



Python 기본 개념과 문법

컬렉션 데이터 타입

함수 & 파일 읽기 쓰기

탐색과 정렬

객체지향과 예외처리

정규표현식

1. 객체지향 프로그래밍

2. 클래스

3. isinstance 함수

4. object

5. 모듈

6. 패키지

7. site-packages

8. 파이썬 라이브러리

9. 객체지향 이론 - 캡슐화

10. 객체지향 이론 - 다형성

11. 예외처리 (try - except -)

12. 예외처리 (else, finally)

13. 예외 발생 시키기 (raise)

14. 예외 형식 만들기

객체지향 프로그래밍 (OOP: Object Oriented Programming)

컴퓨터 프로그래밍 패러다임 중 하나로, 프로그래밍에서 필요한 데이터를 추상화시켜, 상태와 행위를 가진 객체를 만들고 그 객체들 간의 유기적인 상호작용을 통해 로직을 구성하는 프로그래밍 방법

객체지향

- 프로그램을 여러 개의 객체 단위로 나누어서 작업하는 방식
- 객체들이 유기적으로 상호작용
- 대표적인 OOP 언어는 Java, C++, Python 등이 있다.

장점

- 코드의 재사용
- 생산성 향상
- 유지보수에 용이

단점

- 개발 속도가 느림
- 실행 속도가 느림
- 코딩이 어려움

클래스

연관이 있는 처리 부분(함수)과 데이터 부분(변수)을 하나로 묶은 클래스로, 객체를 생성해서 사용 가능합니다.

클래스

Animal



객체

Pig



Monkey



Panda



Rabbit



클래스

연관이 있는 처리 부분(함수)과 데이터 부분(변수)을 하나로 묶은 클래스로, 객체를 생성해서 사용 가능합니다.

클래스 정의

```
class Animal:  
    # 클래스 변수  
    # 클래스 함수  
    # 생성자  
    # 인스턴스 함수
```

```
class Animal:  
    pass
```

```
pig = Animal()  
print(type(pig))
```

객체 생성

```
pig = Animal()
```

클래스

연관이 있는 처리 부분(함수)과 데이터 부분(변수)을 하나로 묶은 클래스로, 객체를 생성해서 사용 가능합니다.

클래스 정의

```
class Animal:
    age = 1
    def set_name(self, data):
        self.name = data
```

객체 생성 및 함수 호출

```
pig = Animal()
pig.set_name('돼지') # Animal.set_name(pig, '돼지')
print(f'{pig.name}고, {pig.age}살입니다.')
```

클래스

class 변수의 변경은 생성된 객체의 값에 영향을 주지 않습니다.

클래스 정의

```
class Animal:
    age = 1
    def set_name(self, data):
        self.name = data
```

객체 생성 및 함수 호출

```
pig = Animal()
pig.set_name('돼지')

panda = Animal()
Animal.set_name(panda, '판다')

pig.age = 2
panda.age = 3

print(f'{pig.name}이고, {pig.age}살입니다.')
print(f'{panda.name}이고, {panda.age}살입니다.')
```

클래스

생성자는 객체를 만들 때, 최초로 호출되는 함수로 일반적으로 초기화 할 때 사용됩니다.

클래스 정의

```
class Animal:
    def __init__(self):
        self.name = 'unnamed'
        self.age = -1
    def info(self):
        print(f'{self.name}이고, {self.age}살입니다.')
```

객체 생성 및 함수 호출

```
pig = Animal()
pig.name='돼지'
pig.age=1
pig.info()

panda = Animal()
panda.info()
```


클래스

상속을 하면 다른 클래스의 기능을 그대로 사용할 수 있습니다.

클래스 정의

```
class Animal:
    def __init__(self):
        self.name = 'unnamed'
        self.age = -1
    def info(self):
        print(f'이름: {self.name}, 나이: {self.age}')

class Human(Animal):
    def speak(self, data):
        print(f'{self.name}: {data}')
```

객체 생성 및 함수 호출

```
hong = Human()
hong.name = '홍길동'
hong.age = 30
hong.info()
hong.speak('안녕하세요')
```

클래스

```
class Animal:
    def __init__(self):
        self.name = 'unnamed'
        self.age = -1
    def info(self):
        print(f'이름: {self.name}, 나이: {self.age}')
```

```
class Human(Animal):
    def __init__(self):
        self.job = 'student'
        super().__init__()
    def speak(self, data):
        print(f'{self.name}: {data}')
    def info(self):
        print(f'이름: {self.name}, 나이: {self.age}, 직업: {self.job}')
```

부모 생성자 호출

오버라이딩

```
hong = Human()
hong.name = '홍길동'
hong.age = 30
hong.info()
hong.speak('안녕하세요')
```

클래스

```
class Car:
    def __init__(self, id):
        self.id = id
    def __len__(self):
        return len(self.id)
    def __str__(self):
        return 'Vehicle number: ' + self.id
```

```
def main():
    c = Car('1271234')
    print(len(c))
    print(str(c))
```

```
main()
```

isinstance 함수

isinstance()는 객체가 어떤 클래스의 인스턴스인지, 즉 객체에 어떤 타입이 있는지를 알려줍니다.

```
# 'abc'는 str의 인스턴스이지만 123은 아닙니다.  
isinstance('abc', str) # True  
isinstance(123, str) # False  
  
# 모든 클래스는 object 클래스를 기반으로 합니다.  
isinstance('abc', object) # True  
isinstance(123, object) # True  
  
# 클래스와 함수 조차도 object의 인스턴스입니다.  
isinstance(str, object) # True  
isinstance(max, object) # True  
  
# 이는 파이썬 내 모든 클래스가 object로부터 상속받는다를 것을 의미합니다.
```

object

파이썬 내 모든 클래스들은 object 클래스에 들어 있는 속성들을 상속 받습니다.

```
print(dir(object))

class Book:
    pass

print(dir(Book))

print(set(dir(Book)) - set(dir(object)))
```

모듈

모듈은 파이썬 파일(.py)에 변수와 함수 등을 모아놓은 것을 말합니다.
자주 쓰는 것들은 모듈로 만들어 두면 유용하게 사용할 수 있으며, 모듈은 "import 모듈명" 으로 이용 가능합니다.

```
import math

print(type(math))

print(math.pi) # math 모듈에 있는 파이 변수
print(math.sqrt(9)) # math 모듈에 있는 제곱근 구하는 함수

area = 78.53981633974483
print('반지름은', math.sqrt(area / math.pi))
```

모듈

만약 모듈명과 모듈 내 요소명을 결합해서 사용하는 것이 불편한 경우에는 사용자가 모듈에서 어떤 요소를 import할지 명시하여, namespace로 import 할 수 있습니다.

서로 다른 모듈이 같은 이름의 변수 또는 함수를 가지고 있는 경우 문제가 될 수 있습니다.

또한 * 문자를 사용하면 모듈 내 모든 요소를 한 번에 전부 import 할 수도 있지만, 부정확한 함수에 접근하거나 올바르게 동작하지 않을 위험이 있습니다.

```
from math import pi, sqrt
from anotherModule import pi, sqrt

print(pi)
print(sqrt(9))
```

```
from math import * # 위험!!!

print(pi)
print(sqrt(9))
```

모듈

모듈은 파이썬 파일(.py)에 변수, 클래스 등을 모아놓은 것을 말합니다.
자주 쓰는 함수는 모듈로 만들면 유용하게 사용할 수 있으며, 모듈은 "import 모듈명" 으로 이용 가능합니다.

```
def gender(data):  
    data = data.replace(" ", "").replace("-", "")  
    if int(data[6]) % 2 == 1: return '남자'  
    else : return '여자'  
  
def birth(data):  
    data = data.replace(" ", "").replace("-", "")  
    year = data[0:2]  
    month = data[2:4]  
    day = data[4:6]  
    if int(data[6]) < 3: year = '19' + year  
    else : year = '20' + year  
    return year, month, day
```

```
import rrn_util as rn  
  
data = "930625-1*****"  
  
print(rn.gender(data))  
print(rn.birth(data))
```


모듈

import 를 하면 어떤 일이 일어날까요?

```
print("이 모듈을 읽어들였습니다.")
```

```
import experiment_module
```

모듈

import할 때는 실행되지 않도록 하고, 모듈이 직접 실행될 때만 실행되도록 하고 싶다면?

```
print("이 모듈을 읽어들였습니다.", __name__)
```

```
import experiment_module  
print("파일 자체의 __name__은", __name__)
```

모듈

import할 때는 실행되지 않도록 하고, 모듈이 직접 실행될 때만 실행되도록 하고 싶다면?

```
def gender(data):
    data = data.replace(" ", "").replace("-", "")
    if int(data[6]) % 2 == 1: return '남자'
    else : return '여자'

def birth(data):
    data = data.replace(" ", "").replace("-", "")
    year = data[0:2]
    month = data[2:4]
    day = data[4:6]
    if int(data[6]) < 3: year = '19' + year
    else : year = '20' + year
    return year, month, day

if __name__ == "__main__" :
    rrn_data = input("""주민등록번호를 입력하세요:
예시) 930625-1*****
""")
    print(gender(rrn_data))
    print(birth(rrn_data))
```

```
import rrn_util as rn
data = "930625-1*****"
print(rn.gender(data))
print(rn.birth(data))
```

패키지

패키지는 디렉토리에 모아진 모듈의 묶음을 의미합니다.

```
import rrn.rrn_util as rn  
data = "930625-1*****"  
print(rn.gender(data))  
print(rn.birth(data))
```

```
from rrn import rrn_util as rn  
data = "930625-1*****"  
print(rn.gender(data))  
print(rn.birth(data))
```

site-packages

파이썬의 기본 라이브러리 패키지 외에 추가적인 패키지를 설치하는 디렉토리.
각종 서드파티 모듈들이 이 곳에 설치됩니다.

```
import sysconfig  
  
print(sysconfig.get_paths()["purelib"])
```

파이썬 라이브러리

패키지는 디렉토리에 모아진 모듈의 묶음을 의미합니다.

```
# time - 시간과 관련된 모듈
import time

# 시간을 초단위(utc_협정세계표준시)로 반환
time.time()

# 연월일시분초의 형태로 반환
time.localtime()

# 시간을 요일, 월, 일, 시, 분, 초, 년도 순서의 문자열로 반환
time.asctime()

# 반복문 등에서 사용, sleep(초)만큼 딜레이
time.sleep(3)
```

파이썬 라이브러리

패키지는 디렉토리에 모아진 모듈의 묶음을 의미합니다.

```
import glob # 파일들의 목록을 뽑을 때 사용
```

```
print(glob.glob("*.py")) # 현재 디렉토리에 있는 모든 .py 파일을 리스트로 반환  
print(glob.glob("C:/*")) # C 드라이브에 있는 모든 파일을 리스트로 반환
```

```
import shutil # 파일을 복사해주는 모듈
```

```
shutil.copy("src.txt", "dst.txt") # src.txt를 dst.txt로 복사  
shutil.move("src.txt", "dst.txt") # src.txt를 dst.txt로 이동
```

```
import json # json 형태로 데이터를 주고 받을 때 사용
```

```
json.dumps({'name':'kim', 'age':30}) # json 데이터 만들기  
json.loads('{"name":"kim", "age":30}') # json 데이터 읽기
```

객체 지향 이론 - 캡슐화

캡슐화 - 객체의 속성과 행위를 하나로 묶고, 실제 구현 내용 일부를 외부에 감추어 은닉하는 것

```
width = 3  
height = 2
```

```
width2 = 4  
height2 = 6
```

```
perimeter1 = 2 * (width + height)  
perimeter2 = 2 * (width2 + height2)  
print(perimeter1, perimeter2)
```

```
class Rectangle:  
    def __init__(self, width, height):  
        self.width = width  
        self.height = height  
  
    def get_perimeter(self):  
        return 2 * (self.__get_width_plus_height())  
  
    def __get_width_plus_height(self): # private method  
        return self.width + self.height
```

```
rectangle1 = s.Rectangle(3, 2)  
rectangle2 = s.Rectangle(4, 6)  
perimeter1 = rectangle1.get_perimeter()  
perimeter2 = rectangle2.get_perimeter()  
  
print(perimeter1, perimeter2)
```


객체 지향 이론 - 다형성

다형성 - 같은 이름의 메서드나 연산자를 다른 클래스에서 다르게 구현하는 것.

즉, 같은 이름의 메서드를 호출하더라도, 서로 다른 클래스에서는 그 결과가 다르게 반환될 수 있도록 하는 것

```
class Animal:
    def sound(self):
        print("동물의 소리입니다.")
```

```
class Cat(Animal):
    def sound(self):
        print("야옹~")
```

```
class Dog(Animal):
    def sound(self):
        print('멍! 멍!')
```

```
cat = Cat()
cat.sound()
```

```
dog = Dog()
dog.sound()
```

예외처리

예외 : 문법적으로 문제가 없는 코드를 실행하던 중에 발생하는 오류

오탈자 또는 대소문자 체크

NameError: name '..' is not defined

import하려는 모듈 이름 또는 위치 체크

ImportError: No module named

따옴표, 괄호의 여닫기 체크

SyntaxError: EOL while scanning string literal

SyntaxError: unexpected EOF while parsing

SyntaxError: invalid syntax

import한 모듈의 내 함수, 변수명 체크

AttributeError: 'module' object has no attribute

함수 매개변수 개수 체크

TypeError: ... missing ... required positional argument: ...

들여쓰기 체크

SyntaxError: expected an indented block

SyntaxError: unindent does not match any outer indentation level

SyntaxError: unexpected indent

자료형 체크

ValueError: invalid literal for ...()

예외처리

예외 처리는 try ~ except 구문을 이용합니다.

```
try:  
    # 실행할 코드  
except:  
    # 문제가 생겼을 때 실행할 코드
```

예외처리

```
try:
    num = int(input("100을 나누기 위한 정수를 입력하세요: "))
    print(100/num)
except:
    # 0, 소수점 입력 시 예외 발생
    print("예외가 발생했습니다.")

print("프로그램이 정상 종료되었습니다.")
```

예외처리

만약 상세한 예외 정보를 얻고 싶다면 as 문을 사용하면 됩니다.

```
try:
    num = int(input("100을 나누기 위한 정수를 입력하세요: "))
    print(100/num)
except Exception as e:
    # 0, 소수점 입력 시 예외 발생
    print(e)
    print(e.__class__)

print("프로그램이 정상 종료되었습니다.")
```

예외처리

예외 클래스는 계층 구조를 형성하여 다양한 예외 상황을 처리할 수 있도록 설계되어 있습니다.

BaseException

BaseExceptionGroup

ExceptionGroup

Exception

ArithmeticError

⋮

ValueError

Warning

```
def exception_hierarchy(exception_class, indent=0):  
    # 들여쓰기를 적용하여 예외 클래스 이름 출력  
    print(" " * indent + exception_class.__name__)  
    # 하위 클래스 목록  
    subclasses = exception_class.__subclasses__()  
    # 각 하위 클래스에 대해 재귀적으로 출력  
    for subclass in subclasses:  
        exception_hierarchy(subclass, indent + 2)  
  
# BaseException부터 시작하여 예외 클래스 계층 구조 출력  
exception_hierarchy(BaseException)
```

예외처리

늘어나는 예외 종류에 따른 예외 처리를 구현하기 위해서는 하나 이상의 except 절을 이용해야 합니다.

```
def hundred_divisor():  
    try:  
        input_number = int(input("100의 약수를 입력하세요: "))  
        if 100 % input_number == 0:  
            print("맞습니다.")  
        else:  
            print("틀립니다.")  
    except ValueError as err:  
        print("정수를 입력하지 않아 프로그램을 종료합니다.", err)  
    except ZeroDivisionError as err:  
        print("0으로 나눌 수 없습니다.", err)  
        hundred_divisor()  
  
hundred_divisor()
```

예외처리

```
def hundred_divisor():  
    try:  
        input_number = int(input("100의 약수를 입력하세요: "))  
        if 100 % input_number == 0:  
            print("맞습니다.")  
        else:  
            print("틀립니다.")  
    except ValueError as err:  
        print("숫자 형식에 맞지 않아 프로그램을 종료합니다.", err)  
    except ZeroDivisionError as err:  
        print("0으로 나눌 수 없습니다.", err)  
        hundred_divisor()  
    except:  
        print("알 수 없는 에러가 발생했습니다.")  
  
hundred_divisor()
```


예외처리 - else

try절에 있는 코드들이 무사히 실행하게 되면 else 절에 있는 내용이 실행됩니다.

```
try:
    # 실행할 코드
except:
    # 문제가 생겼을 때, 실행할 코드
else:
    # except절을 만나지 않았을 때, 실행할 코드
```

예외처리 - finally

어떤 일이 있어도 반드시 실행되는 finally는 예외의 발생 여부에 상관없이 무조건 실행됩니다.

```
try:
    # 실행할 코드
except:
    # 문제가 생겼을 때, 실행할 코드
finally:
    # 무조건 실행할 코드
```

예외발생 시키기 - raise

raise문을 통해 예외를 발생시킬 수 있습니다.

```
def guess_number_up_down():  
    import random  
    random = random.randint(1, 31)  
  
    while True :  
        number = int(input("1부터 31 사이의 숫자를 입력하세요 : "))  
        if(number < 1 or number > 31):  
            raise Exception("1부터 31 사이의 숫자를 입력하세요.")  
        if number == random :  
            print("맞습니다.")  
            break  
        elif number < random : print("UP")  
        else : print("DOWN")
```

```
guess_number_up_down()
```

예외발생 시키기 - raise

Exception 클래스를 상속하는 클래스를 정의하는 것으로 예외 형식을 만들 수 있습니다.

```
def sum_odd_to_ten():  
    try:  
        number = get_number()  
        sum = 0  
        range_number = number  
        if number % 2 == 1: range_number += 1  
        for i in range(1, range_number, 2):  
            sum += i  
        print(f"1) 1부터 {number}까지 홀수의 합은 {sum}입니다.")  
    except Exception as err:  
        print(err)  
        raise Exception("2) 에러를 전달합니다.")  
  
def get_number():  
    number = int(input("숫자를 입력하세요: "))  
    if number < 1:  
        raise Exception("1) 예외가 발생했습니다. 자연수만 입력할 수 있습니다.")  
    return number
```

```
try:  
    sum_odd_to_ten()  
except Exception as err:  
    print(err)
```

예외 형식 만들기

Exception 클래스를 상속하는 클래스를 정의하는 것으로 예외 형식을 만들 수 있습니다.

```
class MyException(Exception):  
    def __init__(self):  
        super().__init__(self, "예상치 못한 답변으로 에러가 발생했습니다.")
```

```
select = input("파이썬이 재미있으면 1번, 아니면 2번")  
if select == '1': print("파이썬은 역시 재미있어요")  
elif select == '2' : raise MyException  
else: print("1번과 2번만 입력하세요")
```