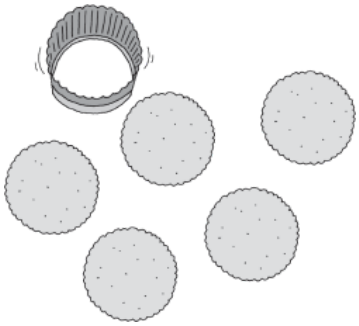
Basic Python: Part III

Date	@2022년 8월 18일 오전 9:30
Code: Lecture	C2-004_Lecture_03.ipynb
Code: Practice	C2-004_Practice_03.ipynb
Code: Practice-Sol	C2-004_Practice_03-Sol.ipynb
Lecture Note	
No.	3

▼ 1. 클래스와 상속

▼ 1-1) 클래스



- **과자 틀 → 클래스 (Class)**
  - 클래스: 똑같은 무엇인가를 계속해서 만들어 낼 수 있는 설계 도면
- **과자 틀을 이용해서 만들어진 과자 → 객체 (Object)**
  - 객체: 클래스로 만든 피조물(예: 과자 틀을 이용해 만든 과자)
  - 특징: 객체마다 고유한 성격을 지님. 즉, 클래스로 만든 객체는 서로 영향을 주지 않음

- 스타크래프트 예시: 마린 1기, 탱크 1기 생성 및 공격

```
# 스타크래프트 예시:  마린 1기, 탱크 1기

# 마린: 공격 유닛; 군인; 총으로 공격
name = "마린" # 유닛의 이름
hp = 40 # 유닛의 체력
damage = 5 # 유닛의 공격력

print("[{0}] 유닛이 생성되었습니다.".format(name))
print("체력: {0}, 공격력: {1}\n".format(hp, damage))


# 탱크: 공격 유닛; 탱크; 포로 공격; 일반 모드/시즈 모드
tank_name = "탱크" # 유닛의 이름
tank_hp = 150 # 유닛의 체력
tank_damage = 35 # 유닛의 공격력

print("[{0}] 유닛이 생성되었습니다.".format(tank_name))
print("체력: {0}, 공격력: {1}\n".format(tank_hp, tank_damage))


# 공격에 대한 함수
def attack(name, location, damage):
  print("{0}: {1} 방향으로 적군을 공격합니다. [공격력: {2}]".format(name, location, damage))
```

```
attack(name, "1시", damage)
attack(tank_name, "1시", tank_damage)
```

- 클래스 사용하지 않는 예시: 마린 1기, 탱크 2기 생성

```
# 스타크래프트 예시(클래스 사용 X): 마린 1기, 탱크 2기

# 마린: 공격 유닛; 군인; 총으로 공격
name = "마린" # 유닛의 이름
hp = 40 # 유닛의 체력
damage = 5 # 유닛의 공격력

print("[{0}] 유닛이 생성되었습니다.".format(name))
print("체력: {0}, 공격력: {1}\n".format(hp, damage))


# 탱크1: 공격 유닛; 탱크; 포로 공격; 일반 모드/시즈 모드
tank_name = "탱크" # 유닛의 이름
tank_hp = 150 # 유닛의 체력
tank_damage = 35 # 유닛의 공격력

print("[{0}] 유닛이 생성되었습니다.".format(tank_name))
print("체력: {0}, 공격력: {1}\n".format(tank_hp, tank_damage))

#================================================================
# 탱크2: 공격 유닛; 탱크; 포로 공격; 일반 모드/시즈 모드
tank2_name = "탱크" # 유닛의 이름
tank2_hp = 150 # 유닛의 체력
tank2_damage = 35 # 유닛의 공격력

print("[{0}] 유닛이 생성되었습니다.".format(tank2_name))
print("체력: {0}, 공격력: {1}\n".format(tank2_hp, tank2_damage))
#================================================================

# 공격에 대한 함수
def attack(name, location, damage):
  print("{0}: {1} 방향으로 적군을 공격합니다. [공격력: {2}]".format(name, location, damage))


attack(name, "1시", damage)
attack(tank_name, "1시", tank_damage)
attack(tank2_name, "1시", tank2_damage)
```

- 클래스 사용하는 예시: 마린 1기, 탱크 2기 생성

```
# 스타크래프트 예시(클래스 사용 O): 마린 1기, 탱크 2기 생성

class Unit:
  def __init__(self, name, hp, damage):
    self.name = name
    self.hp = hp
    self.damage = damage

    print("[{0}] 유닛이 생성되었습니다.".format(self.name))
    print("체력: {0}, 공격력: {1}\n".format(self.hp, self.damage))

marine1 = Unit("마린", 40, 5)
tank1 = Unit("탱크", 150, 35)
tank2 = Unit("탱크", 150, 35)
```

▼ 1-2) __init__

- **생성자 (Constructor):** 객체가 생성될 때 자동으로 호출되는 메소드를 의미
  - 파이썬 메서드 이름으로 __init__을 사용하면 이 메소드는 생성자가 됨

- 객체를 생성할 때는 __init__에 정의된 전달값과 동일하게 입력해주어야 함 (self 제외)

```
# __init__에 정의된 전달값과 다르게 입력하여 객체를 생성하면 에러 발생

class Unit:
  def __init__(self, name, hp, damage):
    self.name = name
    self.hp = hp
    self.damage = damage

    print("[{0}] 유닛이 생성되었습니다.".format(self.name))
    print("체력: {0}, 공격력: {1}\n".format(self.hp, self.damage))

marine1 = Unit("마린") # Error
tank1 = Unit("탱크", 150) # Error
```

▼ 1-3) 멤버 변수

- **멤버 변수:** 클래스 내에서 정의된 변수

```
# 멤버 변수를 외부에서 접근하여 사용하기
class Unit:
  def __init__(self, name, hp, damage):
    self.name = name     # 멤버변수
    self.hp = hp         # 멤버변수
    self.damage = damage # 멤버변수

    print("[{0}] 유닛이 생성되었습니다.".format(self.name))
    print("체력: {0}, 공격력: {1}\n".format(self.hp, self.damage))

# 레이스: 공중 유닛; 비행기; 스텔스 기능
wraith1 = Unit("레이스", 80, 5)

# 새로운 멤버 변수 추가
wraith2 = Unit("추가된 레이스", 80, 5)
wraith2.clocking = True

if wraith2.clocking == True:
  print("[{0}]는 현재 스텔스 모드입니다.".format(wraith2.name))

# if wraith1.clocking == True:
#   print("{0}는 현재 스텔스 모드입니다.".format(wraith1.name))
```

▼ 1-4) 메소드

- **메소드 (Method):** 클래스 내부에 구현된 함수

```
# 메소드(클래스 내 함수)를 이용한 예제
class AttackUnit:
  def __init__(self, name, hp, damage):
    self.name = name
    self.hp = hp
    self.damage = damage

  def attack(self, location):
    print("{0}: {1} 방향으로 적군을 공격합니다. [공격력: {2}]".format(self.name, location, self.damage))

  def damaged(self, damage):
    print("{0}: {1} 데미지를 입었습니다.".format(self.name, damage))
    self.hp -= damage
    print("{0}: 현재 체력은 {1} 입니다.".format(self.name, self.hp))

    if self.hp<=0:
      print("{0}: 파괴되었습니다.".format(self.name))

# 파이어뱃 (공격유닛; 화염방사기) 생성 및 공격
firebat1 = AttackUnit("파이어뱃", 50, 16)
firebat1.attack("5시")
print()
```

```
# 파이어뱃이 공격을 2번 받음
firebat1.damaged(25)
firebat1.damaged(25)
```

▼ 1-5) 상속

- **클래스의 상속(inheritance):** 어떤 클래스를 만들 때 다른 클래스의 기능을 물려받을 수 있게 만다는 것
- **상속을 사용하는 이유:** 기존 클래스를 변경하지 않고, 기능을 추가하거나 기존 기능을 변경하려고 할 때 사용

- **클래스 상속의 예시**

```
# 일반 유닛 클래스 (부모 클래스)
class Unit:
  def __init__(self, name, hp):
    self.name = name
    self.hp = hp


# 공격 유닛 클래스 (자식 클래스)
class AttackUnit(Unit):
  def __init__(self, name, hp, damage):
    Unit.__init__(self, name, hp) # Unit 클래스 상속
    self.damage = damage

  def attack(self, location):
    print("{0}: {1} 방향으로 적군을 공격합니다. [공격력: {2}]".format(self.name, location, self.damage))

  def damaged(self, damage):
    print("{0}: {1} 데미지를 입었습니다.".format(self.name, damage))
    self.hp -= damage
    print("{0}: 현재 체력은 {1} 입니다.".format(self.name, self.hp))

    if self.hp<=0:
      print("{0}: 파괴되었습니다.".format(self.name))


# 파이어뱃 (공격유닛; 화염방사기) 생성 및 공격
firebat1 = AttackUnit("파이어뱃", 50, 16)
firebat1.attack("5시")
print()

# 파이어뱃이 공격을 2번 받음
firebat1.damaged(25)
firebat1.damaged(25)
```

▼ 1-6) 다중 상속

- **다중 상속:** 여러 부모 클래스로부터 상속 받는 것

- **다중 상속의 예시**

```
# 일반 유닛 클래스 (부모 클래스)
class Unit:
  def __init__(self, name, hp):
    self.name = name
    self.hp = hp


# 공격 유닛 클래스 (자식 클래스)
class AttackUnit(Unit):
  def __init__(self, name, hp, damage):
    Unit.__init__(self, name, hp) # Unit 클래스 상속
    self.damage = damage

  def attack(self, location):
    print("{0}: {1} 방향으로 적군을 공격합니다. [공격력: {2}]".format(self.name, location, self.damage))
```

```
  def damaged(self, damage):
    print("{0}: {1} 데미지를 입었습니다.".format(self.name, damage))
    self.hp -= damage
    print("{0}: 현재 체력은 {1} 입니다.".format(self.name, self.hp))

    if self.hp<=0:
      print("{0}: 파괴되었습니다.".format(self.name))


# 비행 기능 클래스
class Flyable:
  def __init__(self, flying_speed):
    self.flying_speed = flying_speed

  def fly(self, name, location):
    print("{0}: {1} 방향으로 날아갑니다. [비행 속도: {2}]".format(name, location, self.flying_speed))


# 공격 가능한 공중 유닛 클래스
class FlyableAttackUnit(AttackUnit, Flyable): # 다중 클래스 상속
  def __init__(self, name, hp, damage, flying_speed):
    AttackUnit.__init__(self, name, hp, damage)
    Flyable.__init__(self, flying_speed)


# 발키리: 공격 가능한 공중 유닛; 미사일 발사
valkyrie = FlyableAttackUnit("발키리", 200, 6, 5)
valkyrie.fly(valkyrie.name, "3시")
valkyrie.attack("4시")
```

▼ 1-7) 메소드 오버라이딩

- **메소드 오버라이딩(Overriding):** 부모 클래스(상속한 클래스)에 있는 메소드를 동일한 이름으로 다시 만드는 것
  - 부모 클래스의 메소드 대신 오버라이딩한 메소드가 호출

```
# 일반 유닛 클래스 (부모 클래스)
class Unit:
  def __init__(self, name, hp, speed):
    self.name = name
    self.hp = hp
    self.speed = speed # 변수 추가

  def move(self, location):
    print("[지상 유닛 이동]")
    print("{0}: {1} 방향으로 이동합니다. [이동 속도: {2}]".format(self.name, location, self.speed))


# 공격 유닛 클래스 (자식 클래스)
class AttackUnit(Unit):
  def __init__(self, name, hp, speed, damage):
    Unit.__init__(self, name, hp, speed) # Unit 클래스 상속: speed 변수 추가
    self.damage = damage

  def attack(self, location):
    print("{0}: {1} 방향으로 적군을 공격합니다. [공격력: {2}]".format(self.name, location, self.damage))

  def damaged(self, damage):
    print("{0}: {1} 데미지를 입었습니다.".format(self.name, damage))
    self.hp -= damage
    print("{0}: 현재 체력은 {1} 입니다.".format(self.name, self.hp))

    if self.hp<=0:
      print("{0}: 파괴되었습니다.".format(self.name))


# 비행 기능 클래스
class Flyable:
  def __init__(self, flying_speed):
    self.flying_speed = flying_speed

  def fly(self, name, location):
    print("{0}: {1} 방향으로 날아갑니다. [비행 속도: {2}]".format(name, location, self.flying_speed))


# 공격 가능한 공중 유닛 클래스
```

```
class FlyableAttackUnit(AttackUnit, Flyable): # 다중 클래스 상속
  def __init__(self, name, hp, damage, flying_speed):
    AttackUnit.__init__(self, name, hp, 0, damage) # 공중 유닛의 지상 speed: 0
    Flyable.__init__(self, flying_speed)

  def move(self, location):
    print("[공중 유닛 이동]")
    self.fly(self.name, location)

# 벌쳐: 공격 가능한 지상 유닛; 빠른 기동성
vulture = AttackUnit("벌쳐", 80, 10, 20) # 유닛 이름, 체력, 공격력, 이동 속도

# 배틀크루저: 공격 가능한 공중 유닛; 체력과 공격력 모두 높음
battlecruiser = FlyableAttackUnit("배틀크루저", 500, 25, 3) # 유닛 이름, 체력, 공격력, 비행 속도

# 지상 유닛의 이동(move)과 공중 유닛의 이동(fly)을 구별해야하는 불편함
vulture.move("11시")
battlecruiser.fly(battlecruiser.name, "9시")
print()

# Unit 클래스의 move 함수를 FlyableAttackUnit 클래스에 오버라이딩하여 사용
battlecruiser.move("9시")
```

▼ 1-8) pass

- **pass:** 아무 것도 수행하지 않는 문법. 임시로 코드 작성할 때 주로 사용

```
# pass 사용 예시1
class Unit:
  def __init__(self, name, hp, speed):
    self.name = name
    self.hp = hp
    self.speed = speed # 변수 추가

  def move(self, location):
    print("[지상 유닛 이동]")
    print("{0}: {1} 방향으로 이동합니다. [이동 속도: {2}]".format(self.name, location, self.speed))

# 건물
class BuildingUnit(Unit):
  def __init__(self, name, hp, location):
    pass

supply_depot = BuildingUnit("서플라이 디폿", 500, "7시")
```

```
# pass 사용 예시2
def game_start():
  print("[알림] 새로운 게임을 시작합니다.")

def game_over():
  pass

game_start()
game_over()
```

▼ 1-9) super

```
# super 사용하여 상속받는 예시
class Unit:
  def __init__(self, name, hp, speed):
    self.name = name
    self.hp = hp
    self.speed = speed # 변수 추가

  def move(self, location):
    print("[지상 유닛 이동]")
    print("{0}: {1} 방향으로 이동합니다. [이동 속도: {2}]".format(self.name, location, self.speed))

# 건물
```

```
class BuildingUnit(Unit):
  def __init__(self, name, hp, location):
    super().__init__(name, hp, 0)
    self.location = location

supply_depot = BuildingUnit("서플라이 디폿", 500, "7시")
```

```
# 다중 상속 예시
class Unit:
  def __init__(self):
    print("Unit 생성자")

class Flyable:
  def __init__(self):
    print("Flyable 생성자")

class FlyableUnit(Unit, Flyable):
# class FlyableUnit(Flyable, Unit):
  def __init__(self):
    super().__init__()
    # Unit.__init__(self)
    # Flyable.__init__(self)

dropship = FlyableUnit() # Unit 생성자: 호출 O, Flyable 생성자: 호출 X
```

▼ 1-10) 응용예제: 스타크래프트

```
from random import *

# 일반 유닛 클래스
class Unit:
  def __init__(self, name, hp, speed):
    self.name = name
    self.hp = hp
    self.speed = speed # 변수 추가
    print("[{0}] 유닛이 생성되었습니다.".format(name))

  def move(self, location):
    print("{0}: {1} 방향으로 이동합니다. [이동 속도: {2}]".format(self.name, location, self.speed))

  def damaged(self, damage):
    self.hp -= damage
    if self.hp <= 0:
      self.hp = 0

    print("{0}: {1} 데미지를 입었습니다. [현재 체력: {2}]".format(self.name, damage, self.hp))

    if self.hp == 0:
      print("{0}: 파괴되었습니다.".format(self.name))


# 공격 유닛 클래스
class AttackUnit(Unit):
  def __init__(self, name, hp, speed, damage):
    Unit.__init__(self, name, hp, speed) # Unit 클래스 상속: speed 변수 추가
    self.damage = damage

  def attack(self, location):
    print("{0}: {1} 방향으로 적군을 공격합니다. [공격력: {2}]".format(self.name, location, self.damage))

# 마린 클래스
class Marine(AttackUnit):
  def __init__(self):
    AttackUnit.__init__(self, "마린", 40, 1, 5)

    # 스팀팩: 체력 10을 사용하여 일정시간 동안 이동 및 공격 속도 증가
  def stimpack(self):
    if self.hp > 10:
      self.hp -= 10
      print("[{0}] 스팀팩을 사용합니다. (HP 10 감소)".format(self.name))
    else:
      print("[{0}] 체력이 부족하여 스팀팩을 사용할 수 없습니다.".format(self.name))


# 탱크 클래스
```

```
class Tank(AttackUnit):
  # 시즈모드: 탱크를 고정하여 더 높은 공격력으로 공격
  seize_developed = False # 시즈모드 개발 여부

  def __init__(self):
      AttackUnit.__init__(self, "탱크", 150, 1, 35)
      self.seize_mode = False

  def set_seize_mode(self):
    if Tank.seize_developed == False:
      return

    # 현재 시즈모드가 아닐 때: 시즈모드로 변경
    if self.seize_mode == False:
      print("[{0}] 시즈모드로 전환합니다.".format(self.name))
      self.damage *= 2
      self.seize_mode = True

    # 현재 시즈모드일 때: 시즈모드 해제
    else:
      print("[{0}] 시즈모드를 해제합니다.".format(self.name))
      self.damage /= 2
      self.seize_mode = False


# 비행 기능 클래스
class Flyable:
  def __init__(self, flying_speed):
    self.flying_speed = flying_speed

  def fly(self, name, location):
    print("{0}: {1} 방향으로 날아갑니다. [비행 속도: {2}]".format(name, location, self.flying_speed))


# 공격 가능한 공중 유닛 클래스
class FlyableAttackUnit(AttackUnit, Flyable): # 다중 클래스 상속
  def __init__(self, name, hp, damage, flying_speed):
    AttackUnit.__init__(self, name, hp, 0, damage) # 공중 유닛의 지상 speed: 0
    Flyable.__init__(self, flying_speed)

  def move(self, location):
    self.fly(self.name, location)


# 레이스
class Wraith(FlyableAttackUnit):
  def __init__(self):
    FlyableAttackUnit.__init__(self, "레이스", 80, 20, 5)
    self.clocked = False # 클로킹 모드 (해제 상태)

  def clocking(self):
    if self.clocked == True:
      print("[{0}] 클로킹 모드를 해제합니다.".format(self.name))
      self.clocked = False
    else:
      print("[{0}] 클로킹 모드로 전환합니다.".format(self.name))
      self.clocked = True


# 게임 관련 함수
def game_start():
  print("[알림] 새로운 게임을 시작합니다.")

def game_over():
  print("Player A: gg")
  print("[Player A] 님이 게임에서 퇴장하셨습니다.")

################################################################################
################################################################################

# 게임 시작
game_start()
print()

# 마린 3기 생성
m1 = Marine()
m2 = Marine()
m3 = Marine()

# 탱크 2기 생성
t1 = Tank()
t2 = Tank()
```

```
# 레이스 1기 생성
w1 = Wraith()

# 유닛 일괄 관리: 생성된 모든 유닛을 하나의 부대로 지정
attack_units = [] # 리스트 생성

attack_units.append(m1)
attack_units.append(m2)
attack_units.append(m3)
attack_units.append(t1)
attack_units.append(t2)
attack_units.append(w1)
print()

# 부대 이동
for unit in attack_units:
  unit.move("1시")
print()

# 탱크 시즈모드 개발
Tank.seize_developed = True
print("[알림] 탱크 시즈 모드 개발이 완료되었습니다.")
print()

# 공격 모드 준비 (마린: 스팀팩, 탱크: 시즈모드, 레이스: 클로킹)
for unit in attack_units:
  if isinstance(unit, Marine):
    unit.stimpack()
  elif isinstance(unit, Tank):
    unit.set_seize_mode()
  elif isinstance(unit, Wraith):
    unit.clocking()
print()

# 전군 공격
for unit in attack_units:
  unit.attack("1시")
print()

# 전군 피해
for unit in attack_units:
  unit.damaged(randint(5, 41)) # 5 ~ 40 사이의 공격을 랜덤으로 받음
print()

# 게임 종료
game_over()
```

▼ 2. 오류 예외처리

▼ 2-1) 예외처리

```
# 예외처리 없는 계산기 예제
print("나누기 전용 계산기입니다.")
num1 = int(input("첫 번째 숫자를 입력하세요: "))
num2 = int(input("두 번째 숫자를 입력하세요: "))
print("{0} / {1} = {2}".format(num1, num2, int(num1/num2)))
```

```
# 예외처리 적용된 계산기 예제1
try:
  print("나누기 전용 계산기입니다.")
  num1 = int(input("첫 번째 숫자를 입력하세요: "))
  num2 = int(input("두 번째 숫자를 입력하세요: "))

  print("{0} / {1} = {2}".format(num1, num2, int(num1/num2)))
except ValueError:
  print("[에러] 잘못된 값을 입력하였습니다.")
except ZeroDivisionError as err:
  print(err)
```

```
# 예외처리 적용된 계산기 예제2
try:
```

```
  print("나누기 전용 계산기입니다.")
  nums = []
  nums.append(int(input("첫 번째 숫자를 입력하세요: ")))
  nums.append(int(input("두 번째 숫자를 입력하세요: ")))
  # nums.append(int(num1/num2))
  print("{0} / {1} = {2}".format(nums[0], nums[1], nums[2]))
except ValueError:
  print("[에러] 잘못된 값을 입력하였습니다.")
except ZeroDivisionError as err:
  print(err)
except:
  print("[에러] 알 수 없는 에러가 발생하였습니다.")
```

▼ 2-2) 에러 발생시키기

```
# 의도적으로 에러 발생시켜서 예외처리하는 예제
try:
  print("한 자리 숫자 나누기 전용 계산기입니다.")
  num1 = int(input("첫 번째 숫자를 입력하세요: "))
  num2 = int(input("두 번째 숫자를 입력하세요: "))

  if num1 >= 10 or num2 >= 10:
    raise ValueError  # 에러 발생

  print("{0} / {1} = {2}".format(num1, num2, int(num1/num2)))

except ValueError:
  print("[에러] 잘못된 값을 입력하였습니다. 한 자리 숫자만 입력하세요.")
```

▼ 2-3) 사용자 정의 예외처리

```
# 사용자가 직접 정의한 에러에 대해 예외처리
class BigNumberError(Exception):
  pass

try:
  print("한 자리 숫자 나누기 전용 계산기입니다.")
  num1 = int(input("첫 번째 숫자를 입력하세요: "))
  num2 = int(input("두 번째 숫자를 입력하세요: "))

  if num1 >= 10 or num2 >= 10:
    raise BigNumberError  # 사용자 정의 에러 발생

  print("{0} / {1} = {2}".format(num1, num2, int(num1/num2)))

except ValueError:
  print("[에러] 잘못된 값을 입력하였습니다. 한 자리 숫자만 입력하세요.")
except BigNumberError:
  print("[에러] 너무 큰 수가 입력되었습니다. 한 자리 숫자만 입력하세요.")
```

```
# 사용자가 직접 정의한 에러 및 메세지에 대해 예외처리
class BigNumberError(Exception):
  def __init__(self, msg):
    self.msg = msg

  def __str__(self):
    return self.msg

try:
  print("한 자리 숫자 나누기 전용 계산기입니다.")
  num1 = int(input("첫 번째 숫자를 입력하세요: "))
  num2 = int(input("두 번째 숫자를 입력하세요: "))

  if num1 >= 10 or num2 >= 10:
    raise BigNumberError("입력값: {0}, {1}".format(num1, num2))  # 사용자 정의 에러 발생

  print("{0} / {1} = {2}".format(num1, num2, int(num1/num2)))

except ValueError:
```

```
  print("[에러] 잘못된 값을 입력하였습니다. 한 자리 숫자만 입력하세요.")
except BigNumberError as err:
  print("[에러] 너무 큰 수가 입력되었습니다. 한 자리 숫자만 입력하세요.")
  print(err)
```

▼ 2-4) finally

- **finally:** try문 수행 도중 예외 발생 여부에 상관없이 항상 수행
  - 보통 사용한 리소스를 close 할 때 많이 사용

```
# 사용자가 직접 정의한 에러 및 메세지에 대해 예외처리
class BigNumberError(Exception):
  def __init__(self, msg):
    self.msg = msg

  def __str__(self):
    return self.msg

try:
  print("한 자리 숫자 나누기 전용 계산기입니다.")
  num1 = int(input("첫 번째 숫자를 입력하세요: "))
  num2 = int(input("두 번째 숫자를 입력하세요: "))

  if num1 >= 10 or num2 >= 10:
    raise BigNumberError("입력값: {0}, {1}".format(num1, num2))  # 사용자 정의 에러 발생

  print("{0} / {1} = {2}".format(num1, num2, int(num1/num2)))

except ValueError:
  print("[에러] 잘못된 값을 입력하였습니다. 한 자리 숫자만 입력하세요.")
except BigNumberError as err:
  print("[에러] 너무 큰 수가 입력되었습니다. 한 자리 숫자만 입력하세요.")
  print(err)
finally:
  print("계산기를 이용해주셔서 감사합니다.")
```

▼ 3. 모듈과 패키지

▼ 3-1) 모듈

- **모듈:** 함수나 변수 또는 클래스를 모아 놓은 파이썬 파일(.py)

```
# Google drive 연결
from google.colab import drive
drive.mount('/content/gdrive')

# module 파일이 있는 경로 추가
import sys
sys.path.append('/content/gdrive/MyDrive/Colab Notebooks/C2-004_Python/')
```

```
# 모듈 사용의 다양한 예시1
import module_theater

module_theater.price(3)
module_theater.price_morning(4)
module_theater.price_army(5)
```

```
# 모듈 사용의 다양한 예시2
import module_theater as mv

mv.price(3)
mv.price_morning(4)
mv.price_army(5)
```

```
# 모듈 사용의 다양한 예시3
from module_theater import *

price(3)
price_morning(4)
price_army(5)
```

```
# 모듈 사용의 다양한 예시4
from module_theater import price, price_army

price(3)
price_army(5)
# price_morning(4)
```

▼ 3-2) 패키지

- **패키지(Package):** 도트(.)를 사용하여 파이썬 모듈을 계층적(디렉토리 구조)으로 관리할 수 있게 함.
  - 모듈 이름이 A.B인 경우, A는 패키지 이름이 되고, B는 A 패키지의 B 모듈

```
# 패키지 사용 예제1
import travel.france

trip_to = travel.france.FrancePackage()
trip_to.detail()
```

```
# 패키지 사용 예제2
from travel.france import FrancePackage

trip_to = FrancePackage()
trip_to.detail()
```

```
# 패키지 사용 예제3
from travel import switzerland

trip_to = switzerland.SwitzerlandPackage()
trip_to.detail()
```

▼ 3-3) __all__

- **__all__:** 특정 디렉토리의 모듈을 *을 사용하여 import 할 때에는 해당 디렉토리의 __init__.py 파일 내 __all__ 변수를 설정하고 import 할 수 있는 모듈을 정의해야 함

```
# from random import * 유사한 사용 예제
from travel import *

trip_to = FrancePackage()
trip_to.detail()
```

```
# from random import * 유사한 사용 예제
from travel import *

trip_to = FrancePackage()
trip_to.detail()

trip_to = switzerland.SwitzerlandPackage()
trip_to.detail()
```

▼ 3-4) 모듈 직접 실행

```
# travel 폴더 내 france.py 모듈을 직접 실행
import os
os.chdir('/content/gdrive/MyDrive/Colab Notebooks/C2-004_Python/travel')

%run france.py
```

▼ 3-5) 패키지, 모듈 위치

- **inspect:** 패키지와 모듈의 위치 확인

```
# inspect 를 이용한 패키지/모듈 위치 확인 예제1
import inspect
import random

print(inspect.getfile(random))
```

```
# inspect 를 이용한 패키지/모듈 위치 확인 예제2
import inspect
import france

print(inspect.getfile(france))
```

▼ 3-6) pip install

- pip install 을 이용하여 이미 잘 만들어진 패키지를 설치

PyPI · The Python Package Index

The Python Package Index (PyPI) is a repository of software for the Python programming language. PyPI helps you find and install software developed and shared by the Python community. Learn about installing packages. Package authors use PyPI to distribute their

https://pypi.org/

▼ 3-7) 내장함수

Built-in Functions - Python 3.10.6 documentation

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

https://docs.python.org/3/library/functions.html

▼ 3-8) 외장함수

Python Module Index

Deprecated: Helpers for running Python scripts via the Common Gateway Interface.

https://docs.python.org/3/py-modindex.html