# 지수표현/ 314 ∗ 0.01

number = 314e-2

시퀀스. 여러 값을 순서대로 저장. ≠정렬

Str은 슬라이싱 가능. 슬라이싱으로 새로운 시퀀스 생성 가능. 슬라이싱으로 내용 변경은 불가능\_타입에러.

print('hello'.capitalize()) #첫글자 대문자로 시작 ##Hello

s.strip 문자열 조작에 사용 ##입력받을때 rstrip()

my\_list = [1, 2, 3, 'Python', ['hello', 'world', '!!!']]

print(my\_list[-1][1][0]) # w

리스트.reverse() #배열뒤집기 dsgfh ⇒ hfgsd

Append. 리스트 끝에 항목 추가

Extend 모든 항목을 리스트 끝에 추가

Insert 원하는 위치에 항목 추가

.sort(reverse = true) #내림차순, 기본은 오름차순.

튜플은 내용변경 불가\_타입에러

# dict get

person = {'name': 'Alice', 'age': 25}

print(person.get('name')) # Alice

print(person.get('country')) # None

print(person.get('country', 'Unknown')) # Unknown

# dict

person = {'name': 'Alice', 'age': 25}

print(person.keys()) # dict\_keys(['name', 'age’])

print(person.values()) # dict\_values ([‘Alice’, 25])

print(person.items()) # dict\_items([('name', 'Alice'), ('age', 25)])

print(person.pop('age')) # 25

print(person) # {'name': 'Alice'}

print(person.pop('country', None)) # None

print(person.pop('country')) # KeyError

함수에 리턴이 없으면 None을 가진다. 함수 내에서 프린트를 실행하는 것과 다름. 함수의 종료조건 후 리턴을 적어주는 것이 안정적

새로운 변수에 할당으로 값을 줄 수 있음. 이는 객체 참조를 복사한 것\_new=old.

변수를 얕은 복사하면 분리되어 독립적으로 존재함. 중첩리스트의 경우 통하지 않는다.\_new=old[:]

아예 독립적인 객체를 만들려면 깊은 복사. Import copy / new=copy.deepcopy(old)

동등성을 확인하는 연산자 ==와 식별성을 확인하는 연산자 is를 잘 구분하여 사용해야 한다,

print(1 is True) # False

print(1 == True) # True

print(2 is 2.0) # False

print(2 == 2.0) #True

# 명시적 형변환  
# ValueError: invalid literal for int() with base 10: '3.5'  
print(int('3.5'))

#단축평가\_\_결과는 걍 외워버려ㅋ  
vowels = 'aeiou'  
#and는 뒤를 먼저보고 ,or는 앞을 먼저보는건데, 0은 언제나 반대의 경우까지 고려하게 됨.

print(('a' and 'b') in vowels) # False  
print(('b' and 'a') in vowels) # True  
print(('a' or 'b') in vowels) # True  
print(('b' or 'a') in vowels) # False  
print(3 and 5) # 5  
print(3 and 0) # 0  
print(0 and 3) # 0  
print(0 and 0) # 0  
print(5 or 3) # 5  
print(3 or 0) # 3  
print(0 or 3) # 3  
print(0 or 0) # 0

Braek는 아예 끝내기\_빠져나오기

Continue는 다음 반복으로 넘어가기

Pass는 동작하지 않고 진행하기

내장함수 enumerate를 사용한 반복은 인덱스와 원소를 동시에 접근하여 리스트를 튜플로 만들 수 있다.

fruits = ['apple', 'banana', 'cherry']

for index, fruit in enumerate(fruits):

print(f'인덱스 {index}: {fruit}')

함수 생성시 인자에 \*을 달면 리스트로 받을 수 있음. 인자 개수를 정하지 않고 입력받을 수 있음. 함수(1,2,3) ⇒[1,2,3]

함수 생성시 인자에 \*\*을 달면 딕셔너리로 받을 수 있음. 인자 개수를 정하지 않고 딕셔너리 정의하듯 하면 됨. 함수(a=1, b=2, c=3) ⇒{‘a’:1, ‘b’:2, ‘c’:3}

함수를 만들어서 파라미터로 패킹하면 튜플로 받아지고

직접 할당하면..? 리스트 형식.

# 인자의 모든 종류를 적용한 예시  
def func(pos1, pos2, default\_arg='default', \*args, \*\*kwargs):  
 print('pos1:', pos1) #1  
 print('pos2:', pos2) #2  
 print('default\_arg:', default\_arg) #3  
 print('args:', args) #(4,5,6) #파라미터를 이용해서 튜플  
 print('kwargs:', kwargs) #{'key1': 'value1', 'key2': 'value2'}  
func(1, 2, 3, 4, 5, 6, key1='value1', key2='value2')

<map object at 0x00000239C915D760> 같은 출력은 list등으로 분해해서 볼 수 있음.

lia = [10,20,30,40]

lib = [40,50,60,70]

lic = [70,80,90,100]

fot i in zip(lia, lib, lic) :

print(i)

#(10, 40, 70)

#(20, 50, 80)

#(30, 60, 90)

#(40, 70, 100)

add = lambda x,y : x+y

result = add(3,5)

print(result) #8

#스코프

a = 1  
b = 2  
def enclosed():  
 a = 10  
 c = 3  
 def local(c):  
 print(a, b, c) # 10 2 500  
 local(500)  
 print(a, b, c) # 10 2 3  
enclosed()  
print(a, b) # 1 2

=======================

클래스. 타입을 표현하는 방법. 객체 생성을 위한 설계도. 데이터와 기능을 묶는 방법 정의

객체 : 메모리에 할당된 속성과 행동 등 모든 것.

클래스로 만든 객체 : 인스턴스\_아이유는 가수 클래스의 인스턴스 이다.

클래스를 만든다 = 타입을 만든다

하나의 객체는 특정 타입의 인스턴스이다.

객체.(타입\_어떤 연산자(오퍼레이터)와 조작(메소드)이 가능한지. / 속성\_어떤 상태(데이터)를 가지는지. / 조작\_어떤 행위(함수)를 할 수 있는지.)

클래스( 생성자 함수. 객체 생성시에 자동 호출. \_\_init\_\_하면 객체 초기화 / 클래스 변수. 내부선언변수. 클래스로 생성된 모든 인스턴스들이 공유.) class.class\_variable로 클래스 변수 참조.

메서드 종류( 인스턴스 메서드/클래스 메서드/정적 메서드)

인스턴스 메서드 : 클래스에서 생성된 인스턴스에서 호출할 수 있는 메서드. 조작동작. 첫번째 변수로 자신을 전달받음\_self /생성자 메서드 = 인스턴스 메서드. 객체 생성시에 자동 호출.

클래스 메서드 : 클래스가 호출하는 메서드. 조작 동작. @classmethod를 데코레이터로 사용하여 정의. 첫번째 변수로 호출\_cls

정적 메서트 : 클래스와 인스턴스의 특징의 상호작용없이 독립적으로 동작. @staticmethod를 데코레이터로 사용하여 정의. 필수 매개변수 없음.

인스턴스와 클래스의 이름공간. 클래스를 정의하면 클래스와 해당하는 이름공간이 생성됨. 인스턴스를 정의하면 객체가 생성되고 독립적 이름공간 생성

인스턴스에서 속성에 접근하면 인스턴스-> 클래스로 탐색.

class를 정의하면 메모리(이름공간)에 생성자 함수와 클래스 변수가 저장됨. 인스턴스를 생성하면 인스턴스 별로 따로 저장된다.

누가 어떤 메서드를 사용할지

클래스 : 클래스 메서트, 스태틱(정적) 메서드

인스턴스 : 인스턴스 메서드

# 인스턴스 메서드

class Person:

def \_\_init\_\_(self, name):

self.name = name

print('인스턴스가 생성되었습니다.')

def greeting(self):

print(f'안녕하세요. {self.name}입니다.')

person1 = Person('지민') # 인스턴스가 생성되었습니다.

person1.greeting() # 안녕하세요. 지민입니다. #그레이팅함수를 불러야 뭐가 되고 아니고는 걍 메서드만 가지고 있는 상태..?

# 클래스 메서드

class Person: #클래스 이름공간1

count = 0 #클래스 이름공간1

def \_\_init\_\_(self, name):

self.name = name

Person.count += 1

@classmethod

def number\_of\_population(cls):

print(f'인구수는 {cls.count}입니다.')

person1 = Person('iu') #인스턴스별 이름공간1

person2 = Person('BTS') #인스턴스별 이름공간2

print(person1.number\_of\_population()) #인구수는 2입니다. // 모든 클래스에서 클래스메서드가 사용돼서 첫번째를 불러도 2라고 나오는듯?

# 정적 메서드

class StringUtils:

@staticmethod

def reverse\_string(string):

return string[::-1]

@staticmethod

def capitalize\_string(string):

return string.capitalize()

text = 'hello, world'

reversed\_text = StringUtils.reverse\_string(text) #스트링언틸의 리버스함수 사용해서 뒤집어 출력

print(reversed\_text) # dlrow ,olleh

capitalized\_text = StringUtils.capitalize\_string(text) #스트링언틸의 케피탈라이즈함수 사용해서 첫글자 대문자

print(capitalized\_text) # Hello, world

매직 메서드(스페셜 메서드/인스턴스 메서드). 특정 상황에서 자동 호출. \_\_str\_\_(self) 등 더블언더스코어를 사용함

\_\_str\_\_(self)는 print로 호출된 객체 출력을 문자열로 표현 변경. 상태말고 내용을 볼 수 있도록.

Class 클래스이름: def 클래스 함수 def \_\_str\_\_(self): return self.변수 ⇒ 클래스 수행후에 할당되는 것이 아니라 바로 변수가 출력된다.

이터레이터(Iterator) : 반복가능한 객체의 요소를 하나씩 반환하는 객체. 내부적으로 for 문이 동작할 때 반복가능한 객체에 대해 iter() 호출. iter()에서는 메서드 \_\_next()\_\_를 정의하는 이터레이터 객체를 돌려줌. \_\_next()\_\_에서는 반복가는 객체를 한번에 하나씩 접근하도록 제어하고 남은게 없으면 StopIteration으로 for문을 종료

class Reverse : #이터레이터 동작을 클래스로 정의

def \_\_init\_\_(self, data):

self.data = data #값이 들어오면 내용 저장

self.index = len(data) #값이 들어오면 길이 확인

def \_\_iter\_\_(self) :

return self

def \_\_next\_\_(self) :

if self.index == 0 :

raise StopIteration #강제종료 발생

self.index = self.index -1

return self.data[self.index]

rev = Reverse('abc')

for char in rev :

print(char)

#c #b #a

에러와 예외

문법 에러 : Syntax Error : 문법오류나 잘못된 할당. 소괄호문제 등 코딩 문제

예외 : Exception : 내장 예외\_파이썬에서 이미 정의된 상황에 대한 처리.

ZeroDivisonError : 나누기 또는 모듈 연산의 두번째 인자가 0일 때.

NameError : 지역 또는 전역 이름을 찾을 수 없을 때

TypeError : 타입 불일치 또는 인자 누락, 인자 초과, 인자 타입 불일치

ValueError : 연산이나 함수에 문제가 없지만 부적절한 값을 가진 인자를 받음

IndexError : 시퀀스 인덱스가 범위를 벗어날 때 발생

KeyError : 딕셔너리에 해당 키가 존재하지 않는 경우

ModuleNotFoundError : 모듈을 찾을 수 없을 때 발생

ImportError : import 이름을 찾을 수 없을 때 발생

KeyboardInterrupt : 사용자가 ctrl-C또는 del을 누를 때 발생. 무한루프 강제종료

IndentationError : 잘못된 들여쓰기 등 문법 오류

예외처리 : Exception Handling : 예외 발생 시에 적절하게 처리되도록 하는 방법. 복수처리 가능\_하위 예외클래스부터 확인하도록

try : 예외 발생 가능 코드

except : 예외 발생 시 실행 코드\_오류가 있어도 코드가 돌아감.

else : 예외 발생하지 않을 시 실행 코드

finally : 예외발생여부에 무관하게 항상 실행할 코드

모듈 : 한 파일로 묶인 변수와 함수의 모음. 특정한 기능을 하는 코드가 작성된 파이썬 파일.

import 모듈이름\_으로 불러와서 모듈이름.기능()\_으로 사용

from 모듈이름 import 기능\_으로 불러와서 기능()\_으로 사용

모듈명.내용\_\_온점 왼쪽 객체에서 온점 오른쪽 이름을 찾아라.

서로 다른 모듈이 같은 이름의 함수를 제공한다면, 마지막에 import 한 모듈이 사용됨.

import 모듈 as 별칭\_\_as를 이용해서 별칭 설정 가능. 위에서 언급한 이름 충돌 방지.

모듈 직접 만들기

1. 파이썬 파일에 함수 작성

2. 저장\_제목.py

3. 새 파이썬 파일에서 import 제목 하고 함수 호출

help(모듈명)으로 모듈 내부 살펴보기 가능

표준 라이브러리 : 파이썬 언어와 함께 제공되는 모듈과 패키지

패키지 : 연관 모듈을 하나의 디렉토리에 모아 놓은 것

패키지 폴더 → 모듈폴더(→ 모듈파일) & 모듈폴더(→ 모듈파일) 의 방식으로 구조화 // from 패키지폴더.모듈폴더 import 모듈파일 \_\_로 호출하여 사용 가능

PSL내부 패키지는 import 로 바로 사용. 외부패키지는 pip install로 설치하여 import 하고 사용

패키지가 모듈의 이름공간을 구분하므로 충돌 방지 가능

패키지는 모듈을 효율적으로 관리하고 재사용 할 수 있도록 도와줌

