5. Classes and Interfaces

▼ Item 37: Compose Classes Instead of Nesting Many Levels of Built-in Types

1. 클래스 사용

2. 딕셔너리와 그에 관련된 built-in 타입들은 취약한 코드(brittle code)를 작성하도록 과도하게 확장할 위험이 있다.

→ inner dictionary는 defaultdict 사용 (from collections import defaultdict)

- **collections.defaultdict**

  : defaultdict 클래스의 생성자로 기본값을 생성해주는 함수를 넘기면, 키에 대응하는 값이 없는 경우에는 생성자의 인자로 넘어온 함수를 호출해서 그 결과 값을 값으로 설정한다.

  ```
  lt = defaultdict(list)
  ```

  이렇게 하면 lt의 특정 키에 대한 값을 정해주지 않으면, 기본값인 빈 리스트가 값으로 들어가게 된다.

- **brittle code 가 무엇일까?**

3. 튜플 사용

4. 딕셔너리와 튜플은 internal bookkeeping에 레이어를 계속 쌓아가게 하는 위험성이 있다.

→ namedtuple 사용 (from collections import namedtuple)

- **collections.namedtuple**

  : 클래스처럼 객체를 생성할 수 있으며, 튜플처럼 값을 변경할 수 없고, 딕셔너리처럼 값에 대한 label을 부여할 수 있다.

  ```
  from collections import namedtuple

  Book = namedtuple('Book', ['title', 'price'])
  mybook = Book('Effective Python', 25000)
  ```

```
# label 붙일 수 있다.
print(mybook.title, mybook.price)
# indexing 가능
print(mybook[0], mybook[1])
```

- **bookkeeping이 무엇일까?**

[Things to Remeber]

- Avoid making dictionaries with values that are dictionaries, long tuples, or complex nestings of other built-in types.
- Use namedtuple for lightweight, immutable data containers before you need the flexibility of a full class.
- Move your bookkeeping code to using multiple classes when your internal state dictionaries get complicated.

▼ Item 38: Accept Functions Instead of Classes for Simple Interfaces

1. Hook

Many of Python’s built-in APIs allow you to customize behavior by passing in a function. These ***hooks*** are used by APIs to call back your code while they execute.

(파이썬의 많은 내장 API들은 함수를 전달함으로써 동작을 사용자 정의할 수 있게 한다. 이러한 “hook” 들은 API들이 실행되는 동안 코드를 다시 호출하는 데 사용된다.)

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=len)
print(names)
```

여기서 리스트를 정렬할 때 길이를 기준으로 했다. len 이라는 내장 함수를 key hook으로 제공한 것이다.

2. hook으로는 함수가 이상적이다

Functions are easier to describe and simpler to define than classes.

(함수가 클래스 보다 설명하기 쉽고 정의하기에 간단하다.)

3. stateful closure한 동작을 제공하기 위해 helper class를 사용하면 명확해진다

4. __call__ 은 object를 함수처럼 불러올 수 있게 한다

[Things to Remember]

- Instead of defining and instantiating classes, you can often simply use functions for simple interfaces between components in Python.
- References to functions and methods in Python are first class, meaning they can be used in expressions (like any other type).
- The __call__ special method enables instances of a class to be called like plain Python functions.
- When you need a function to maintain state, consider defining a class that provides the __call__ method instead of defining a stateful closure.

▼ Item 39: Use @classmethod Polymorphism to Construct Objects Generically

```
# 다형성(Polymorphism)
# 상위 클래스
class InputData:
  def read(self):
    raise NotImplementedError

# 하위 클래스
class PathInputData(InputData):
  def __init__(self, path):
    super().__init__()
    self.path = path
  def read(self):
    with open(self.path) as f:
    return f.read()

# 상위 클래스
class Worker:
  def __init__(self, input_data):
    self.input_data = input_data
    self.result = None
  def map(self):
    raise NotImplementedError
  def reduce(self, other):
    raise NotImplementedError
```

```
# 하위 클래스
class LineCountWorker(Worker):
  def map(self):
    data = self.input_data.read()
    self.result = data.count('\n')
  def reduce(self, other):
    self.result += other.result

# 이 조각들(따로따로 구현된 클래스들)을 어떻게 연결할까?
# -> helper function을 만든다
import os

# input data 생성(helper function)
def generate_inputs(data_dir):
  for name in os.listdir(data_dir):
    yield PathInputData(os.path.join(data_dir, name))

# LineCountWorker 생성(helper function)
def create_workers(input_list):
  workers = []
  for input_data in input_list:
    workers.append(LineCountWorker(input_data))
  return workers


from threading import Thread

# reduce 수행(helper function)
def execute(workers):
  threads = [Thread(target=w.map) for w in workers]
  for thread in threads: thread.start()
  for thread in threads: thread.join()

  first, *rest = workers
  for worker in rest:
    first.reduce(worker)
  return first.result

# 모든 helper functions 조립(helper function)
def mapreduce(data_dir):
  inputs = generate_inputs(data_dir)
  workers = create_workers(inputs)
  return execute(workers)

# ==============> 전혀 제네릭 하지 않다는 문제점
class GenericInputData:
  def read(self):
    raise NotImplementedError
  @classmethod
  def generate_inputs(cls, config):
    raise NotImplementedError

class PathInputData(GenericInputData):
  ...
  @classmethod
  def generate_inputs(cls, config):
    data_dir = config['data_dir']
```

```
    for name in os.listdir(data_dir):
      yield cls(os.path.join(data_dir, name))

class GenericWorker:
  def __init__(self, input_data):
    self.input_data = input_data
    self.result = None
  def map(self):
    raise NotImplementedError
  def reduce(self, other):
    raise NotImplementedError

  @classmethod
  def create_workers(cls, input_class, config):
    workers = []
    for input_data in input_class.generate_inputs(config):
      workers.append(cls(input_data))
    return workers
```

[Things to Remember]

- Python only supports a single constructor per class: the **__init__** method.
- Use @classmethod to define alternative constructors for your classes.
- Use class method polymorphism to provide generic ways to build and connect many concrete subclasses.

▼ Item 40: Initialize Parent Classes with super

문제1 : __init__ 의 호출 순서가 모든 하위 클래스에 대해 구체적(specified)이지 않다.

문제2 : 다이아몬드 상속 문제

Diamond inheritance happens when a subclass inherits from two separate classes that have the same superclass somewhere in the hierarchy.

Diamond inheritance causes the common superclass's __init__ method to run multiple times, causing unexpected behavior.

(다이아몬드 상속 문제는 하위 클래스가 같은 계층의 상위 클래스를 가지는 두 개의 개별적인 클래스로부터 상속 받을 때 발생한다.

다이아몬드 상속 문제는 공통된 상위 클래스의 __init__ 메소드가 여러 번 실행되어 예상치 못한 행동을 하게 만든다.)

→ 2개의 문제 해결을 위해 파이썬은 super 라는 내장 함수가 있고, 표준 메서드 결정 순서(MRO)가 있다.

[Things to Remember]

- Python’s standard method resolution order (MRO) solves the problems of superclass initialization order and diamond inheritance.
- Use the super built-in function with zero arguments to initialize parent classes.

▼ Item 41: Consider Composing Functionality with Mix-in Classes

1. Mix-in Class

A mix-in is a class that defines only a small set of additional methods for its child classes to
provide.

Mix-in classes don’t define their own instance attributes nor require their **__init__** constructor to be called.

(mix-in 클래스는 그것의 자식 클래스들이 제공할 추가 메서드의 작은 집합만을 정의한다.

mix-in 클래스는 그들의 자체 인스턴스 속성을 정의하지 않으며, __init__ 생성자를 불러올 필요가 없다.)

[Things to Remember]

- Avoid using multiple inheritance with instance attributes and __init__ if mix-in classes can achieve the same outcome.
- Use pluggable behaviors at the instance level to provide per-class customization when mix-in classes may require it.
- Mix-ins can include instance methods or class methods, depending on your needs.
- Compose mix-ins to create complex functionality from simple behaviors.

▼ Item 42: Prefer Public Attributes Over Private Ones

1. 파이썬에는 클래스 속성의 가시성은 두 가지 종류 뿐이다: Public과 Private

- ***public***: can be accessed by anyone using the dot operator on the object
- ***private***: be specified by prefixing an attribute's name with a double underscore. They can be accessed directly by methods of the containing class

2. Private 속성에 대한 문법은 사실 가시성을 엄격히 적용하지 않는다

그 이유는 파이썬의 모토에서 간단히 찾을 수 있다.

: "We are all consenting adults here." (우리는 모두 책임질 줄 아는 어른이다.)

→ 우리가 하고 싶어하는 것을 막는 언어는 필요없다는 뜻

[Things to Remember]

- Private attributes aren't rigorously enforced by the Python compiler.
- Plan from the beginning to allow subclasses to do more with your internal APIs and attributes instead of choosing to lock them out.
- Use documentation of protected fields to guide subclasses instead of trying to force access control with private attributes.
- Only consider using private attributes to avoid naming conflicts with subclasses that are out of your control.

▼ Item 43: Inherit from collections.abc for Custom Container Types

[Things to Remember]

- Inherit directly from Python's container types (like list or dict) for simple use cases.
- Beware of the large number of methods required to implement custom container types correctly.
- Have your custom container types inherit from the interfaces defined in collections.abc to ensure that your classes match required interfaces and behaviors.