파이썬 신규 정보

\*문자열 포맷팅\*

1. %formatting

-> %.nf : n의자리만큼만 소수 표현

-> %% : %자체 표현

-> %x, %o : 16진수, 8진수

-> %c : 문자 1개 표현 (charactor)

-> %Nd or %Ns : 정수 or 문자열을 포함하여 N칸 출력폭 지정 가능 (왼쪽으로 정렬하고 싶으면 %-(N)d

ex> %8s %류건 -> (ㅡㅡㅡㅡㅡㅡ류건) < ㅡ는 공백 >

2. format Fucntion

"{index} {index} {index} ".format(variable) 형식

-> format() 안에 변수선언 후 초기화도 가능, 변수만 삽입 가능

ex> "{height}".format( height(=69) )

-> format 함수도 출력 폭 설정 가능!

"{index or variable : (출력형식)}".format(요소) 형식

출력 형식

1. (문자)<폭 -> 왼쪽 정렬로 포맷팅 한 후 나머지를 문자로 채움

ex> "{0[index임]:<10}".format(5)

"{0[index임]:(공백문자-변수도가능)<폭(출력형식)}".format(5)

ex> "{length: (공백 변수 or 문자)>10d}".format(length)

2. 문자>폭 -> 오른쪽 정렬로 포맷팅 한 후 나머지를 문자로 채움

3.문자^폭 -> 폭만큼 확보한 공간에 가운데 정렬로 포맷팅 후 나머지 문자로 채움

3. f{} 포맷팅

형식 : f "안녕하세요 저는 {함수식 or 변수명} 입니다."

ex> f"제 이름은 {name}이고 나이는 {age + 10} 이며 어쩌구"

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

문자열 함수

count()

find()

index()

join()

rstrip()

lstrip()

upper / lower

complex()

swapcase()

capitalize()

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

리스트

리스트의 요솟값을 인덱싱을 이용해 수정할때는 1대1 대응이다.

예시)

list[0:6] = "hello"라고 한다면 [h,e,l,l,o] 라고 슬라이스 됌

인덱싱과 슬라이싱의 차이점

list[2:3] = [a,b,c] => a b c가 list의 구성 요소로 들어감

list[2] = [a,b,c] => [a,b,c]라는 리스트가 list의 구성 요소로 들어감 (즉, 리스트 내의 리스트)

\*요소 삭제

list[3] = "" (인덱싱)

list[1:3] = [] (슬라이싱)

del list[3] <del 키워드는 객체 자체를 삭제함>

\*리스트 함수

append(x) : 리스트 마지막에 x 추가

insert(x,y) : x번째 인덱스에 y값 추가, 한번에 하나씩 가능

extend(x) : 기존 리스트에 x라는 리스트 합치기

remove(x) : 가장 처음 나오는 x라는 값을 삭제

a = list.pop() : 리스트 마지막 요소를 a에 초기화 시킨 뒤 삭제

sort() : 리스트 요소 순서대로 정리(오름차순), 전달 인자가 필요 없음

reverse() : 순서 뒤집기

index()

count()

len()

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

dictionary

\*순서 존재 x

\* key:value 형태 (JS의 객체와 비슷)

\*how to initialize

1. 딕셔너리 이름 = {key1 : value1, key2 : value2, ...}

2. 이름 = dict(key1=value1, key2=value2, ...)

=> 보통 1번이 직관적이므로 1번 많이 사용

\*값 추가

딕셔너리 이름[key] = value(리스트도 value로써 가능)

\*값 삭제

del 딕셔너리 이름[key] (객체 자체를 지워버림)

\*딕셔너리는 key가 immutable 타입.

주의사항

key는 value를 찾기 위한 유일한 값이므로 중복 x

key는 리스트 사용 불가

value는 어떤 값이는 상관없이 가능

\*dict 함수

x.keys() : dict의 key값만 모아 객체형식 dict\_keys로 반환

x.values() : dict의 value값만 모아 객체형식 dict\_values로 반환

x.items() : dict의 key와 value를 튜플로 묶어 dict\_items[(key1, value1), (key2, value2), ...]

x.clear() : 딕셔너리의 모든 값 삭제

x.get(key) : 해당 키의 value값 반환 (x[key]와 동일)

key in x : key 값이 x dict에 있는지 알려주는 조건문

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

tuple & set

tuple => immutable인 리스트

\*tuple -> ()로 선언!

\*tuple 입력 시 값이 하나이면 뒤에 콤마 입력

ex> t1 = ("hello, )

\*괄호 생략도 가능! ☆★

ex> t2 = "hello", 'b', 3, "goorm"

\*tuple은 저장되어있는 값을 수정하지 않는 선에서 리스트의 기능을 모두 지원

ex> t1 = ('a', 'b', 1, 2)

t2 = ("hello", )

print (t1 + t2) => ('a', 'b', 1, 2 "hello")

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

set => 중복,순서 없는 자료형

(수학 시간에 배운 집합의 특징을 그대로 구현한 자료형)

\*how to Initailize

ex> set = {3,1,2,3,5}

값 초기화 이외에 이미 존재하는 다른 묶음 자료형을 집합으로 변형 가능!

\*set(묶음 자료형)

-> string, tuple, list, dict 등을 set으로 변환 가능

순서, 중복 없애고 dict는 key 값만 저장!!

set은 리스트 or 튜플에 속한 요소의 중복을 제거하기 위한 필터로 사용

\*set function

교집합 -> &, intersection()

합집합 -> | , union()

차집합 -> - , difference()

set.add(a) : 집합에 a 추가

set.update( [a,b,c,...] ) : 집합 set에 여러 개의 값 추가

set.remove(a) : set에서 a를 삭제

\* 튜플은 한 개의 원소로 사용할 수 있지만 list & set 그 자체는 set의 원소로 사용 불가!!

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

반복문

for문에서 조건식에 여러값을 한번에 사용 가능

-> 조건문에서 묶여있는 값을 한번에 접근할때 사용!

ex> num1 = [[1,2,3],[4,5,6],['a','b','c']]

for i, j, k in num1:

print(i, j, k)

-> 1 2 3

4 5 6

a b c

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

조건문

따옴표, 괄호 안이 공백 // 숫자 0일 경우 False

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

function(함수)

지정 기능을 실행하는 단위

코드의 가독성과 프로그램 효율성 증대

# parameter

매개변수가 많아질 경우, 동일한 기능을 수행하지만 매개변수만 다른 함수 -> 가변 인자 함수로 선언 가능

def 함수name (\*parameter): 형식!

가변인자는 모든 인자에 대해 얼마나 많은 변수를 선언해야 할지 모르기 때문에 "tuple" 형식으로 저장됌

가변매개변수는 항상 parameter중 맨 뒤에 서술!

\* 키워드 parameter

매개변수 앞에 \*\*를 입력하면 함수 호출 시 전달 인자를 (key1 = value1, key2 = value2, ... ) 형식으로 입력해 딕셔너리 형태로 선언 가능!

ex> def func(\*\*key):

print(key)

num=10

func(apple = 사과, a=num, num=4)

-> {'apple' : '사과', 'a':10, 'num' : 4}

# 주의 : 키워드 parameter를 사용 시 key 값을 변수명 처럼 입력해야함. (따옴표, integer 노노)

\* return

return은 함수를 종료함과 동시에 값을 반환하는 키워드

return 값을 자료형의 값을 몇개 반환할 것인지 함수를 선언할 때 명시해야하고 선언한 대로 반환해야 함.

ex2 > def func(parameter):

,,,,

return a,b,c,d

rst1, rst2, rst3, rst4 = func(parameter)

-> result 출력

ex2> return 값을 list type으로 print

ex3> return 값 여러개 일때 list type으로 안묶으면 자동으로 tuple로 반환

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

# global variable & local variable

함수 내에서 선언된 변수 : local variable

외부에서 함수 내부의 변수에 접근하고 싶을 때 global variable 사용!

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

# 파일 접근

파일 read => f = open("파일경로/파일이름", "열기모드")

경로 미지정 시 현재 지정된 경로에 파일 생성

경로 "C:/data/test.txt" 형식으로 지정 가능

f.close()로 파일을 닫아야함.

열기 모드 : r(읽기), w(새로 작성), a(추가)

\* file read function

1. a = f.readline()

=> 파일 객체의 한 줄 씩 읽고 그 문자열을 반환합니다.

2. a = f.readlines()

=> 파일 객체의 모든 줄을 읽고 각 줄을 list로 return

3. a = f.read()

a = 파일 객체의 모든 문자열을 읽고 반환

\* file write function

f.write() > 파일에 쓰기 모드로 작성 (기존 내용 삭제)

\* with as

f.open()과 f.close()를 같이 쓰는 용법과 같은 구문

-> with open() as 파일객체명(f) :

구문

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

# class

클래스로부터 생성된 객체 => instance

인스턴스 생성 시 객체이름 = 클래스() 형식으로 ㄱㄱ

\* 클래스 변수 / 인스턴트 변수

클래스 안에서 선언된 변수 => 같은 클래스로 만들어진 인스턴스끼리 공유하고 접근 가능한 변수

-> instance에서 접근 시 객체이름.변수이름 형식

ex> class A:

a = 0

def area(self):

A.a += 1

\* method

-> 클래스 안에서 어떠한 기능을 수행하는 함수

(인스턴스 함수)

-> 사용자가 전달 인자를 입력하지 않는 매개변수 self 사용!

self는 실제 메소드를 사용할 떄 객체를 전달 받는 역할!

self.변수이름 형태로 사용하는 변수 => 인스턴스 변수

\* Constructor (생성자)

객체 생성과 동시에 초깃값 설정하는 메소드

기본 생성자는 클래스 선언 시 자동으로 생성

\_\_init\_\_(self) 라는 이름으로만 선언!

\*static method

self 매개변수를 갖지 않는 메소드

클래스안에서 선언되고 인스턴스로 실행하지만 일반 함수 같이 사용 가능!

형식

@staticmethod

def 정적메소드(self가 아닌 매개변수):

클래스명.정적메소드명으로 접근 가능

\* class method

클래스 변수에 접근할 때 사용, cls 로 클래스변수를 전달받음

인스턴스 필드에 접근하지 않을 때 static or class method를 사용하는데, 클래스 변수에 접근해야할 때 클래스 메소드로 접근하는 것이 용이!

\* class inheritance ☆☆☆

부모 클래스 <-> 자식 클래스

ex> 삼각형 -> 정삼각형 (세 각의 크기가 60인 특징 이외엔 삼각형의 고유 특징을 가지고있음)

형식

class 부모:

class 자식(부모)

-> 자식 클래스 내에서 부모에서 선언된 클래스변수값 변경 가능

-> 메소드 오버라이딩 가능

-> 부모 클래스에서 선언된 클래스 메소드는 자식 클래스의 인스턴트의 클래스 변수를 참조함.

-> 클래스 메소드는 부모 클래스에서 클래스 메소드를 실행하거나 자식 클래스 변수에 접근할 때 사용

-> 자식 클래스로 생성한 인스턴스는 부모 클래스의 메소드에 접근가능

\* 다형성

-> python은 부모 클래스를 여러 개 상속 가능

-> 메소드 오버라이딩 가능

-> 자식 클래스(부모1, 부모2, ...) 형식

-> MRO(method resolution order)에 따라 왼쪽에서 오른쪽 순서로 탐색함. 따라서 부모1의 메소드를 실행함.

따라서 부모2에서만 있는 메소드를 실행해도, 겹치는 변수가 있다면 부모1에서 가져옴

\* override vs overload

전자 : 상속받는 클래스에서 부모 클래스의 메소드 재정의

후자 : 한 클래스 안에서 같은 이름의 메소드를 여러 번 선언하는 것

python은 오버로딩을 지원하지 않음 -> 오버로딩 되어있으면 마지막 메소드만 실행됌.

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

# module

1. 효율성 -> 사용 용도에 따라 파일로 구분 뒤, 다른 파일에서 해당 클래스나 함수가 필요할 떄 가져와서 사용 가능

-> 모듈은 프로그램의 꾸러미

-> 모듈 가져오는 법 import

다른 파일에 있는 함수를 현재 사용 중인 파일에 포함하기 위한 함수

-> 모듈명.함수명 형식으로 다른 파일에 있는 함수 사용가능

from 모듈명 import 모듈함수

-> 함수명 앞에 모듈명 안붙이고도 사용 가능!

from 모듈명 import \*

- 모듈 내 모든 함수 바로 사용가능

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

# Exception handling

\* try ~ except~else~finally

try:

실행코드

except 에러이름 (as 메세지변수(e)) :

에러 발생 시 실행할 코드(ex> print(e))

else:

정상적 실행 시 실행될 구문

finally:

try문이 종결 시 실행될 구문

\* 강제적 오류 발생

raise 에러명

\* exception type

-> syntax Error (문법 오류)

-> IndentationError (들여쓰기)

-> typeError

-> ZeroDivisionError

->

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

# list 차이를 원하면 집합(set) 사용!

ㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡㅡ

# 자료(데이터) 구조와 알고리즘

1. 알고리즘 복잡도

\* 필요 이유 : 하나의 문제를 푸는 알고리즘은 많다. 어느 알고리즘이 더 좋은지 분석 하기 위함

\* 복잡도 항목

시간 복잡도(실행 속도), 공간 복잡도(메모리 사이즈)

\* 알고리즘 성능 표기법

1. Big O notation : O(N)

-> 알고리즘 최악의 실행시간 표기(아무리 최악이어도 이 정도 성능 보장)

-> 가장 많이 일반적으로 사용

2. Omega notation