

객체 지향 프로그래밍

목차

1. 객체 지향 프로그래밍의 이해
2. 파이썬의 객체 지향 프로그래밍
3. Lab: 노트북 프로그램 만들기
4. 객체 지향 프로그래밍의 특징

01

객체 지향 프로그래밍의 이해

01. 객체 지향 프로그래밍의 이해

■ 객체 지향 프로그래밍을 배우는 이유

- 객체 지향 프로그래밍(Object Oriented Programming, OOP)은 함수처럼 어떤 기능을 함수 코드에 묶어 두는 것이 아니라, 그런 기능을 묶은 하나의 단일 프로그램을 객체라고 하는 코드에 넣어 다른 프로그래머가 재사용할 수 있도록 하는, 컴퓨터 공학의 오래된 프로그래밍 기법 중 하나이다.

01. 객체 지향 프로그래밍의 이해

■ 객체와 클래스

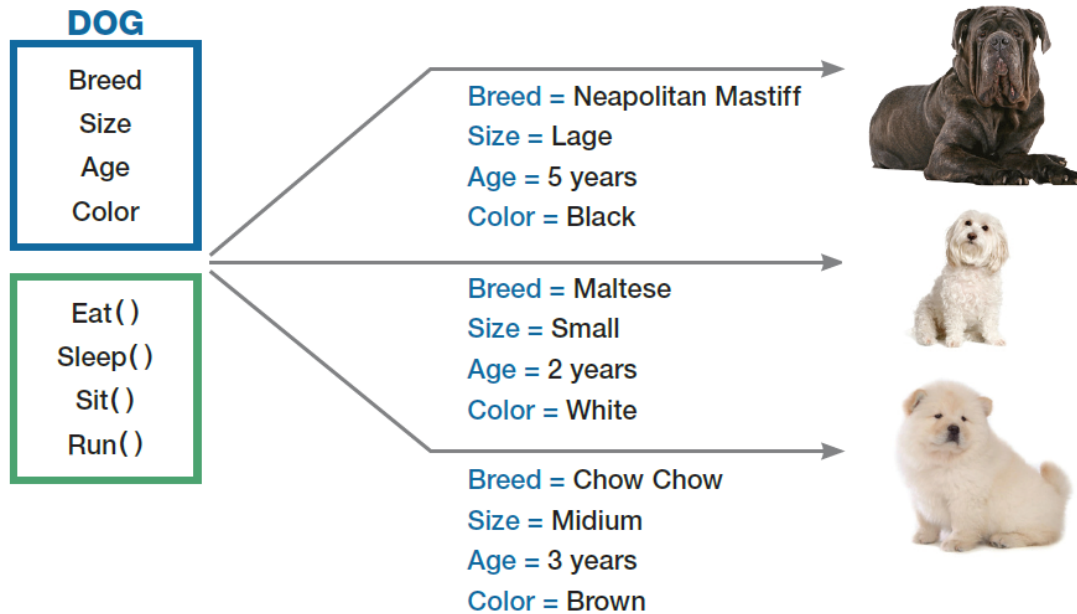
개념	설명	예시
객체(object)	실생활에 존재하는 실제적인 물건 또는 개념	심판, 선수, 팀
속성(attribute)	객체가 가지고 있는 변수	선수의 이름, 포지션, 소속팀
행동(action)	객체가 실제로 작동할 수 있는 함수, 메서드	공을 차다, 패스하다

[객체, 속성, 행동]

01. 객체 지향 프로그래밍의 이해

■ 객체와 클래스

- 클래스(class) : 객체가 가져야 할 기본 정보를 담은 코드이다.
- 클래스는 일종의 설계도 코드이다.
- 실제로 생성되는 객체를 인스턴스(instance)라고 한다.



[Dog 인스턴스]

01. 객체 지향 프로그래밍의 이해

■ 객체와 클래스

- 잘 만든 붕어빵틀이 있다면 새로운 종류의 다양한 붕어빵을 만들 수 있다.
- 다시 말해, 잘 만든 클래스 코드가 있다면 이 코드로 다양한 종류의 인스턴스를 생성할 수 있다.



[클래스와 인스턴스의 관계]

02

파이썬의 객체 지향 프로그래밍

02. 파이썬의 객체 지향 프로그래밍

■ 클래스 구현하기

- 파이썬에서 클래스를 선언하기 위한 기본 코드 템플릿은 다음과 같다.

class SoccerPlayer(object):

클래스 예약어 클래스 이름 상속받는 객체명

[파이썬에서의 클래스 선언]

- 먼저 예약어인 class를 코드의 앞에 쓰고, 만들고자 하는 클래스 이름을 작성한다.
- 그 다음으로 상속받아야 하는 다른 클래스의 이름을 괄호 안에 넣는다.

02. 파이썬의 객체 지향 프로그래밍

여기서 잠깐! 파이썬에서 자주 사용하는 작명 기법

- 클래스의 이름을 선언할 때 한 가지 특이한 점은 기존과 다르게 **첫 글자와 중간 글자가 대문자**라는 것이다. 이것은 클래스를 선언할 때 사용하는 작명 기법에 의해 생성된다. 파이썬뿐 아니라 모든 컴퓨터 프로그래밍 언어에서는 변수, 클래스, 함수명을 짓는 작명 기법이 있다. 아래 표는 프로그래머가 흔히 사용하는 두 가지 작명 기법이다.

작명 기법	설명
snake_case	띄어쓰기 부분에 '_'를 추가하여 변수의 이름을 지정함, 파이썬 함수나 변수명에 사용됨
CamelCase	띄어쓰기 부분에 대문자를 사용하여 변수의 이름을 지정함, 낙타의 혹처럼 생겼다하여 Camel 이라고 명명, 파이썬 클래스명에 사용됨

[파이썬에서 자주 사용하는 작명 기법]

02. 파이썬의 객체 지향 프로그래밍

■ 클래스 구현하기 : 속성의 선언

- 속성에 대한 정보를 선언하기 위해서는 `__init__()`이라는 예약 함수를 사용한다.

```
class SoccerPlayer(object):  
    def __init__(self, name, position, back_number):  
        self.name = name  
        self.position = position  
        self.back_number = back_number
```

- ➡ `__init__()` 함수는 이 class에서 사용할 변수를 정의하는 함수이다. `__init__()` 함수의 첫 번째 매개변수는 반드시 `self` 변수를 사용해야 한다. `self` 변수는 클래스에서 생성된 인스턴스에 접근하는 예약어이다.
- ➡ `self` 뒤의 매개변수들은 실제로 클래스가 가진 속성으로, 축구 선수의 이름, 포지션, 등번호 등이다. 이 값들은 실제 생성된 인스턴스에 할당된다. 할당되는 코드는 `self.name = name` 이다.

02. 파이썬의 객체 지향 프로그래밍

■ 클래스 구현하기 : 함수의 선언

- 함수는 이 클래스가 의미하는 어떤 객체가 하는 다양한 동작을 정의할 수 있다. 만약 축구 선수라면, 등번호 교체라는 행동을 할 수 있고, 이를 다음과 같은 코드로 표현할 수 있다.

```
class SoccerPlayer(object):  
    def change_back_number(self, new_number):  
        print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_number))  
        self.back_number = new_number
```

- 클래스 내에서의 함수도 기존 함수와 크게 다르지 않다. 함수의 이름을 쓰고 매개변수를 사용하면 된다. 여기서 가장 큰 차이점은 바로 `self`를 매개변수에 반드시 넣어야 한다는 것이다. `self`가 있어야만 실제로 인스턴스가 사용할 수 있는 함수로 선언된다.

02. 파이썬의 객체 지향 프로그래밍

■ 클래스 구현하기 : _의 쓰임

- 일반적으로 파이썬에서 _의 쓰임은 개수에 따라 여러 가지로 나눌 수 있다. 예를 들어, _ 1개는 **이후로 쓰이지 않을 변수에 특별한 이름을 부여하고 싶지 않을 때** 사용한다

코드 10-1 underscore.py

```
1 for _ in range(10):
2     print("Hello, World")
```

[illegible]

02. 파이썬의 객체 지향 프로그래밍

■ 클래스 구현하기 : _의 쓰임

- ➔ 위 코드는 'Hello, World'를 화면에 10번 출력하는 함수이다. **횟수를 세는 _ 변수는 특별한 용도가 없으므로 뒤에서 사용되지 않는다. 따라서 _를 임의의 변수명 대신에 사용한다.**
- 다른 용도로는 _ 2개를 사용하여 특수한 **예약 함수나 변수**에 사용하기도 한다. 대표적으로 **__str__이나 __init__()** 같은 함수이다. **__str__()** 함수는 클래스로 인스턴스를 생성했을 때, 그 인스턴스 자체를 **print()** 함수로 화면에 출력하면 나오는 값을 뜻한다. 다양한 용도가 있으니 **_**의 특수한 용도에 대해서는 인지해 두는 것이 좋다

02. 파이썬의 객체 지향 프로그래밍

■ 인스턴스 사용하기

- 클래스에서 인스턴스를 호출하는 방법은 아래 그림과 같다. 먼저 클래스 이름을 사용하여 호출하고, 앞서 만든 `__init__()` 함수의 매개변수에 맞추어 값을 입력한다. 여기에서는 함수에서 배운 초깃값 지정 등도 사용할 수 있다. 여기서 `self` 변수는 아무런 값도 할당되지 않는다.
- `jinhyun`이라는 인스턴스가 기존 `SoccerPlayer`의 클래스를 기반으로 생성되는 것을 확인할 수 있다. 이 `jinhyun`이라는 인스턴스 자체가 `SoccerPlayer` 클래스에서 `self`에 할당된다.

`jinhyun` = `SoccerPlayer`(`"Jinhyun"`, `"MF"`, `10`):

객체명

클래스 이름

`__init__` 함수 Interface, 초깃값

```
def __init__(self, name, position, back_number);
```

[파이썬에서 자주 사용하는 작명 기법]

02. 파이썬의 객체 지향 프로그래밍

■ 인스턴스 사용하기

- 실제로 생성된 코드는 다음 [코드 10-2]와 같다.


코드 10-2 instance.py

```
1  # 전체 SoccerPlayer 코드
2  class SoccerPlayer(object):
3      def __init__(self, name, position, back_number):
4          self.name = name
5          self.position = position
6          self.back_number = back_number
7      def change_back_number(self, new_number):
8          print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_
          number))
9          self.back_number = new_number
10     def __str__(self):
11         return "Hello, My name is %. I play in %s in center." % (self.name,
            self.position)
12
```


02. 파이썬의 객체 지향 프로그래밍

■ 인스턴스 사용하기

```
13 # SoccerPlayer를 사용하는 instance 코드
14 jinhyun = SoccerPlayer("Jinhyun", "MF", 10)
15
16 print("현재 선수의 등번호는:", jinhyun.back_number)
17 jinhyun.change_back_number(5)
18 print("현재 선수의 등번호는:", jinhyun.back_number)
```



```
현재 선수의 등번호는: 10          ← 16행 실행 결과
선수의 등번호를 변경한다: From 10 to 5 ← 17행 실행 결과
현재 선수의 등번호는: 5          ← 18행 실행 결과
```

- ➡ 14행에서 `jinhyun = SoccerPlayer("Jinhyun", "MF", 10)` 코드로 인스턴스를 새롭게 생성할 수 있다. 생성된 인스턴스인 `jinhyun`은 `name`, `position`, `back_number`에 각각 `Jinhyun`, `MF`, `10`이 할당되었다. 해당 값을 생성된 인스턴스에서 사용하기 위해서는 `jinhyun.back_number`로 인스턴스 내의 값을 호출하여 사용할 수 있다.

02. 파이썬의 객체 지향 프로그래밍

■ 인스턴스 사용하기

- ➔ 여기서 중요한 것은 인스턴스가 생성된 후에는 해당 인스턴스의 이름으로 값을 할당하거나 함수를 부르면 되지만, 클래스 내에서는 self로 호출된다. 즉, 생성된 인스턴스인 jinhyun과 클래스 내 self가 같은 역할을 하는 것이다.

함수를 호출할 때도 인스턴스의 이름과 함수명을 사용한다. 여기서는 17행에서 jinhyun.change_back_number(5)를 사용해 클래스 내의 함수를 사용하였다.

02. 파이썬의 객체 지향 프로그래밍

■ 인스턴스 사용하기

- [코드 10-2]에 이어 19행에 `print(jinhyun)`을 입력하면 다음과 같은 결과가 출력된다.



```
Hello, My name is Jinhyun. I play in MF in center.
```

- ➡ 생성된 인스턴스인 `jinhyun`을 단순히 `print()` 함수로 썼을 때 나오는 결과이다. 이는 [코드 10-2]의 2 · 10 · 11행에 클래스 내 함수가 선언되었기 때문이다.

9행에서 `__str__` 함수로 선언된 부분이 `print()` 함수를 사용하면 반환되는 함수이다. 인스턴스의 정보를 표시하거나 구분할 때는 `__str__` 문을 사용하면 된다. 이처럼 예약 함수는 특정 조건에서 작동하는 함수로 유용하다

02. 파이썬의 객체 지향 프로그래밍

여기서 잠깐! Hydrogen 패키지

- Atom에서 [코드 10-2]의 실행 결과를 확인하려면 Hydrogen 패키지를 사용하면 된다. Hydrogen 패키지를 사용하면 코드를 작성한 후 결과를 확인하기 위해 일일이 cmd 창으로 가지 않아도 되어 편리하고, 출력 결과가 어떤 코드로 인해 발생한 것인지 직접 확인할 수 있어 매우 유용하다.

02. 파이썬의 객체 지향 프로그래밍

■ 클래스를 사용하는 이유

- 자신이 만든 코드가 데이터 저장뿐 아니라 데이터를 변환하거나 데이터베이스에 저장하는 등의 역할이 필요할 때가 있다. 이것을 리스트와 함수로 각각 만들어 공유하는 것보다 하나의 객체로 생성해 다른 사람들에게 배포한 다면 훨씬 더 손쉽게 사용할 수 있을 것이다.
- 또한, 코드를 좀 더 손쉽게 선언할 수 있다는 장점도 있다.

코드 10-3 class.py

```
1 # 데이터
2 names = ["Messi", "Ramos", "Ronaldo", "Park", "Buffon"]
3 positions = ["MF", "DF", "CF", "WF", "GK"]
4 numbers = [10, 4, 7, 13, 1]
5
6 # 이차원 리스트
7 players = [[name, position, number] for name, position, number in zip(names,
    positions, numbers)]
8 print(players)
9 print(players[0])
10
```

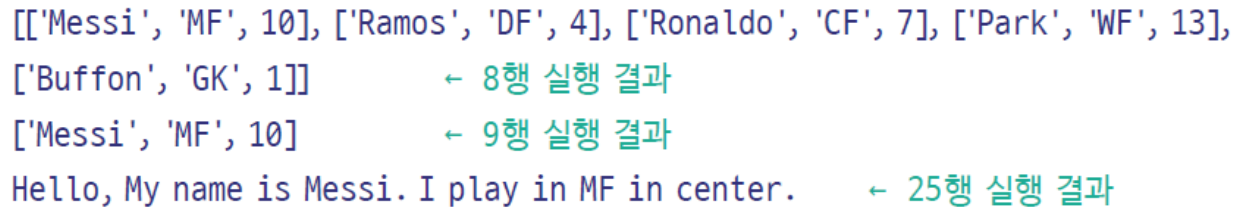
02. 파이썬의 객체 지향 프로그래밍

■ 클래스를 사용하는 이유

```
11 # 전체 SoccerPlayer 코드
12 class SoccerPlayer(object):
13     def __init__(self, name, position, back_number):
14         self.name = name
15         self.position = position
16         self.back_number = back_number
17     def change_back_number(self, new_number):
18         print("선수의 등번호를 변경한다: From %d to %d" % (self.back_number, new_
19             number))
20         self.back_number = new_number
21     def __str__(self):
22         return "Hello, My name is %. I play in %s in center." % (self.name,
23             self.position)
24 # 클래스-인스턴스
25 player_objects = [SoccerPlayer(name, position, number) for name, position,
26     number in zip(names, positions, numbers)]
27 print(player_objects[0])
```

02. 파이썬의 객체 지향 프로그래밍

■ 클래스를 사용하는 이유



```
[[ 'Messi', 'MF', 10], [ 'Ramos', 'DF', 4], [ 'Ronaldo', 'CF', 7], [ 'Park', 'WF', 13],  
[ 'Buffon', 'GK', 1]]      ← 8행 실행 결과  
[ 'Messi', 'MF', 10]      ← 9행 실행 결과  
Hello, My name is Messi. I play in MF in center.      ← 25행 실행 결과
```

03

Lab: 노트북 프로그램 만들기

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 만들기 설계

- 이번 Lab에서는 지금까지 배운 객체 지향 프로그래밍의 개념을 이용하여 실제 구현 가능한 노트북(Notebook) 프로그램을 만들고자 한다.

- 노트(note)를 정리하는 프로그램이다.
- 사용자는 노트에 콘텐츠를 적을 수 있다.
- 노트는 노트북(notebook)에 삽입된다.
- 노트북은 타이틀(title)이 있다.
- 노트북은 노트가 삽입될 때 페이지를 생성하며, 최대 300페이지까지 저장할 수 있다.
- 300페이지를 넘기면 노트를 더는 삽입하지 못한다.

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 만들기 설계

구분	Notebook	Note
메서드	add_note remove_note get_number_of_pages	write_content remove_all
변수	title page_number notes	contents

[노트북 프로그램의 객체 설계]

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 구현 : Note 클래스

```
class Note(object):  
    def __init__(self, contents = None):  
        self.contents = contents  
  
    def write_contents(self, contents):  
        self.contents = contents  
  
    def remove_all(self):  
        self.contents = ""  
  
    def __str__(self):  
        return self.contents
```

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 구현 : Note 클래스

➔ Note 클래스에서 가장 중요한 변수는 contents, 즉 내용을 적는 변수이다. 이를 위해 Note 객체를 생성할 때 contents의 내용을 넣을 수 있도록 `__init__()` 함수 안에 `self.contents`를 초기값으로 만든다. Note는 Notebook에 있는 여러 장의 노트 중 한 장이라고 생각 할 수 있다. 당연히 새로운 Note를 만들고 아무런 내용도 넣지 않을 수 있으므로, 초기값을 `contents = None`으로 한다.

다음으로 Note에 새로운 내용을 쓰는 `write_contents()`와 Note의 모든 내용을 지우는 `remove_all()` 함수를 만든다. 각각은 문자열형을 입력받은 후, `self.contents` 변수에 할당하거나 내용을 삭제하는 `self.contents = ""`를 호출한다.

마지막으로 `print()` 함수를 유용하게 사용하기 위해 `__str__`을 선언하여 contents를 반환한다.

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 구현 : Notebook 클래스

```
class NoteBook(object):
    def __init__(self, title):
        self.title = title
        self.page_number = 1
        self.notes = {}

    def add_note(self, note, page = 0):
        if self.page_number < 300:
            if page == 0:
                self.notes[self.page_number] = note
                self.page_number += 1
            else:
                self.notes = {page : note}
                self.page_number += 1
        else:
            print("페이지가 모두 채워졌다.")
```

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 구현 : Notebook 클래스

```
def remove_note(self, page_number):
    if page_number in self.notes.keys():
        return self.notes.pop(page_number)
    else:
        print("해당 페이지는 존재하지 않는다.")

def get_number_of_pages(self):
    return len(self.notes.keys())
```

➡ Notebook 클래스에는 다음 세 가지가 필요하다.

- ① 타이틀(title): Notebook의 제목이 필요하다.
- ② 페이지 수(page_number): 현재 Notebook에 총 몇 장의 노트가 있는지 기록하는 page_number가 필요하며, 1페이지부터 시작한다.
- ③ 저장 공간: 노트 자체를 저장하기 위한 공간이 필요하다.

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 구현 : Notebook 클래스

- ➡ 이 세 가지 정보를 저장하기 위해 `__init__()` 함수 안에 정보를 모두 입력하였다. 여기서 관심을 두어야 할 변수는 `self.notes`이다. 위 코드에서는 `notes` 변수를 딕셔너리형으로 선언하였다. 이는 클래스를 설계하는 사람이 자신의 기호에 맞게, 또는 개발 목적에 따라 적절히 지정할 수 있다. 여기서 `notes` 변수 안에는 `Note`의 인스턴스가 값value으로 들어가게 하고, 키로 각 `Note`의 `page_number`를 사용할 예정이다. 이는 `page_number`로 `Note`를 쉽게 찾기 위해서다.
- ➡ 다음으로는 함수 부분이다. 먼저 `add_note()` 함수는 새로운 `Note`를 `Notebook`에 삽입하는 함수이다. 몇 가지 요구 조건에 대한 로직이 들어간다. 예를 들어, `page_number`가 300이하이면 새로운 `Note`를 계속 추가할 수 있지만, 그 이상일 때는 `Note`를 더 추가하지 못하도록 한다. 새로운 `Note`가 들어갈 때는 임의의 페이지 번호를 넣을 수 있다. 하지만 사용자가 입력하지 않을 때는 맨 마지막에 입력된 `Note` 다음 장에 `Note`를 추가한다. 맨 마지막 페이지는 늘 `page_number`에 저장되어 있다.

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 구현 : Notebook 클래스

- ➡ 두 번째 함수는 `remove_note()` 함수이다. `remove_note()` 함수는 특정 페이지 번호에 있는 Note를 제거하는 함수이다. 앞에서 Note가 저장된 객체를 딕셔너리형으로 저장하고 페이지 번호를 키로 저장했으므로, 딕셔너리형인 `self.notes`에 해당 페이지 번호의 키가 있는지 확인하면 된다. 이는 `page_number in self.notes.keys()`로 쉽게 확인할 수 있다. 만약 페이지가 있다면 해당 페이지를 삭제하면서 반환하고, 없다면 `print()` 함수를 사용하여 없다고 사용자에게 알려 줄 수 있다.
- ➡ `page_number`를 넣고 그 페이지가 기존의 노트에 있다면 해당 페이지를 팝하여 돌려주고, 없다면 '해당 페이지는 존재하지 않는다.'라고 출력한다

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 사용

- 앞에서 구현한 Note 클래스와 Notebook 클래스를 활용하여 실제 프로그램을 작성해 보자. 소스 파일에서 제공하는 'notebook.py'와 'notebook_client.py' 파일을 활용한다.
- 먼저 해당 클래스를 불러 사용할 수 있는 클라이언트 코드를 만들어 보자. 지금부터 작성하는 코드는 'notebook_client.py'에 있는 코드이다. 먼저 2개의 클래스를 호출해야 하는데, 호출하는 코드는 내장 모듈을 사용한 것처럼 from과 import를 사용한다. 'from 파일명 import 클래스명'의 형태로 생각하면 된다. 이 코드가 실행되면 두 클래스는 메모리에 로딩된다

```
from notebook import Note
from notebook import Notebook
```

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 사용

- 다음으로 몇 장의 Note를 생성한다. 넣고 싶은 Note의 내용과 함께 문자열형을 사용하여 생성자로 만들면 된다.

```
good_sentence = """"세상 사는 데 도움이 되는 명언, 힘이 되는 명언, 용기를 주는 명언, 위로되는 명언, 좋은 명언 모음 100가지. 자주 보면 좋을 것 같아 선별했습니다."""
```

```
note_1 = Note(good_sentence)
```

```
good_sentence = """"삶이 있는 한 희망은 있다. - 키케로 """"
```

```
note_2 = Note(good_sentence)
```

```
good_sentence = """"하루에 3시간을 걸으면 7년 후에 지구를 한 바퀴 돌 수 있다. - 새뮤얼 존슨""""
```

```
note_3 = Note(good_sentence)
```

```
good_sentence = """"행복의 문이 하나 닫히면 다른 문이 열린다. 그러나 우리는 종종 닫힌 문을 멍하니 바라보다가 우리를 향해 열린 문을 보지 못하게 된다. - 헬렌 켈러""""
```

```
note_4 = Note(good_sentence)
```

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 사용

- 모두 4개의 Note이다. 이 Note를 파이썬 셸에서 확인하기 위해 다음과 같이 코드를 입력하면, Note의 인스턴스 생성을 확인할 수 있다. Note를 생성한 후 `print()` 함수를 사용하면 해당 Note에 있는 텍스트를 확인할 수 있다. 또한, `remove()` 함수도 사용할 수 있다.

```
>>> from notebook import Note
>>> from notebook import Notebook
>>> good_sentence = """세상 사는 데 도움이 되는 명언, 힘이 되는 명언, 용기를 주는 명언, 위로
>>> 되는 명언, 좋은 명언 모음 100가지. 자주 보면 좋을 것 같아 선별했습니다."""
>>> note_1 = Note(good_sentence)
>>>
>>> note_1
<notebook.Note object at 0x0000022278C06DD8>
>>> print(note_1)
세상 사는 데 도움이 되는 명언, 힘이 되는 명언, 용기를 주는 명언, 위로되는 명언, 좋은 명언 모음
100가지. 자주 보면 좋을 것 같아 선별했습니다.
>>> note_1.remove()
>>> print(note_1)
삭제된 노트입니다.
```

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 사용

- 다음은 새로운 Notebook을 생성하는 코드이다. 새로운 노트를 생성한 후 기존의 Note들을 add_note() 함수로 추가하였다.

```
wise_saying_notebook = Notebook("명언 노트")
wise_saying_notebook.add_note(note_1)
wise_saying_notebook.add_note(note_2)
wise_saying_notebook.add_note(note_3)
wise_saying_notebook.add_note(note_4)
```

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 사용

- 다음 코드처럼 실제 추가된 것을 확인할 수 있다. Note 4장을 추가하였으므로 `get_number_of_all_pages()`를 사용하면 총 페이지 수가 출력되고, `get_number_of_all_characters()`로 총 글자 수를 확인할 수 있다.

```
>>> wise_saying_notebook = Notebook("명언 노트")
>>> wise_saying_notebook.add_note(note_1)
>>> wise_saying_notebook.add_note(note_2)
>>> wise_saying_notebook.add_note(note_3)
>>> wise_saying_notebook.add_note(note_4)
>>> print(wise_saying_notebook.get_number_of_all_pages())
4
>>> print(wise_saying_notebook.get_number_of_all_characters())
159
```

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 사용

- 또한, 노트의 삭제나 추가도 여러 가지 명령어로 가능하다. 다음과 같이 특정 Note를 지우는 `remove_note()`를 사용할 수 있고, 객체 지향 프로그래밍을 사용하여 새로운 빈 노트를 임의로 추가할 수도 있다. 기존 페이지에 노트를 추가하려고 하면 오류 메시지도 출력된다

```
>>> wise_saying_notebook.remove_note(3)
>>> print(wise_saying_notebook.get_number_of_all_pages())
3
>>>
>>> wise_saying_notebook.add_note(note_1, 100)
>>> wise_saying_notebook.add_note(note_1, 100)
해당 페이지에는 이미 노트가 존재합니다.
>>>
>>> for i in range(300):
...     wise_saying_notebook.add_note(note_1, i)
... 
```

03. Lab: 노트북 프로그램 만들기

■ 노트북 프로그램 사용

해당 페이지에는 이미 노트가 존재합니다.

해당 페이지에는 이미 노트가 존재합니다.

해당 페이지에는 이미 노트가 존재합니다.

해당 페이지에는 이미 노트가 존재합니다.

```
>>> print(wise_saying_notebook.get_number_of_all_pages())
```

```
300
```

04

객체 지향 프로그래밍의 특징

04. 객체 지향 프로그래밍의 특징

■ 상속 (Inheritance)

- 상속(inheritance) 은 이름 그대로 무엇인가를 내려받는 것을 뜻한다. 부모 클래스에 정의된 속성과 메서드를 자식 클래스가 물려받아 사용하는 것이다.

```
class Person(object):  
    pass
```

- ➡ class라는 예약어 다음에 클래스명으로 Person을 쓰고 object를 입력하였다. 여기서 object가 바로 Person 클래스의 부모 클래스이다. 사실 **object**는 파이썬에서 사용하는 가장 기본 객체(base object) 이며, 파이썬 언어가 객체 지향 프로그래밍이므로 모든 변수는 객체이다. 예를 들어, 파이썬의 문자열형은 다음과 같이 객체 이름을 확인할 수 있다.

```
>>> a = "abc"  
>>> type(a)  
<class 'str'>
```

04. 객체 지향 프로그래밍의 특징

■ 상속

```
>>> class Person(object):
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> class Korean(Person):
...     pass
...
>>> first_korean = Korean("Sungchul", 35)
>>> print(first_korean.name)
Sungchul
```

04. 객체 지향 프로그래밍의 특징

■ 상속

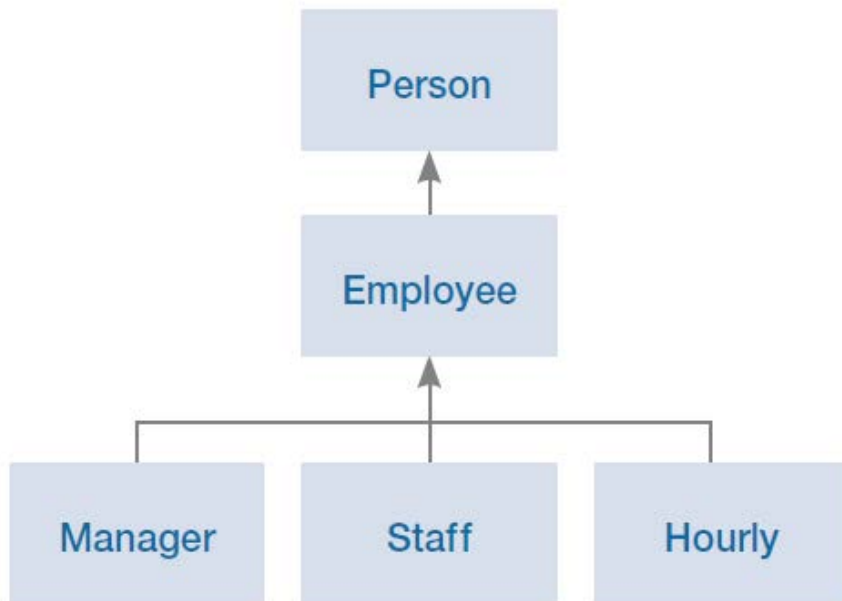
```
>>> class Person(object):
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> class Korean(Person):
...     pass
...
>>> first_korean = Korean("Sungchul", 35)
>>> print(first_korean.name)
Sungchul
```

- ➔ 위 코드에서는 먼저 Person 클래스를 생성하였다. Person 클래스에는 생성자 `__init__()` 함수를 만들어 name과 age에 관련된 정보를 입력할 수 있도록 하였다. 다음으로 Korean 클래스를 만들면서 Person 클래스를 상속받게 한다. 상속은 `class Korean(Person)` 코드처럼 매우 간단하다. 그리고 별도의 구현 없이 **pass**로 클래스만 존재하게 만들고, Korean 클래스의 인스턴스를 생성해 준다. Korean 클래스는 별도의 생성자가 없지만, Person 클래스가 가진 생성자를 그대로 사용하여 인스턴스를 생성할 수 있다. 그리고 Person 클래스에서 생성할 수 있는 변수를 그대로 사용할 수 있다. 이러한 객체 지향 프로그래밍의 특징을 상속이라고 한다.

04. 객체 지향 프로그래밍의 특징

■ 상속

- 사각형이 클래스이고, 화살표는 각 클래스의 상속 관계이다. Person 클래스를 Employee가 상속하고, 이 클래스를 다시 한번 Manager, Staff, Hourly 등이 상속하는 것이다.



[상속 구조]

04. 객체 지향 프로그래밍의 특징

■ 상속

- 상속이 진행될수록 부모 클래스에 대해 각 클래스의 기능이 구체화되도록 **부모 객체에는 일반적인 기능을, 자식 객체에는 상세한 기능을** 넣어야 한다. 그리고 같은 일을 하는 메서드이지만 부모 객체보다 자식 객체가 좀 더 많은 정보를 줄 수도 있다. 이를 '**부모 클래스의 메서드를 재정의한다**'라고 한다.

코드 10-4 inheritance1.py

```
1 class Person(object):                                # 부모 클래스 Person 선언
2     def __init__(self, name, age, gender):
3         self.name = name
4         self.age = age
5         self.gender = gender
6
7     def about_me(self):                                # 메서드 선언
8         print("저의 이름은", self.name, "이고요, 제 나이는", str(self.age), "살입니다.")
```

04. 객체 지향 프로그래밍의 특징

■ 상속

코드 10-4 inheritance1.py

```
1 class Person(object):                                # 부모 클래스 Person 선언
2     def __init__(self, name, age, gender):
3         self.name = name
4         self.age = age
5         self.gender = gender
6
7     def about_me(self):                                # 메서드 선언
8         print("저의 이름은", self.name, "이고요, 제 나이는", str(self.age), "살입니다.")
```

- ➡ 먼저 부모 클래스 Person이다. name, age, gender에 대해 변수를 받을 수 있도록 선언하였고, about_me 함수를 사용하여 생성된 인스턴스가 자신을 설명할 수 있도록 하였다. 사실 str()에 들어가도 되는 클래스이지만 임의의 about_me 클래스를 생성하였다.

04. 객체 지향 프로그래밍의 특징

■ 상속

- 다음으로 상속받는 Employee 클래스는 [코드 10-5]와 같다.

코드 10-5 inheritance2.py

```
1 class Employee(Person):                                # 부모 클래스 Person으로부터 상속
2     def __init__(self, name, age, gender, salary, hire_date):
3         super().__init__(name, age, gender)             # 부모 객체 사용
4         self.salary = salary
5         self.hire_date = hire_date                      # 속성값 추가
6
7     def do_work(self):                                   # 새로운 메서드 추가
8         print("열심히 일을 한다.")
9
10    def about_me(self):                                  # 부모 클래스 함수 재정의
11        super().about_me()                              # 부모 클래스 함수 사용
12        print("제 급여는", self.salary, "원이고, 제 입사일은", self.hire_date, "입니다.")
```

코드 10-4 inheritance1.py

```
1 class Person(object):                                  # 부모 클래스 Person 선언
2     def __init__(self, name, age, gender):
3         self.name = name
4         self.age = age
5         self.gender = gender
6
7     def about_me(self):                                  # 메서드 선언
8         print("저의 이름은", self.name, "이고요, 제 나이는", str(self.age), "살입니다.")
```

04. 객체 지향 프로그래밍의 특징

■ 상속

- ➔ Person 클래스가 단순히 사람에 대한 정보를 정의했다면, Employee 클래스는 사람에 대한 정의와 함께 일하는 시간과 월급에 대한 변수를 추가한다. 즉 `__init__()` 함수를 재정의한다. 이때 부모 클래스의 `__init__()` 함수를 그대로 사용하려면 별도의 `__init__()` 함수를 만들지 않아도 된다. 하지만 기존 함수를 사용하면서 새로운 내용을 추가하기 위해서는 자식 클래스에 `__init__()` 함수를 생성하면서 **`super().__init__(매개변수)`**를 사용해야 한다. 여기서 `super()`는 부모 클래스를 가리킨다. 즉, 부모 클래스의 `__init__()` 함수를 그대로 사용한다는 뜻이다.
- ➔ 그 아래에는 필요한 자식 클래스의 새로운 변수를 추가하면 된다. 이러한 함수의 재정의를 **오버라이딩(overriding)**이라고 한다. 오버라이딩은 상속 시 함수 이름과 필요한 매개변수는 그대로 유지하면서 함수의 수행 코드를 변경하는 것이다. 같은 방식으로 `about_me()` 함수가 오버라이딩된 것을 확인할 수 있다. Person과 Employee에 대한 설명을 추가한 것이다.

04. 객체 지향 프로그래밍의 특징

■ 다형성 (polymorphism)

- 다형성(polymorphism)은 같은 이름의 메서드가 다른 기능을 할 수 있도록 하는 것을 말한다.

코드 10-6 polymorphism1.py

```
1 n_crawler = NaverCrawler()
2 d_crawler = DaumCrawler()
3 crawlers = [n_crawler, d_crawler]
4 news = []
5 for crawler in crawlers:
6     news.append(crawler.do_crawling())
```

- ➡ [코드 10-6]을 보면 두 Crawler 클래스가 같은 do_crawling() 함수를 가지고 이에 대한 역할이 같으므로 news 변수에 결과를 저장하는 데 문제가 없다. [코드 10-6]은 의사 코드(pseudo code)로, 실제 실행되는 코드는 아니지만 작동의 예시를 보기에는 적당하다. 이렇게 클래스의 다형성을 사용하여 다양한 프로그램을 작성할 수 있다.

04. 객체 지향 프로그래밍의 특징

■ 다형성

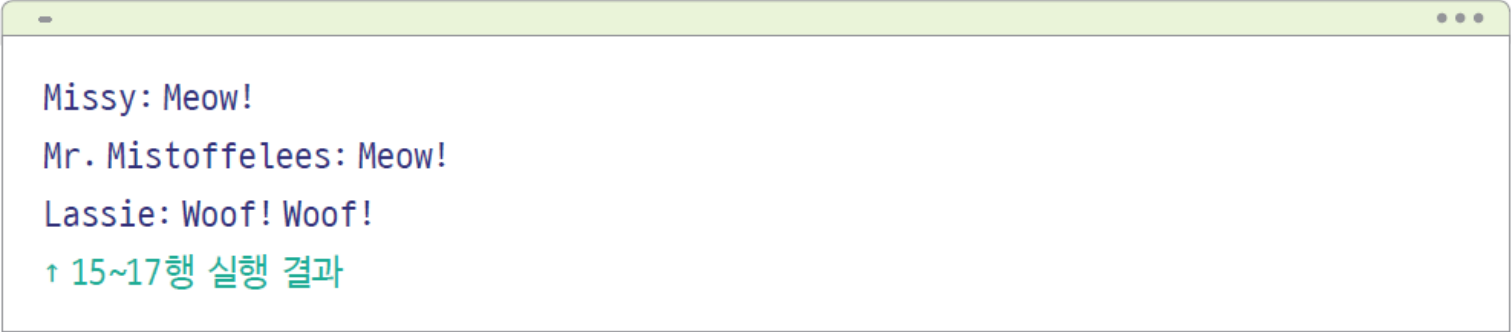
코드 10-7 polymorphism2.py

```
1 class Animal:
2     def __init__(self, name):
3         self.name = name
4     def talk(self):
5         raise NotImplementedError("Subclass must implement abstract method")
6
7 class Cat(Animal):
8     def talk(self):
9         return 'Meow!'
10
11 class Dog(Animal):
12     def talk(self):
13         return 'Woof! Woof!'
14
15 animals = [Cat('Missy'), Cat('Mr. Mistoffelees'), Dog('Lassie')]
```

04. 객체 지향 프로그래밍의 특징

■ 다형성

```
16 for animal in animals:  
17     print(animal.name + ': ' + animal.talk())
```



```
Missy: Meow!  
Mr. Mistoffelees: Meow!  
Lassie: Woof! Woof!  
↑ 15~17행 실행 결과
```

- ➡ [코드 10-7]에서 부모 클래스는 Animal이며, Cat과 Dog는 Animal 클래스를 상속받는다. 핵심 함수는 talk로, 각각 두 동물 클래스의 역할이 다른 것을 확인할 수 있다. Animal 클래스는 NotImplementedError라는 클래스를 호출한다. 이 클래스는 자식 클래스에만 해당 함수를 사용할 수 있도록 한다. 따라서 두 클래스가 내부 로직에서 같은 이름의 함수를 사용하여 결과를 출력하도록 한다. 실제로는 15~17행과 같이 사용할 수 있다.

04. 객체 지향 프로그래밍의 특징

■ 가시성 (Visibility / Encapsulation)

- 가시성visibility 은 객체의 정보를 볼 수 있는 레벨을 조절하여 객체의 정보 접근을 숨기는 것을 말하며, 다양한 이름으로 불린다. 파이썬에서는 가시성이라고 하지만, 좀 더 중요한 핵심 개념은 **캡슐화(encapsulation)** 와 **정보 은닉(information hiding)**이다.
- 파이썬의 가시성 사용 방법에 대한 예시 코드를 작성해야 하는 상황은 다음과 같다.

- Product 객체를 Inventory 객체에 추가
- Inventory에는 오직 Product 객체만 들어감
- Inventory에 Product가 몇 개인지 확인이 필요
- Inventory에 Product items는 직접 접근이 불가

04. 객체 지향 프로그래밍의 특징

■ 가시성

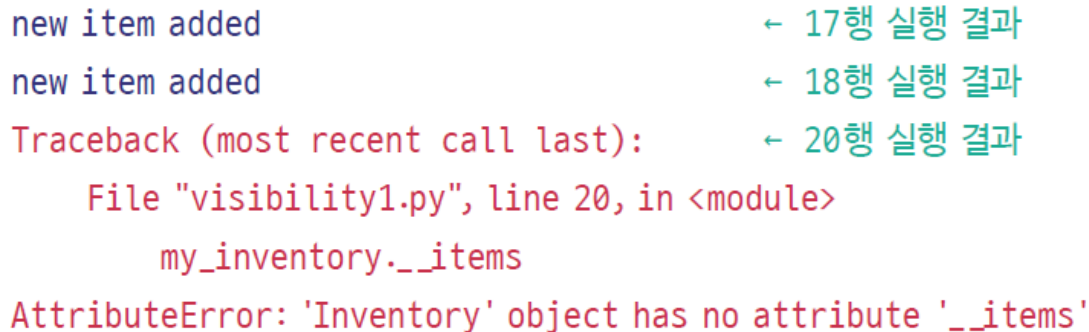
코드 10-8 visibility1.py

```
1 class Product(object):
2     pass
3
4 class Inventory(object):
5     def __init__(self):
6         self.__items = []
7     def add_new_item(self, product):
8         if type(product) == Product:
9             self.__items.append(product)
10            print("new item added")
11        else:
12            raise ValueError("Invalid Item")
13    def get_number_of_items(self):
14        return len(self.__items)
15
```

04. 객체 지향 프로그래밍의 특징

■ 가시성

```
16 my_inventory = Inventory()  
17 my_inventory.add_new_item(Product())  
18 my_inventory.add_new_item(Product())  
19  
20 my_inventory.__items
```



A terminal window with a light green title bar and three dots in the top right corner. It displays the output of a Python program. The first two lines are 'new item added' in blue. The third line is 'Traceback (most recent call last):' in red. The fourth line is 'File "visibility1.py", line 20, in <module>' in red. The fifth line is 'my_inventory.__items' in red. The sixth line is 'AttributeError: 'Inventory' object has no attribute '__items'' in red. To the right of the first three lines are green arrows pointing to the right, followed by Korean text: '← 17행 실행 결과', '← 18행 실행 결과', and '← 20행 실행 결과'.

```
new item added ← 17행 실행 결과  
new item added ← 18행 실행 결과  
Traceback (most recent call last): ← 20행 실행 결과  
    File "visibility1.py", line 20, in <module>  
        my_inventory.__items  
AttributeError: 'Inventory' object has no attribute '__items'
```

04. 객체 지향 프로그래밍의 특징

■ 가시성

- ➡ [코드 10-8]에서는 Inventory 객체에 `add_new_item()` 함수를 사용하여 새롭게 생성된 Product 객체를 넣어 준다. `__items`는 Product 객체가 들어가는 공간으로, `get_number_of_items()`를 사용하여 총 객체의 개수를 반환한다.
- ➡ 여기서 핵심은 `__items` 변수이다. Inventory를 저장하는 공간으로, `add_new_item()`을 통해 Product 객체를 넣을 수 있다. 하지만 다른 프로그램이 `add_new_item()`이 아니라 직접 해당 객체에 접근해 새로운 값을 추가하려고 한다면 어떻게 할까? 다른 코드에서는 잘 실행되다가 20행의 `my_inventory.__items`에서 오류가 발생한다. 왜냐하면 `__`가 특수 역할을 하는 예약 문자로 클래스에서 변수로 두 개 붙어, 사용될 클래스 내부에서만 접근할 수 있고, 외부에는 호출하여 사용하지 못하기 때문이다. 즉, **클래스 내부용으로만 변수를 사용하고 싶다면 '`__변수명`' 형태로 변수를 선언한다.** 가시성을 클래스 내로 한정하면서 값이 다르게 들어가는 것을 막을 수 있다. 이를 정보 은닉이라고 한다.
- ➡ 이러한 정보를 클래스 외부에서 사용하기 위해서는 어떻게 해야 할까? **데코레이터 (decorator)**라고 불리는 `@property`를 사용한다.

04. 객체 지향 프로그래밍의 특징

■ 가시성

- [코드 10-8]의 14행 뒷부분에 [코드 10-9]를 추가하여 @property를 사용하면 해당 변수를 외부에서 사용할 수 있다

코드 10-9 visibility2.py

```
1 class Inventory(object):
2     def __init__(self):
3         self._items = []           # private 변수로 선언(타인이 접근 못 함)
4
5     @property                       # property 데코레이터(숨겨진 변수 반환)
6     def items(self):
7         return self._items
```

- ➡ 다른 코드는 그대로 유지하고 마지막에 items라는 이름으로 메서드를 만들면서 @property를 메서드 상단에 입력한다. 그리고 외부에서 사용할 변수인 _items를 반환한다.

04. 객체 지향 프로그래밍의 특징

■ 가시성

- 코드를 추가하면 다음과 같이 외부에서도 해당 메서드를 사용할 수 있다.

```
>>> my_inventory = Inventory()  
>>> items = my_inventory.items  
>>> items.append(Product())
```

- ➡ 이번 코드에서는 오류가 발생하지 않았다. 여기서 주목할 부분은 `_items` 변수의 원래 이름이 아닌 `items`로 호출할 수 있다는 것이다. 바로 **@property를 붙인 함수 이름으로 실제 `_items`를 사용할 수 있는 것이다.** 이는 기존 private 변수를 누구나 사용할 수 있는 public 변수로 바꾸는 방법 중 하나이다.

모듈과 패키지

목차

1. 모듈과 패키지의 이해
2. 모듈 만들기
3. 패키지 만들기
4. 가상환경 사용하기

01

모듈과 패키지의 이해

01. 모듈과 패키지의 이해



(a) 파이썬



(b) 자바

[파이썬과 자바의 작성 비유]

01. 모듈과 패키지의 이해

■ 모듈의 개념

- 프로그래밍에서의 모듈은 **작은 프로그램 조각**을 뜻한다. 즉, 하나하나 연결해 어떤 목적을 가진 프로그램을 만드는 작은 프로그램이다. 각 모듈 역시 저마다 역할이 있고, 서로 다른 모듈과 인터페이스만 연결되면 사용할 수 있다.

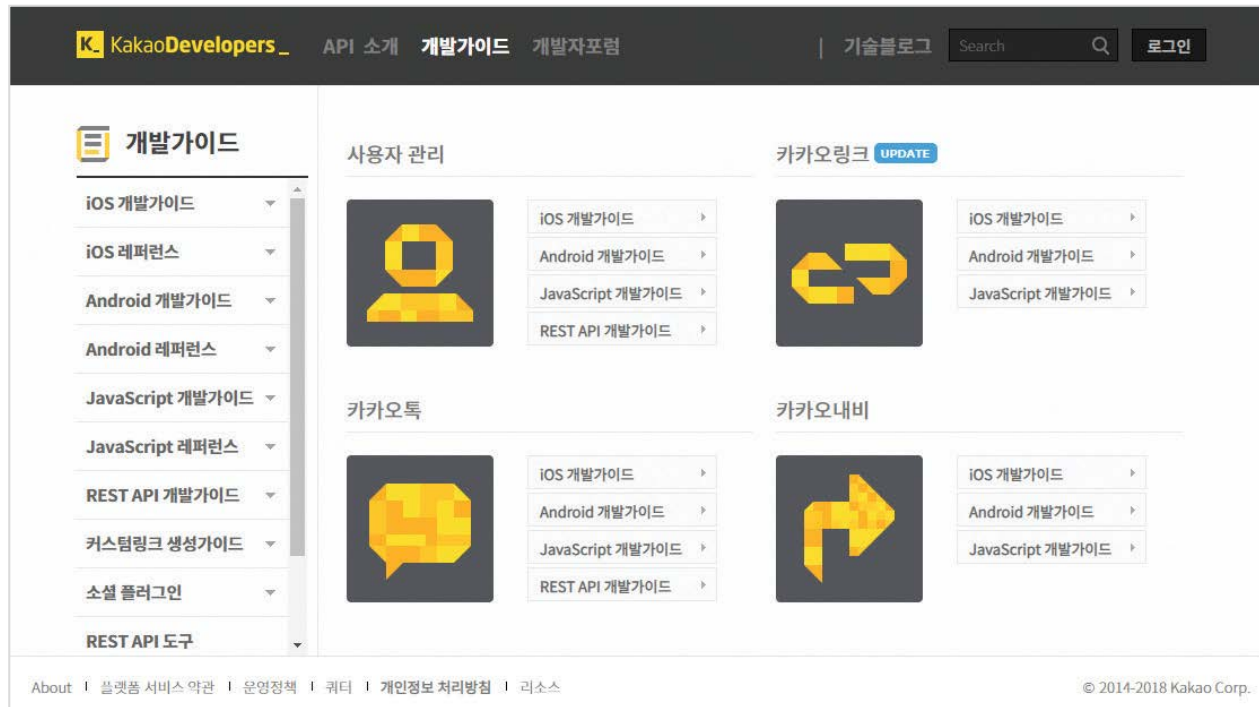


[구글의 프로젝트 Ara 콘셉트]

01. 모듈과 패키지의 이해

■ 모듈의 개념

- 모듈화된 프로그램을 사용하면, 다른 개발자가 만든 프로그램이나 자신이 만든 프로그램을 매우 쉽게 사용하거나 제공할 수 있다.



01. 모듈과 패키지의 이해

■ 모듈의 개념

- **내장 모듈**이라고 하여 파이썬에서 기본적으로 제공하는 모듈 중 대표적으로 random 모듈이 있다. 이는 난수를 쉽게 생성해 주는 모듈이다. random 모듈을 호출하기 위한 코드는 다음과 같다.

```
>>> import random
>>> random.randint(1, 1000)
198
```

- ➔ import 구문이 중요하다. import 구문은 뒤에 있는 모듈, 즉 random을 사용할 수 있도록 호출하라는 명령어이다. 다음으로 해당 모듈의 이름을 사용하여 그 모듈 안에 있는 함수, 여기서는 randint() 함수를 사용할 수 있다. randint() 함수를 사용하기 위해서는 이 randint 함수의 인터페이스, 즉 매개변수의 설정이 어떻게 되어 있는지 알아야 한다.

Random (난수)

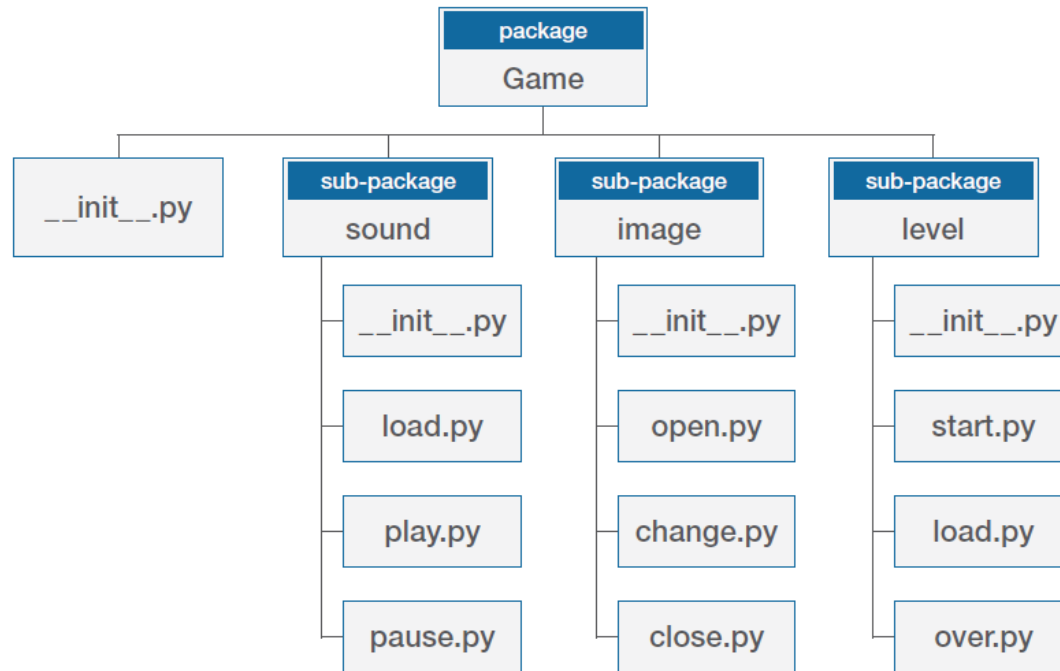
- randint(최소, 최대) : 입력 파라미터인 최소부터 최대까지 중 임의의 정수를 리턴한다
- random() : 0 부터 1 사이의 부동소수점(float) 숫자를 리턴한다
- uniform(최소, 최대) : 입력 파라미터인 최소부터 최대까지 중 임의의 부동소수점(float) 숫자를 리턴한다
- randrange(시작, 끝[, 간격]) : 입력 파라미터인 시작부터 끝값까지 (지정된 간격으로 나열된) 숫자 중 임의의 정수를 리턴한다

```
1  from random import *
2
3  i = randint(1, 100) # 1부터 100 사이의 임의의 정수
4  print(i)
5
6  f = random() # 0부터 1 사이의 임의의 float
7  print(f)
8
9  f = uniform(1.0, 36.5) # 1부터 36.5 사이의 임의의 float
10 print(f)
11
12 i = randrange(1, 101, 2) # 1부터 100 사이의 임의의 짝수
13 print(i)
14
15 i = randrange(10) # 0부터 9 사이의 임의의 정수
16 print(i)
```

01. 모듈과 패키지의 이해

■ 패키지의 개념

- 패키지(packages)는 모듈의 묶음이다. 일종의 디렉터리처럼 하나의 패키지 안에 여러 개의 모듈이 있는데, 이 모듈들이 서로 포함 관계를 가지며 거대한 패키지를 만든다.



[모듈과 패키지의 관계]

02

모듈 만들기

02. 모듈 만들기

■ 모듈 만들기 실습

- 간단한 모듈을 만들어 보자. [코드 11-1]과 같은 코드를 작성하여 'fah_converter.py'로 저장한다.

코드 11-1 fah_converter.py

```
1 def covert_c_to_f(celcius_value):  
2     return celcius_value * 9.0 / 5 + 32
```

02. 모듈 만들기

■ 모듈 만들기 실습

- 다음으로 해당 모듈을 사용하는 코드(흔히 클라이언트 코드라고 한다.)를 [코드 11-2]와 같이 작성하여 'module_ex.py'에 저장한다.

코드 11-2 module_ex.py

```
1 import fah_converter
2
3 print ("Enter a celsius value:")
4 celsius = float(input())
5 fahrenheit = fah_converter.covert_c_to_f(celsius)
6 print ("That's", fahrenheit, "degrees Fahrenheit.")
```

Enter a celsius value:

10

← 사용자 입력

That's 50.0 degrees Fahrenheit.

← 결과값 출력

02. 모듈 만들기

■ 모듈 만들기 실습

- ➡ [코드 11-2]에서 가장 중요한 핵심 코드는 1행의 `import fah_converter`로, 기존에 만든 코드 파일에서 `.py`를 빼고 해당 파일의 이름만으로 파일의 함수를 불러 사용할 수 있다. 즉, `.py` 자체가 하나의 모듈이 되어 해당 모듈의 코드를 가져다 사용할 수 있다. 이 코드에서는 `fah_converter`가 모듈이고, 해당 모듈 안의 함수 `covert_c_to_f()`를 가져다 사용하기 위해 5행에서 `fahrenheit = fah_converter.covert_c_to_f(celsius)` 코드를 작성하였다.
- ➡ 여기서 핵심은 호출받는 모듈과 호출하여 사용하는 클라이언트 프로그램이 같은 디렉터리 안에 있어야 한다는 것이다. 여기서는 `fah_converter.py`와 `module_ex.py`가 같은 디렉터리안에 있어야 문제없이 실행된다.

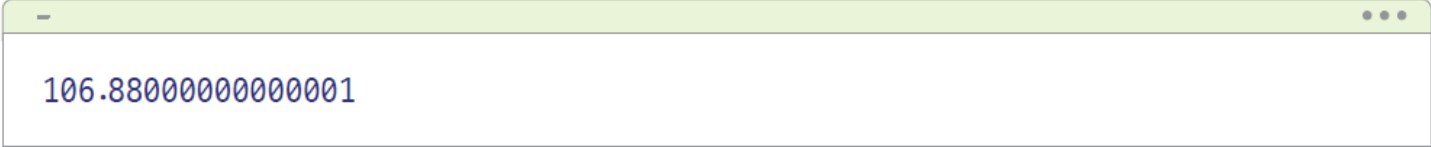
02. 모듈 만들기

■ 네임스페이스

- 네임스페이스는 **모듈 호출의 범위를 지정**한다.
- 네임스페이스를 만드는 방법에 대해 알아보자. 첫 번째는 모듈 이름에 **알리아스(alias)**를 생성하여 모듈 안으로 코드를 호출하는 방법이다. 알리아스는 모듈의 이름을 바꿔 부를 때 사용한다.

코드 11-3 namespace1.py

```
1 import fah_converter as fah
2 print(fah.covert_c_to_f(41.6))
```



106.88000000000001

- ➡ fah_converter 모듈을 fah로 이름을 변경하여 호출하였다. 그리고 fah.covert_c_to_f(41.6) 코드로 fah_converter 모듈 안에 covert_c_to_f() 함수를 호출하였다. '모듈명.함수명(또는 클래스명/변수명)'은 해당 모듈 안에 있는 함수, 클래스, 변수를 호출할 수 있다.


02. 모듈 만들기

■ 네임스페이스

- 두 번째 방법은 from 구문을 사용하여 모듈에서 특정 함수 또는 클래스만 호출하는 방법이다.

코드 11-4 namespace2.py

```
1 from fah_converter import covert_c_to_f
2 print(covert_c_to_f(41.6))
```



106.88000000000001

- ➔ 1행에서처럼 'from 모듈명 import 모듈 안에 있는 함수명'을 작성하여 해당 모듈 안에 있는 함수를 가져다 사용할 수 있다. 주의할 점은 from은 패키지를 호출하고, 해당 패키지 안에 있는 모듈을 호출할 때도 from 키워드를 사용할 수 있으니 참고하기 바란다. 패키지와 패키지, 패키지와 모듈 간에는 서로 중첩 구조를 가질 수 있고, 이 중첩 구조를 호출하는 것이 바로 from의 역할이다.


02. 모듈 만들기

■ 네임스페이스

- 세 번째 방법은 해당 모듈 안에 있는 모든 함수, 클래스, 변수를 가져오는 별표(*)를 사용하는 것이다. 일반적으로 컴퓨터에서 별표는 곱셈의 의미도 있지만, 모든 것이라는 뜻도 있다. [코드 11-5]의 1행과 같이 'from 모듈명 import *'라고 입력하면, 해당 모듈 안에 있는 모든 사용가능한 리소스를 호출한다.

코드 11-5 namespace3.py

```
1 from fah_converter import *  
2 print(covert_c_to_f(41.6))
```



106.88000000000001

02. 모듈 만들기

■ 내장 모듈의 사용 : random 모듈

- 난수 생성 모듈은 이미 많이 본 random 모듈을 사용하면 된다. 해당 모듈 안에는 여러 가지 함수가 있는 정수 모듈을 생성하는 randint() 함수와 임의의 난수를 생성하는 random() 함수를 쓸 수 있다.

```
>>> import random
>>> print (random.randint (0, 100))           # 0~100 사이의 정수 난수를 생성
7
>>> print (random.random())                   # 일반적인 난수 생성
0.056550421789531846
```

02. 모듈 만들기

■ 내장 모듈의 사용 : **time** 모듈

- 시간과 관련된 time 모듈은 일반적으로 시간을 변경하거나 현재 시각을 출력한다. 대표적으로 프로그램이 동작하는 현재 시각을 출력할 수 있다.

```
>>> import time
>>> print(time.localtime())           # 현재 시각 출력
time.struct_time(tm_year=2018, tm_mon=8, tm_mday=19, tm_hour=22, tm_min=9,
tm_sec=21, tm_wday=6, tm_yday=231, tm_isdst=0)
```

02. 모듈 만들기

■ 내장 모듈의 사용 : urllib 모듈

- 웹과 관련된 urllib 모듈은 웹 주소의 정보를 불러온다. 대표적으로 urllib의 request 모듈을 사용하면 특정 URL의 정보를 불러올 수 있다. urllib.request.urlopen()의 괄호에 특정 웹주소를 입력하면 해당 주소의 HTML 정보를 가져온다.

```
>>> import urllib.request
>>> response = urllib.request.urlopen("http://theteamlab.io")
>>> print(response.read())
```

02. 모듈 만들기

여기서 잠깐! 파이썬 모듈 검색

- 이외에도 많은 파이썬 모듈이 있다. 그렇다면 이 모듈들은 어떻게 불러 사용할 수 있을까? 가장 좋은 방법은 구글에서 검색하는 것이다. 특히 영어로 검색하는 것이 좋다. 예를 들어, 프로그래밍이 걸린 시간을 쓰는 모듈을 찾고 싶다면 다음과 같은 방식으로 검색어를 입력하여 찾으면 된다.

```
python time module run time
```

- 다른 방법으로는 우리나라의 대표 사이트인 파이썬 코리아에 문의할 수 있다. 파이썬 코리아의 페이스북 페이지는 파이썬 개발자에게 많은 정보를 제공하는 대표적인 커뮤니티이다. 다음 주소에서 여러 질문을 하면 많은 개발자가 친절히 알려줄 것이다.

```
https://www.facebook.com/groups/pythonkorea
```

03

패키지 만들기

03. 패키지 만들기

■ 패키지의 구성

- 패키지는 하나의 대형 프로젝트를 수행하기 위한 모듈의 묶음이다. 모듈은 하나의 파일로 이루어져 있고, 패키지는 파일이 포함된 디렉터리(폴더)로 구성된다. 즉, 여러 개의 .py 파일이 하나의 디렉터리에 들어가 있는 것을 패키지라고 한다.
- 흔히 다른 사람이 만든 프로그램을 불러 사용하는 것을 라이브러리(library)라고 하는데, 파이썬에서는 패키지를 하나의 라이브러리로 이해하면 된다.
- 파이썬의 모듈을 구성할 때와 마찬가지로 패키지에도 예약어가 있다. 한 가지 주의할 점은 패키지에서는 파일명 자체가 예약어를 반드시 지켜야만 실행되는 경우가 있다. 따라서 패키지 내의 몇몇 파일에는 `__init__`, `__main__` 등의 키워드 파일명이 사용된다.

03. 패키지 만들기

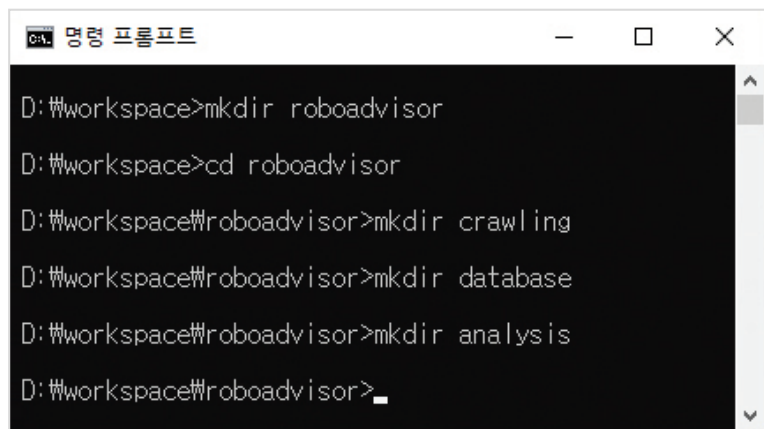
■ 패키지 만들기 실습 : 1단계: 디렉터리 구성하기

- 이번 실습에서 만들 패키지 이름은 'roboadvisor'이다. roboadvisor에는 세 가지 기능이 있다고 가정하자.
 - ① **crawling(크롤링)**: 주식 관련 데이터를 인터넷에서 가져오는 기능
 - ② **database(데이터베이스)**: 가져온 데이터를 데이터베이스에 저장하는 기능
 - ③ **analysis(분석)**: 해당 정보를 분석하여 의미 있는 값을 뽑는 기능

03. 패키지 만들기

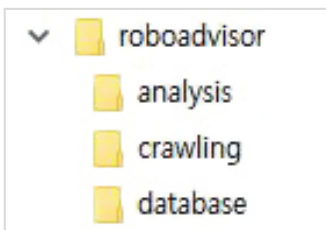
■ 패키지 만들기 실습 : 1단계: 디렉터리 구성하기

- 패키지를 구성하기 위한 첫 번째 단계는 각 패키지의 세부 패키지에 맞춰 디렉터리를 구성하는 것이다. 먼저 cmd 창에 다음 명령을 입력하여 디렉터를 생성한다



```
D:\workspace>mkdir roboadvisor
D:\workspace>cd roboadvisor
D:\workspace\roboadvisor>mkdir crawling
D:\workspace\roboadvisor>mkdir database
D:\workspace\roboadvisor>mkdir analysis
D:\workspace\roboadvisor>.
```

[세부 패키지에 맞춰 디렉터리 생성]

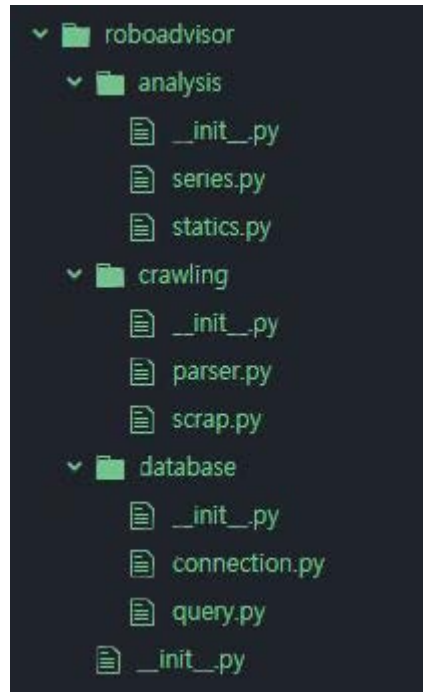


[roboadvisor 디렉터리 구성]

03. 패키지 만들기

■ 패키지 만들기 실습 : 2단계: 디렉터리별로 필요한 모듈 만들기

- 만들어진 디렉터리에 필요한 모듈을 만든다. 하나의 패키지는 중첩된 구조로 만들 수 있으므로 패키지 안에 또 하나의 패키지가 들어갈 수 있다. 하지만 이렇게 각각의 디렉터리를 하나의 패키지로 선언하기 위해서는 예약된 파일을 만들어야 한다. 바로 `__init__.py`이다.



[패키지의 구조]

03. 패키지 만들기

여기서 잠깐! 패키지의 구조 설계

- 앞의 패키지 구조는 임의로 작성한 것이다. 패키지의 구조를 만들기 위해 프로그램 개발자는 설계를 해야 한다. 하위 패키지별로 해야 하는 일과 하위 패키지에 소속된 모듈들이 해야 할 일을 따로 정의해 각 모듈에 역할을 부여하는 것이다. 여기까지 가는 과정에는 많은 경험과 지식이 필요하다. 지금 단계에서는 어려울 수 있으니, 이 예제에서는 대략적인 구조와 역할을 임의로 작성한다.

03. 패키지 만들기

■ 패키지 만들기 실습 : 2단계: 디렉터리별로 필요한 모듈 만들기

- 이제 각 하위 패키지에 포함된 모듈에 필요한 기능을 구현하기 위해 코딩을 하자.
- [코드11-6]과 [코드 11-7]과 같은 방식으로 crawling 디렉터리 아래 parser.py와 scrap.py에, database 디렉터리 아래 connection.py와 query.py에 코드를 입력한다.

코드 11-6 series.py(analysis 디렉터리)

```
1 def series_test():  
2     print("series")
```

코드 11-7 statics.py(analysis 디렉터리)

```
1 def statics_test():  
2     print("statics")
```

03. 패키지 만들기

■ 패키지 만들기 실습 : 2단계: 디렉터리별로 필요한 모듈 만들기

- 실제 해당 모듈을 사용하기 위해 다음과 같이 파이썬 셸에서 코드를 작성한다. 이 코드는 roboadvisor의 상위 디렉터리에서 파이썬 셸을 실행해야 정상적으로 진행된다.

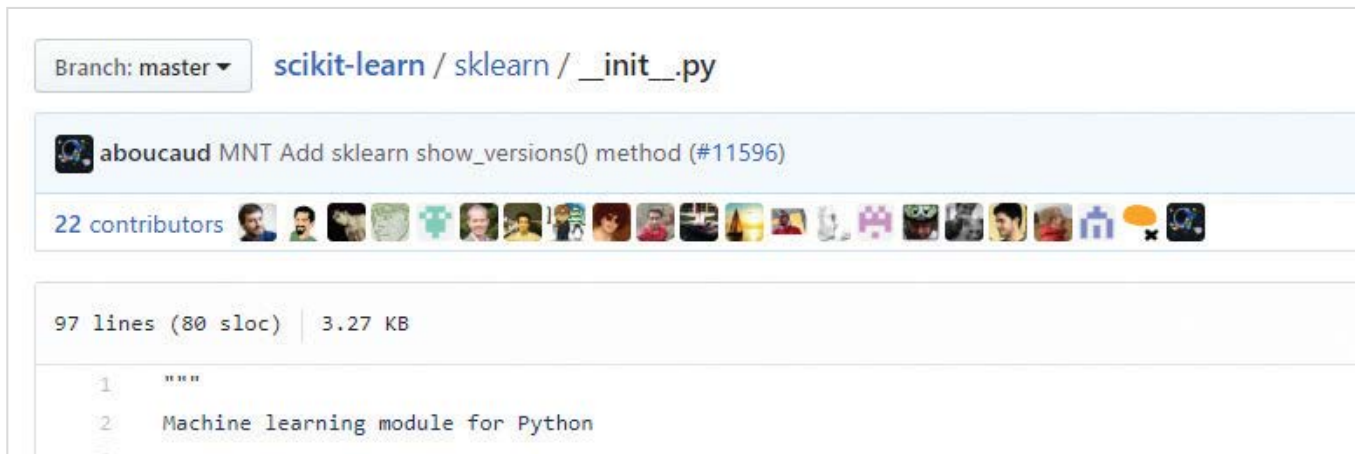
```
>>> from roboadvisor.analysis import series
>>> series.series_test()
series
```

- ➡ 이 코드를 실행하면 roboadvisor 디렉터리 안에는 ' _ _pycache_ _'라는 디렉터리가 생성되는데, 이는 파이썬의 언어적 특성으로 생기는 결과이다. _ _pycache_ _ 디렉터리에는 해당 프로그램이 작동될 때 사용하기 위한 모듈들을 컴파일하고, 그 결과를 저장한다. 이렇게 한 번 _ _pycache_ _ 디렉터리가 생성되면 그 시점에서 해당 모듈을 수정해도 결과가 반영되지 않는다. 해당 프로그램 또는 파이썬 셸이 완전히 종료한 후 수정해야 해당 모듈의 결과를 반영할 수 있다. 인터프리터 언어이지만 내부적으로 컴파일도 하고, 효율적으로 사용하기 위한 여러 가지 작업이 있다는 것을 기억하기 바란다.

03. 패키지 만들기

■ 패키지 만들기 실습 : 3단계: 디렉터리별로 `__init__.py` 구성하기

- 디렉터리별로 `__init__.py` 파일을 구성한다. `__init__.py`은 해당 디렉터리가 파이썬의 패키지라고 선언하는 초기화 스크립트이다. `__init__.py` 파일은 파이썬의 거의 모든 라이브러리에 있다. 예를 들어, 대표적인 파이썬 머신러닝 라이브러리인 scikit-learn의 경우 다음과 같이 가장 상위 디렉터리부터 `__init__.py` 파일이 있는 것을 확인할 수 있다.



[scikit-learn에서 확인할 수 있는 `__init__.py` 파일]

03. 패키지 만들기

■ 패키지 만들기 실습 : 3단계: 디렉터리별로 `__init__.py` 구성하기

- `__init__.py` 파일은 패키지 개발자, 설치 시 확인해야 할 내용 등 메타데이터라고 할 수 있는 내용을 담고 있다. 하지만 가장 중요한 내용은 이 패키지의 구조이다. 일반적으로 `__init__.py` 파일에는 다음과 같이 해당 패키지가 포함된 모듈에 관한 정보가 있다. 다음 코드를 `roboadvisor` 디렉터리의 `__init__.py`에 입력한다

코드 11-8 `__init__.py`(`roboadvisor` 디렉터리)

```
1 import analysis
2 import crawling
3 import database
4
5 __all__ = ['analysis', 'crawling', 'database']
```

- ➔ `roboadvisor` 디렉터리에는 3개의 하위 패키지, 즉 `analysis`, `crawling`, `database`가 있다. 이 각각의 패키지를 `__init__.py` 안에 `__all__`과 `import`문을 사용해 선언해야 한다. 따라서 `__all__`이라는 리스트형의 변수를 만들어 차례대로 하위 패키지의 이름을 작성하고, 같은 방법으로 각 하위 패키지를 `import`문으로 호출한다.

03. 패키지 만들기

■ 패키지 만들기 실습 : 3단계: 디렉터리별로 `__init__.py` 구성하기

- 하위 패키지의 `__init__.py` 파일도 마찬가지이다. 예를 들어, `analysis` 디렉터리의 `__init__.py` 파일은 다음과 같이 각 패키지에 포함된 모듈명을 모두 작성해야 한다. 패키지로 표시하기 위해 꼭 해야 하는 작업이며 패키지별로 모두 처리해야 한다. `crawling`과 `database` 디렉터리의 `__init__.py` 파일에도 다음과 같은 방식으로 코드를 입력하고 저장한다.

코드 11-9 `__init__.py`(`analysis` 디렉터리)

```
1 from . import series
2 from . import statics
3
4 __all__ = ['series', 'statics']
```


03. 패키지 만들기

여기서  잠깐! `__init__.py` 파일을 만들지 않으면 어떤 문제가 발생할까?

- 파이썬 3.6 버전 이상에서는 `__init__.py` 파일을 만들지 않아도 큰 문제가 생기지 않는다. 하지만 파이썬 3.3 버전 이하에서는 `__init__.py`가 없을 경우 해당 디렉토리를 패키지로 인정하지 않는다. 물론 많은 사람이 상위 버전의 파이썬을 사용하기 때문에 문제 되지 않을 수도 있지만, 하위 버전의 파이썬을 사용할 수도 있으니 `__init__.py` 파일은 패키지를 만들 때 반드시 생성하도록 한다.

03. 패키지 만들기

■ 패키지 만들기 실습 : 4단계: `__main__.py` 파일 만들기

- 4단계에서는 패키지를 한 번에 사용하기 위해 `roboadvisor` 디렉터리에 `__main__.py` 파일을 만든다. `__main__.py` 파일을 만드는 이유는 패키지 자체를 실행하기 위해서이다. 지금까지 계속 파이썬 파일명 형태인 `.py` 파일로 실행하였다. 같은 방법으로 패키지 자체를 실행하기 위해 만들어야 하는 것이 `__main__.py` 파일이다. `__main__.py` 파일의 구성은 간단하다. 기본적으로 호출해야 하는 여러 모듈을 `from`과 `import`문으로 호출한 후, `if __name__ == '__main__':` 구문 아래에 실제 실행 코드를 작성하면 된다.

코드 11-10 `__main__.py`

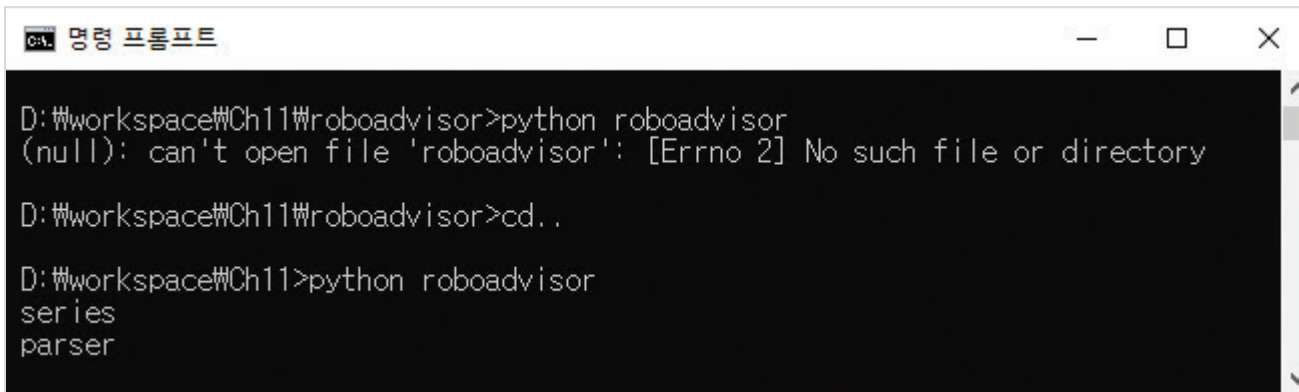
```
1 from analysis.series import series_test
2 from crawling.parser import parser_test
3
4 if __name__ == '__main__':
5     series_test()
6     parser_test()
```

03. 패키지 만들기

■ 패키지 만들기 실습 : 5단계: 실행하기(패키지 이름만으로 호출하기)

- 마지막 5단계에서는 해당 패키지를 실행한다. 모든 코드를 작성한 후, 해당 패키지의 최상위 디렉터리(본 예시에서는 roboadvisor의 상위 디렉터리)에서 'python 패키지명'을 입력하여 실행한다.

```
python roboadvisor  
series  
parser
```



```
명령 프롬프트  
D:\workspace\Ch11\roboadvisor>python roboadvisor  
(null): can't open file 'roboadvisor': [Errno 2] No such file or directory  
  
D:\workspace\Ch11\roboadvisor>cd..  
  
D:\workspace\Ch11>python roboadvisor  
series  
parser
```

[패키지 실행]

03. 패키지 만들기

■ 패키지 네임스페이스 : 절대 참조

- 먼저 절대 참조의 예시부터 보자.

```
from roboadvisor.analysis import series
```

- ➡ 위 코드에서 from은 roboadvisor부터 시작한다. 즉, 패키지 이름부터 시작하여 series까지 모든 경로를 입력한다. 'from 전체 패키지.서브 패키지 import 모듈' 형식이다. 이렇게 전체 경로를 모두 입력하는 것을 절대 참조라고 한다. `__init__.py` 파일을 만들 때도 절대 참조로 모듈을 호출하는 것이 좋다. 한 가지 주의할 점은 가장 상위에 있는 `__init__.py` 파일도 상위 디렉터리 roboadvisor를 넣는 것이 좋다.

코드 11-11 reference1.py

```
1 __all__ = ['analysis', 'crawling', 'database']
2
3 from roboadvisor import analysis
4 from roboadvisor import crawling
5 from roboadvisor import database
```

03. 패키지 만들기

■ 패키지 네임스페이스 : 상대 참조

- 상대 참조의 핵심은 현재의 디렉터리를 기준으로 모듈을 호출하는 것이다.

코드 11-12 reference2.py

```
1 from .series import series_test
2 from ..crawling.parser import parser_test
```

- ➡ 여기서 가장 중요한 코드는 `.series`와 `..crawling.parser`이다. 먼저 점 1개(`.`)는 현재 디렉터리를, 점 2개(`..`)는 부모 디렉터리를 뜻한다.

04

가상환경 사용하기

04. 가상환경 사용하기

■ 가상환경의 개념

- 일반적으로 어떤 프로젝트를 수행할 때는 파이썬 코드를 수행할 기본 인터프리터에서 추가로 프로젝트별로 필요한 패키지를 설치한다. 이러한 패키지를 설치할 때 서로 다른 프로젝트가 영향을 받지 않도록 독립적인 프로젝트 수행 환경을 구성하는데, 이를 가상환경이라고 한다.

가상환경 도구	특징
virtualenv + pip	<ul style="list-style-type: none">가장 대표적인 가상환경 관리 도구레퍼런스와 패키지가 가장 많음
conda	<ul style="list-style-type: none">상용 가상환경 도구인 miniconda의 기본 가상환경 도구설치가 쉬워 윈도우에서 유용함

[가상환경 도구]

04. 가상환경 사용하기

■ 가상환경 설정하기 : 가상환경 만들기

- 가상환경을 만드는 명령어는 다음과 같다.

```
conda create -n my_project python=3.4
```

conda create -n my_project python=3.4

↑
가상환경 새로 만들기

↑
가상환경 이름

↑
파이썬 버전

[가상환경 만들기 명령어]

04. 가상환경 사용하기

■ 가상환경 설정하기 : 가상환경 만들기

```
명령 프롬프트
D:\workspace>conda create -n my_project python=3.4
Solving environment: done

## Package Plan ##

environment location: C:\Miniconda3\envs\my_project

added / updated specs:
- python=3.4

The following packages will be downloaded:

package                        | build                | size
-----|-----|-----
setuptools-27.2.0              | py34_1               | 762 KB
vc-10                           | 0                    | 702 B
vs2010_runtime-10.00.40219.1 | 2                    | 1.1 MB
python-3.4.5                   | 0                    | 22.9 MB
wheel-0.29.0                   | py34_0               | 123 KB
pip-9.0.1                      | py34_1               | 1.7 MB
-----|-----|-----
Total:                          |                      | 26.6 MB

The following NEW packages will be INSTALLED:

pip: 9.0.1-py34_1
python: 3.4.5-0
setuptools: 27.2.0-py34_1
vc: 10-0
vs2010_runtime: 10.00.40219.1-2
wheel: 0.29.0-py34_0

Proceed ([y]/n)? y
```

[가상환경 설치 명령 입력]

04. 가상환경 사용하기

■ 가상환경 설정하기 : 가상환경 실행하기

```
#  
# To activate this environment, use:  
# > activate my_project  
#  
# To deactivate an active environment, use:  
# > deactivate  
#  
# * for power-users using bash, you must source
```

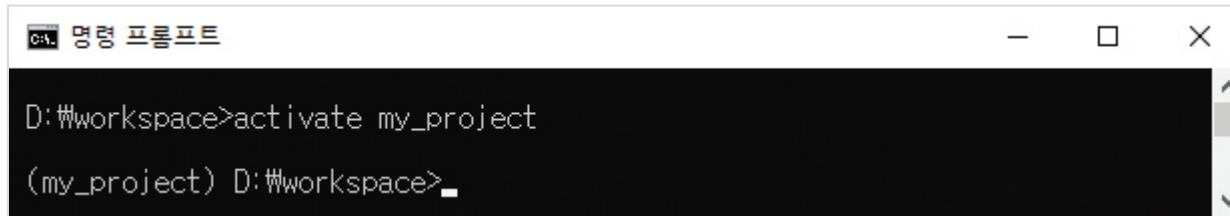
[가상환경 설치 화면]

```
activate my_project
```

- 이 코드는 my_project라는 가상환경을 활성화(activate)하라는 뜻이다. 구성된 가상환경의 이름을 activate 다음에 넣으면 해당 가상환경이 실행되고, 프롬프트 앞에(my_project)라는 가상환경 이름이 붙는다. 이제 이 환경에서는 가상환경의 인터프리터만 실행된다.

04. 가상환경 사용하기

■ 가상환경 설정하기 : 가상환경 실행하기



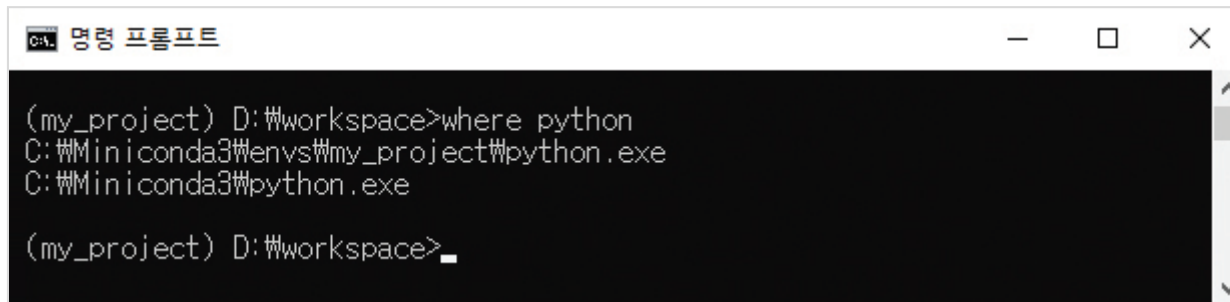
```
C:\> 명령 프롬프트

D:\workspace>activate my_project

(my_project) D:\workspace>_
```

[가상환경 활성화]

- 이 상태에서 'where python'을 입력하면 현재 실행되는 파이썬의 위치가 어디인지 출력된다.



```
C:\> 명령 프롬프트

(my_project) D:\workspace>where python
C:\Miniconda3\envs\my_project\python.exe
C:\Miniconda3\python.exe

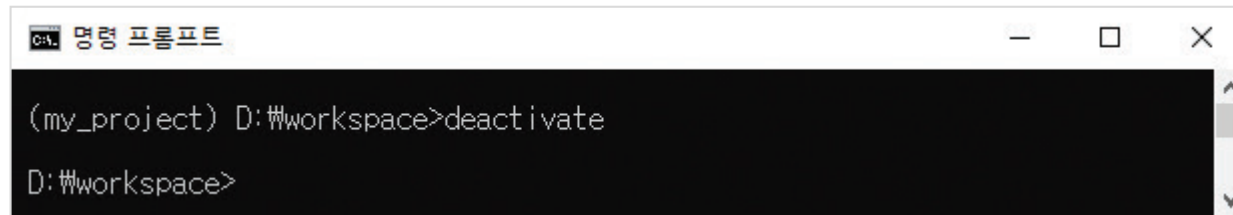
(my_project) D:\workspace>_
```

[파이썬의 위치 확인]

04. 가상환경 사용하기

■ 가상환경 설정하기 : 가상환경 실행하기

- 실행된 가상환경을 종료하기 위해서는 'deactivate'를 입력하면 된다.



```
Ca. 명령 프롬프트
(my_project) D:\workspace>deactivate
D:\workspace>
```

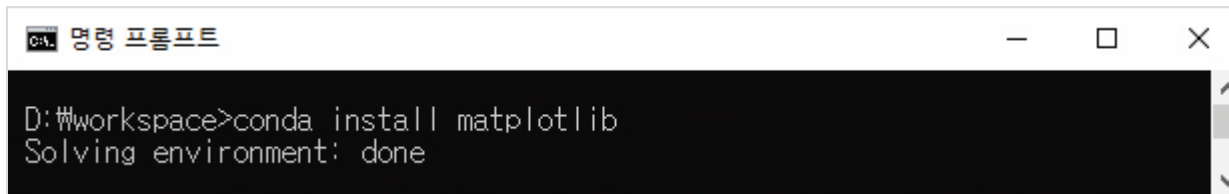
[가상환경 종료]

04. 가상환경 사용하기

■ 가상환경 설정하기 : 가상환경 패키지 설치하기

- 가상환경의 실행을 완료했으니 해당 가상환경에서 새로운 패키지를 설치해야 한다. 패키지를 설치하기 위해서는 다음과 같은 명령어를 입력한다.

```
conda install matplotlib
```



A screenshot of a Windows Command Prompt window. The title bar shows 'C:\> 명령 프롬프트' (C:\> Command Prompt). The window has standard Windows window controls (minimize, maximize, close). The command prompt shows the following text: 'D:\workspace>conda install matplotlib' followed by 'Solving environment: done' on the next line. There is a vertical scrollbar on the right side of the command prompt window.

[matplotlib 설치]

04. 가상환경 사용하기

■ 가상환경 설정하기 : 가상환경 패키지 실습하기

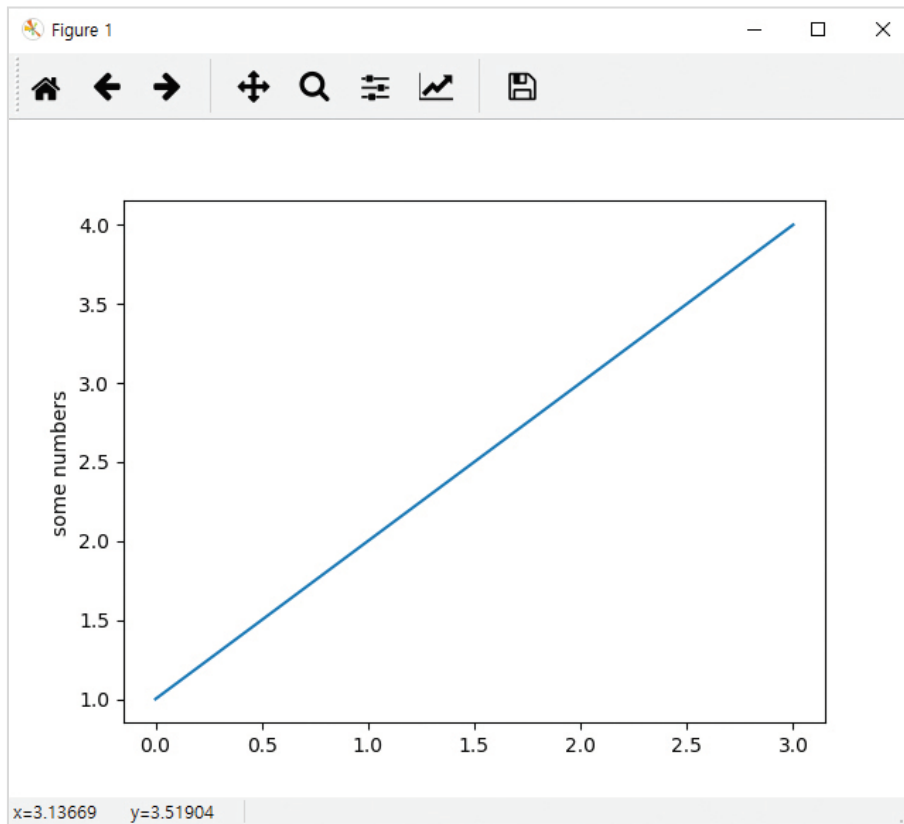
- 설치된 패키지를 실행해 보자. 앞에서 설치한 matplotlib은 대표적인 파이썬 그래프 관리 패키지로, 엑셀과 같은 그래프를 화면에 출력한다. 데이터 분석을 할 경우, 다양한 데이터 분석 도구와 함께 사용한다.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1, 2, 3, 4])
[<matplotlib.lines.Line2D object at 0x000001E8CC52C080>]
>>> plt.ylabel('some numbers')
Text(0, 0.5, 'some numbers')
>>> plt.show()
```

04. 가상환경 사용하기

■ 가상환경 설정하기 : 가상환경 패키지 실습하기

- 코드를 실행하면 다음과 같은 깔끔한 그래프 화면을 확인할 수 있다. matplotlib은 논문을 쓰거나 여러 가지 데이터 분석 결과를 보여 줄 때 매우 유용한 모듈이다.



[matplotlib 실행 결과 화면]

04. 가상환경 사용하기

여기서 잠깐! jupyter 패키지

- 다른 패키지로 데이터를 분석할 때 매우 유용한 패키지로 jupyter가 있다. 먼저 패키지를 설치하기 위해 cmd 창에서 다음과 같은 명령어를 입력한다.

```
conda install jupyter
```

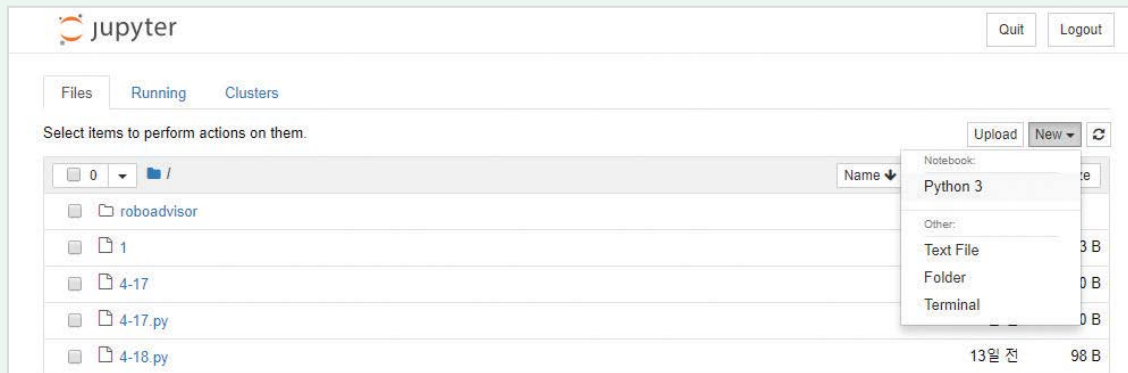
- 다음과 같이 설치한 후, 'jupyter notebook' 을 입력하여 실행하면 jupyter 환경에서 코딩할 수 있다.

```
jupyter notebook
```

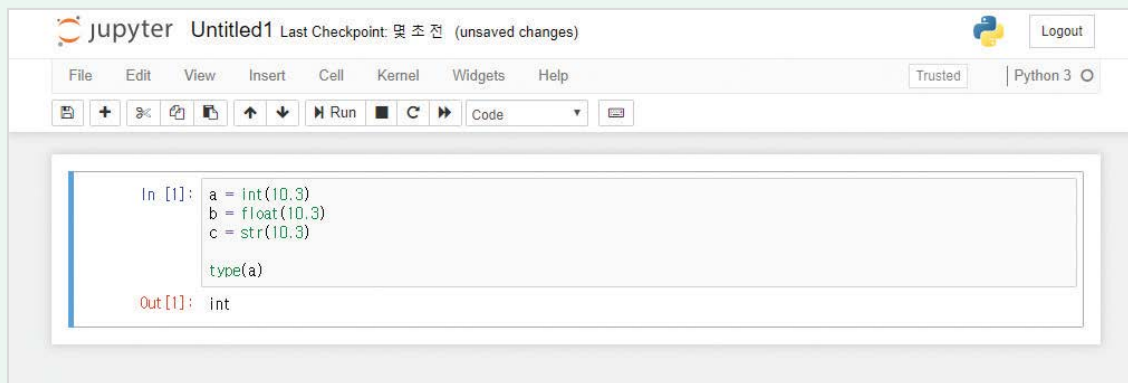
- jupyter를 실행하면 웹에서 코딩할 수 있는 환경이 나온다. 여기에서 [New] 버튼을 클릭하여 새로운 notebook을 생성한 후, 코딩하고 Ctrl + Enter 를 누르면 결과를 볼 수 있다.

04. 가상환경 사용하기

여기서  잠깐! jupyter 패키지



[jupyter 메인 화면]



[jupyter 웹 코딩]

예외 처리

목차

1. 예외 처리

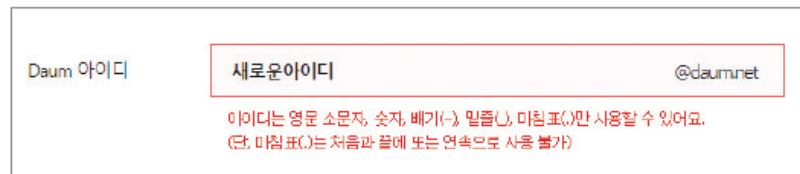
01

예외 처리

01. 예외 처리

■ 예외의 개념과 사례

- 예외(exception) 란 프로그램을 개발하면서 예상하지 못한 상황이 발생한 것이다. 프로그래밍의 예외는 크게 예측 가능한 예외와 예측 불가능한 예외로 나눌 수 있다.

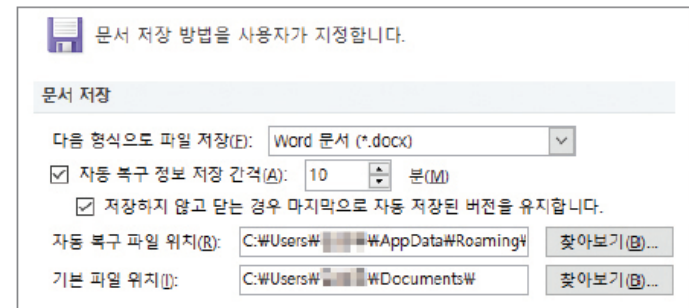


Daum 아이디

새로운아이디 @daumnet

아이디는 영문 소문자, 숫자, 배기(-), 밑줄(_)만 사용할 수 있어요.
(단, 마침표(.)는 처음과 끝에 또는 연속으로 사용 불가)

(a) 아이디 생성 오류 입력



문서 저장 방법을 사용자가 지정합니다.

문서 저장

다음 형식으로 파일 저장(E): Word 문서 (*.docx)

☒ 자동 복구 정보 저장 간격(A): 10 분(M)

☒ 저장하지 않고 닫는 경우 마지막으로 자동 저장된 버전을 유지합니다.

자동 복구 파일 위치(R): C:\Users\...#AppData#Roaming#... 찾아보기(B)...

기본 파일 위치(I): C:\Users\...#Documents#... 찾아보기(B)...

(b) 자동 저장 기능

[예외에 대비한 사례]

01. 예외 처리

■ 예측 가능한 예외와 예측 불가능한 예외

- **예측 가능한 예외** : 발생 여부를 개발자가 사전에 인지할 수 있는 예외이다. 개발자는 예외를 예측하여 명시적으로 예외가 발생할 때는 어떻게 대응하라고 할 수 있다. 대표적으로 사용자 입력란에 값이 잘못 들어갔다면, if문을 사용하여 사용자에게 잘못 입력하였다고 응답하는 방법이 있다. 매우 쉽게 대응할 수 있다
- **예측 불가능한 예외** : 대표적으로 매우 많은 파일을 처리할 때 문제가 발생할 수 있다. 예측 불가능한 예외가 발생했을 경우, 인터프리터가 자동으로 이것이 예외라고 사용자에게 알려준다. 대부분은 이러한 예외가 발생하면서 프로그램이 종료되므로 적절한 조치가 필요하다.

01. 예외 처리

■ 예외 처리 구문 : **try -except**문

- 파이썬 예외 처리의 기본 문법은 try -except문이다.

```
try:
```

```
    예외 발생 가능 코드
```

```
except 예외 타입:
```

```
    예외 발생 시 실행되는 코드
```

01. 예외 처리

■ 예외 처리 구문 : try -except문

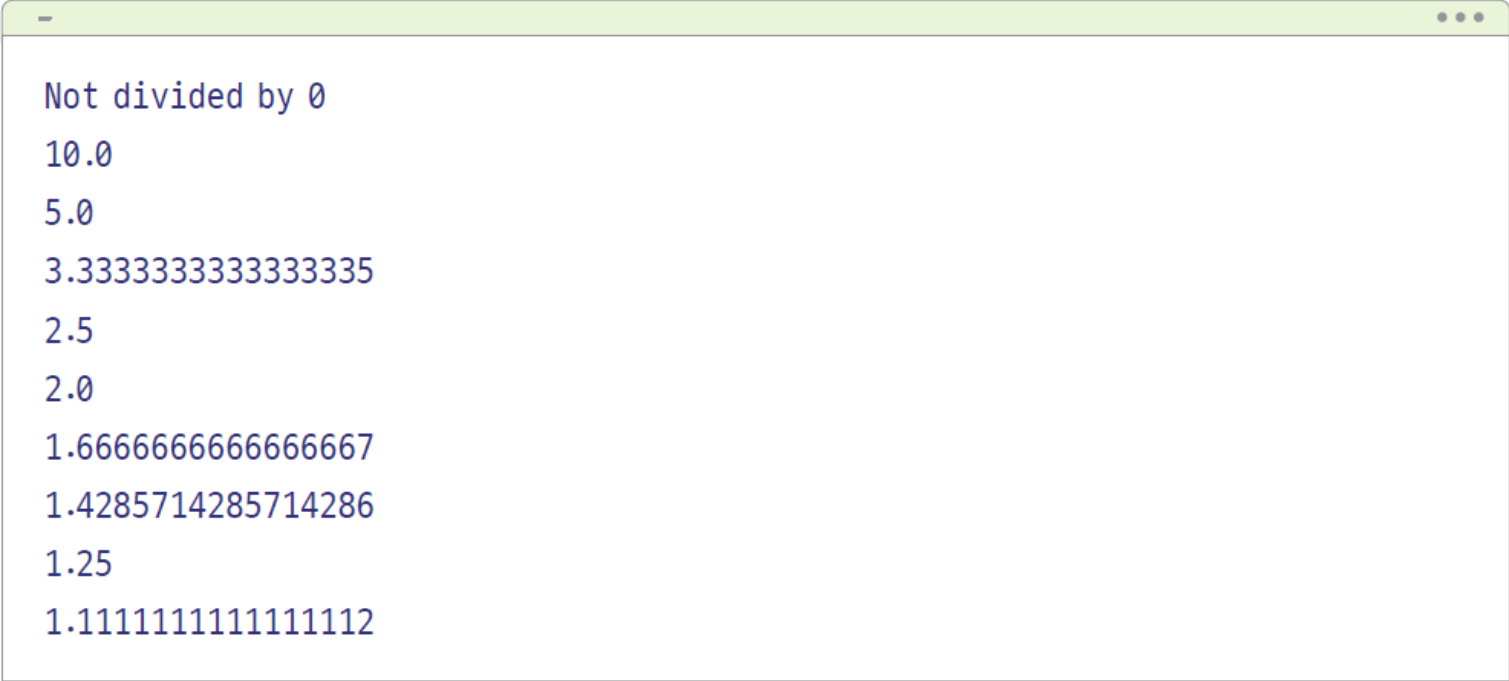
- 간단한 코드를 만들어 보자. [코드 12-1]은 0부터 9까지의 숫자를 i에 하나씩 할당하면서 10으로 나눈 값을 출력하는 코드이다. 이 프로그램은 1이 아닌 0부터 시작하다 보니 10을 0으로 나누는 계산이 가장 먼저 실행된다. 처음에 '10÷0(10/0)'을 하면 0으로는 10을 나눌 수 없으므로 예외가 발생한다. 하지만 이미 이러한 예외의 발생은 예상 가능하므로 try문으로 해당 예외가 발생할 때를 대비할 수 있다. ZeroDivisionError, 즉 0으로 나뉘진 경우에는 except 문 안으로 들어가 해당 구문에서 처리하는 코드가 정의된다. 여기서는 print("Not divided by 0") 코드가 실행된다.

코드 12-1 try-except.py

```
1 for i in range(10):
2     try:
3         print(10 / i)
4     except ZeroDivisionError:
5         print("Not divided by 0")
```


01. 예외 처리

■ 예외 처리 구문 : **try -except**문



```
Not divided by 0
10.0
5.0
3.3333333333333335
2.5
2.0
1.6666666666666667
1.4285714285714286
1.25
1.1111111111111112
```

- 그런데 만약 여기서 try문이 for문 밖으로 나가면 어떤 일이 발생할까? 이 경우, 이 반복문 전체가 종료된다. 즉, try문 내부에서 예외가 발생하면 except문 영역에서 코드가 실행되고, try except문이 종료된다. 이러한 이유로 try문을 적당한 곳에 삽입하여 예외 처리를 해야 한다.

01. 예외 처리

여기서 잠깐! 예외의 종류와 예외 에러 메시지

- 예외의 종류

예외	내용
IndexError	리스트의 인덱스 범위를 넘어갈 때
NameError	존재하지 않는 변수를 호출할 때
ZeroDivisionError	0으로 숫자를 나눌 때
ValueError	변환할 수 없는 문자나 숫자를 변환할 때
FileNotFoundError	존재하지 않는 파일을 호출할 때

01. 예외 처리

여기서 잠깐! 예외의 종류와 예외 에러 메시지

- **예외 에러 메시지** : 내장 예외와 함께 사용하기 좋은 것이 예외 에러 메시지이다. [코드 12-2]와 같이 except문의 마지막에 'as e' 또는 'as 변수명'을 입력하고, 해당 변수명을 출력하면 된다. 실행 결과 'division by zero'라는 에러 메시지를 확인할 수 있는데, 이 에러 메시지는 파이썬 개발자들이 사전에 정의한 것으로, 특정한 에러를 빠르게 이해할 수 있도록 돕는다.

코드 12-2 error_message.py

```
1 for i in range(10):
2     try:
3         print(10 / i)
4     except ZeroDivisionError as e:
5         print(e)
6         print("Not divided by 0")
```

01. 예외 처리

여기서  잠깐! 예외의 종류와 예외 에러 메시지

```
division by zero
Not divided by 0
10.0
5.0
3.3333333333333335
2.5
2.0
1.6666666666666667
1.4285714285714286
1.25
1.1111111111111112
```

01. 예외 처리

■ 예외 처리 구문 : **try-except-else**문

- try-except-else문은 if-else문과 비슷한데, 해당 예외가 발생하지 않을 경우 수행할 코드를 else문에 작성하면 된다.

```
try:  
    예외 발생 가능 코드  
except 예외 타입:  
    예외 발생 시 실행되는 코드  
else:  
    예외가 발생하지 않을 때 실행되는 코드
```

01. 예외 처리

■ 예외 처리 구문 : **try-except-else**문

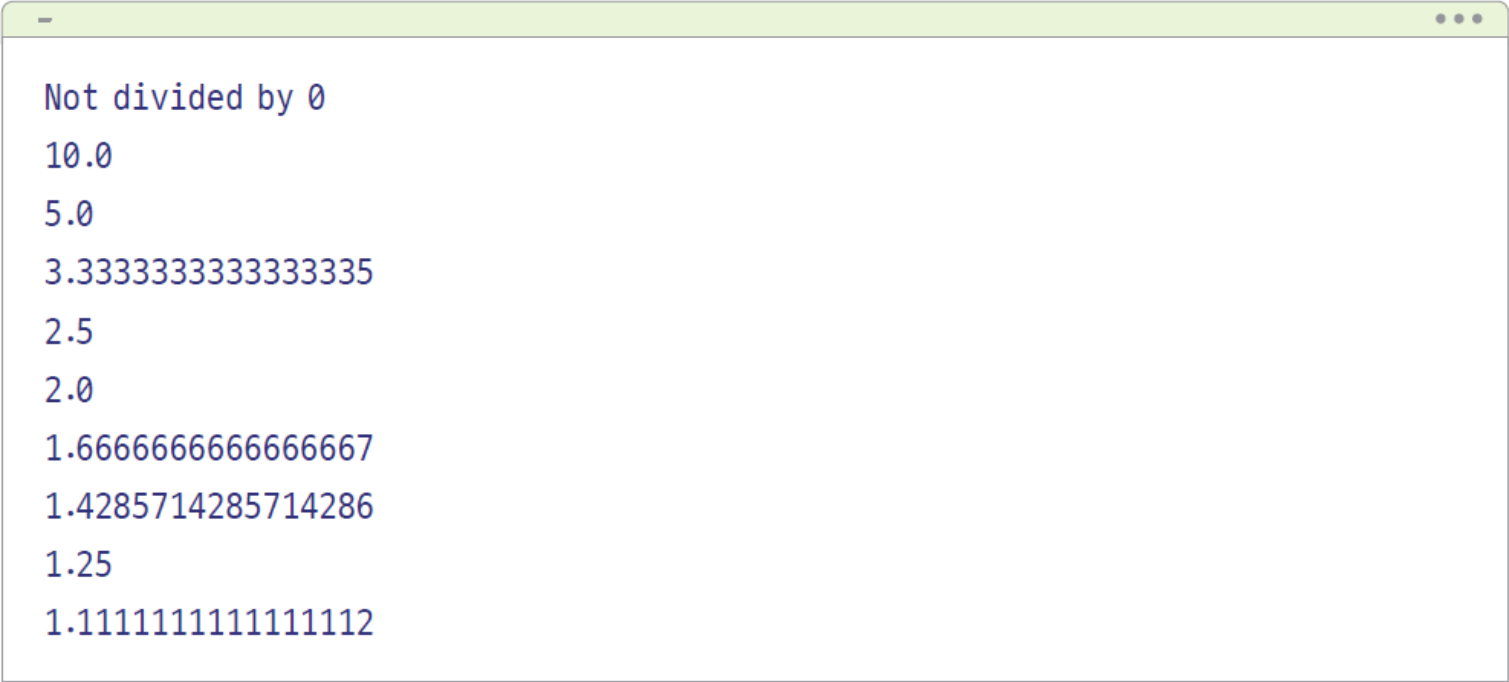
- [코드 12-3]은 10을 i로 나누는 코드를 실행하여 제대로 나누었을 경우 else문에 의해 결과가 화면에 출력되고, 그렇지 않을 경우 사전에 정의된 except문에 의해 에러가 발생하는 코드이다.

코드 12-3 try-except-else.py

```
1 for i in range(10):
2     try:
3         result = 10 / i
4     except ZeroDivisionError:
5         print("Not divided by 0")
6     else:
7         print(10 / i)
```

01. 예외 처리

■ 예외 처리 구문 : **try-except-else**문



```
Not divided by 0
10.0
5.0
3.3333333333333335
2.5
2.0
1.6666666666666667
1.4285714285714286
1.25
1.1111111111111112
```

01. 예외 처리

■ 예외 처리 구문 : **try-except-finally**문

- try-except-finally문에서 finally문은 try-except문 안에 있는 수행 코드가 아무런 문제 없이 종료되었을 경우, 최종으로 호출하는 코드이다

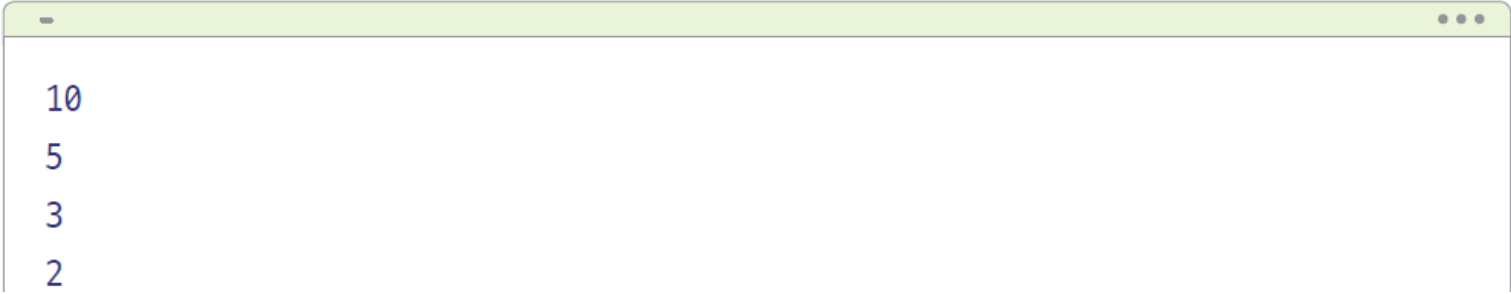
```
try:  
    예외 발생 가능 코드  
except 예외 타입:  
    예외 발생 시 실행되는 코드  
finally:  
    예외 발생 여부와 상관없이 실행되는 코드
```


01. 예외 처리

■ 예외 처리 구문 : **try-except-finally**문

코드 12-4 try-except-finally.py

```
1 try:
2     for i in range(1, 10):
3         result = 10 // i
4         print(result)
5 except ZeroDivisionError:
6     print("Not divided by 0")
7 finally:
8     print("종료되었다.")
```



```
10
5
3
2
```

01. 예외 처리

■ 예외 처리 구문 : **try-except-finally**문

```
2  
1  
1  
1  
1  
1  
종료되었다.
```

- ➡ 이 코드는 try문이 for문 밖으로 나가 i가 1부터 시작한다. 사실상 ZeroDivisionError가 발생할 수 없는 코드이다. 이러한 코드를 작성하면 except문을 사용할 수 없고, 마지막으로 finally문만 실행된다. try-except-finally문도 for문에서 finally문을 사용하는 것과 동일하게 예외 발생 여부와 상관없이 반드시 실행되는 코드이다.

01. 예외 처리

■ 예외 처리 구문 : **raise**문

- raise문은 try-except문과 달리 필요할 때 예외를 발생시키는 코드이다.

```
raise 예외 타입(예외 정보)
```

01. 예외 처리

■ 예외 처리 구문 : **raise**문

코드 12-5 raise.py

```
1 while True:
2     value =input("변환할 정수값을 입력해 주세요: ")
3     for digit in value:
4         if digit not in "0123456789":
5             raise ValueError("숫자값을 입력하지 않았습니다.")
6     print("정수값으로 변환된 숫자 -", int(value))
```

변환할 정수값을 입력해 주세요: 10

정수값으로 변환된 숫자 - 10

변환할 정수값을 입력해 주세요: ab

Traceback (most recent call last):

File "raise.py", line 5, in <module>

raise ValueError("숫자값을 입력하지 않았습니다.")

ValueError: 숫자값을 입력하지 않았습니다.

01. 예외 처리

■ 예외 처리 구문 : **raise**문

- ➡ [코드 12-5]는 while True문으로 반복문이 계속 돌아가면서 사용자에게 입력을 받는다. 하지만 사용자가 입력한 값이 숫자가 아닌 경우에는 숫자값을 입력받지 않았다고 출력하면서 프로그램을 종료하는 것을 목적으로 작성된 프로그램이다. 이때, 예외의 종료는 ValueError로 화면에 출력된다. 사용자가 입력을 잘못했을 때, 입력이 잘못된 것을 알려 주면서 종료하는 프로그램이다.

01. 예외 처리

■ 예외 처리 구문 : **assert문**

- assert문은 미리 알아야 할 예외 정보가 조건에 만족하지 않을 경우, 예외를 발생시키는 구문이다.

`assert` 예외 조건

01. 예외 처리

■ 예외 처리 구문 : **assert**문

코드 12-6 assert.py

```
1 def get_binary_nmubmer(decimal_number):  
2     assert isinstance(decimal_number, int)  
3     return bin(decimal_number)  
4 print(get_binary_nmubmer(10))  
5 print(get_binary_nmubmer("10"))
```

```
0b1010                                ← 5행 실행 결과  
Traceback (most recent call last):    ← 6행 실행 결과  
  File "<assert.py>", line 5, in <module>  
    print(get_binary_nmubmer("10"))  
  File "assert.py", line 2, in get_binary_nmubmer  
    assert isinstance(decimal_number, int)  
AssertionError
```

01. 예외 처리

■ 예외 처리 구문 : **assert문**

- ➡ 1행에서 `get_binary_nmubmer()` 함수에 십진수가 들어온다. 하지만 함수를 사용하는 사용자가 잘못된 인수 `argument`, 예를 들어 문자열값을 입력할 수도 있다. 이를 방지하기 위해 2행에서 `assert문`을 사용하였다. `isinstance()` 함수는 입력된 값이 뒤에 있는 클래스의 인스턴스인지를 확인하는 함수이다. 이 코드에서 `decimal_number` 변수가 정수형인지는 5~6행에서 확인할 수 있다.
- `assert문`은 코드를 작성할 때 잘못된 입력 여부를 사전에 확인하여 나중에 필요 없는 연산을 막아 주며, 다른 사람이 만든 코드를 사용하는 데 좋은 가이드가 될 수 있다.