(msg,)) / t.daemon = True / t.start()
    - 위 결과값에서 볼 수 있듯이 스레드는 메인 프로그램과는 별도로 실행되는 것을 확인할 수 있다.
  - 이러한 스레드 프로그래밍을 가능하게 해주는 것이 바로 threading.Thread 클래스이다. 이 클래스의 첫번째 인수는 함수 이름을, 두 번째 인수는 첫 번째 인수인 함수의 입력 변수를 받는다. 다음과 같이 스레드를 클래스로 정의해도 동일한 결과를 얻을 수 있다.

```

import threading
import time

class MyThread(threading.Thread):
    def __init__(self, msg):
        threading.Thread.__init__(self)
        self.msg = msg
        # self.daemon = True

    def run(self):
        while True:
            time.sleep(1)
            print(self.msg)

for msg in ['you', 'need', 'python']:
    t = MyThread(msg)
    t.start()

# main program
for i in range(100):
    time.sleep(0.1)
    print(i)

```

- 
- 반드시 써야하는 부분. `import threading / class blabla(threading.Thread) : / def $init(self, mag) : /threading.Thread.init$(self)`
- 스레드를 클래스로 정의할 경우에는 *init* 메서드에서 *threading.Thread.init(self)*와 같이 부모 클래스의 생성자를 반드시 호출해야 한다. MyThread로 생성된 객체의 start 메서드를 실행할 때는 MyThread 클래스의 run 메서드가 자동으로 수행된다. 다시 말해 threading.Thread로부터 파생클래스를 만드는 방식은 Thread 클래스를 파생하여 스레드가 실행할 run() 메서드를 재정의해서 사용하는 방식이다. Thread 클래스에서 run() 메서드는 스레드가 실제 실행하는 메서드이며, start() 메서드는 내부적으로 이 run() 메서드를 호출한다.

(to be continue..)