# Python 프로그래밍 고급 <Pycharm 설치 및 사용>

**IDE (Integrated Development Environment) = 통합 개발 환경**
코딩, 디버그, 컴파일, 배포 등 프로그램 개발에 관련된 모든 작업을 하나의 프로그램 안에서 처리하는 환경을 제공하는 소프트웨어

**Anaconda**
파이썬 패키지들의 dependency만을 고려하는 pip과 달리, 파이썬이 아닌 library와의 dependency까지 고려하여 특정 패키지를 온전히 구동할 수 있도록 하는 배포판
- 자주 사용되는 python package들을 이미 포함 -> 별도의 추가 설치 없이 사용 가능
- 프로그램이 무겁고 불필요한 패키지들까지 함께 설치
pip의 패키지 관리 기능과 venv의 가상 환경 구축 기능을 담은 conda라는 프로그램이 함께 설치

**Pycharm**
프로젝트
- 특정한 프로그래밍 목적을 달성하기 위해 밀접하게 관련 있는 소스 파일들을 하나의 프로젝트에서 관리
- 프로젝트마다 사용할 파이썬 인터프리터를 비롯하여 다양한 설정들을 따로 지정할 수 있음
-> 프로젝트마다 요구되는 파이썬 버전, 외부 라이브러리 목록이 다를 수 있으므로, 따로 관리할 필요성이 있음

# Python 프로그래밍 고급 <함수>

**함수**

프로그램 내에서 동일한 기능이 반복적으로 사용될 때, 해당 부분을 함수로 정의
기능 상에서 변화가 필요한 부분을 입력으로 받아, 입력에 대한 결과를 얻을 수 있음

**함수의 일반적인 형태**

헤더
- def 키워드로 함수를 정의할 것임을 표시하며, def 뒤에는 함수명이 등장
- 이때 변수 선언시와 마찬가지로 적절한 형태의 식별자를 사용
- 함수명 뒤에 괄호로 묶여진 부분에 함수에서 사용할 파라미터들이 정의
- 맨 뒤에 반드시 콜론(:) 기호 포함

ex) def max(num1, num2): -> def 함수이름 (매개변수1, 매개변수2):

바디
- 함수가 호출되었을 때, 실행될 문장들의 나열
- 헤더에서 선언된 파라미터들은 바디 내에서 변수처럼 활용 가능
- 헤더 문장보다 반드시 한 단계 더 들여쓰기 되어야 함
- return 키워드를 통해 함수의 출력 결과를 정의

**함수 호출**

호출 방법
- 함수명과 괄호로 묶여진 파라미터들을 argument를 통해 함수를 호출
- 함수를 정의할 때, 헤더에서 선언한 파라미터 순서대로 arguments 각각의 값이 할당
- arguments의 개수는 함수 선언 시의 파라미터 수와 일반적으로 일치해야 함
- 함수의 호출은 함수가 정의된 이후에만 가능

함수 호출의 결과
- 함수 호출 부분은 해당 함수가 반환한 결과값으로 여기고 활용할 수 있음
- 출력 결과값 활용 (변수에 값 할당, 추가 연산 수행)
- 출력 결과값 비활용 (함수를 실행만 하고, 결과값을 이용하지 않음)

**함수의 다양한 형태**

입력 파라미터가 존재하지 않는 함수

- 빈 괄호를 통해서 파라미터가 없는 함수도 정의 가능 ex) def function_name()

**with/without return**

with return

- return 뒤에는 값, 변수 혹은 표현식이 쓰일 수 있으며, 결과적으로 값이 반환
- return을 통해 여러 값을 반환할 수 있음, 해당 경우 반환되는 값의 개수만큼 변수 할당
- 만약 하나의 변수에 값을 할당한다면, tuple 객체가 반환
- return 뒤에 아무 것도 적지 않는다면, None이라는 객체가 존재하지 않는 값 반환

without return

- 함수의 바디가 모두 실행된 뒤에 종료
- return 뒤에 값을 적지 않는 경우와 마찬가지로, 실제로는 None을 반환
- 반환 값이 없는 경우에는 해당 함수는 None을 반환하므로 결과값을 이용할 수 없음

**튜플 자료형**

리스트와 유사하지만 immutable (불변)인 자료형

리스트와의 공통점

- 임의의 자료형을 item으로 포함할 수 있고, 인덱싱, 슬라이싱을 동일하게 사용 가능
- concatenation 연산, 반복 연산 동일하게 사용 가능

리스트와의 차이점

- 문자열과 마찬가지로 값을 수정할 수 없음

**Mutable vs Immutable**

Immutable

- 객체의 값을 수정할 수 없음
- 객체의 값으로 연산을 수행할 경우, 새로운 객체가 생성됨
- int, float, str

Mutable

- 객체의 값을 수정할 수 있음
- 새로운 객체의 생성 없이, 해당 객체의 값을 변경 가능
- 새로운 객체를 생성하는 것도 가능
- list -> slicing이 수행되면 새로운 객체가 생성 : 복제되어 새로운 객체가 생성되기 때문

**The Scope of Variables**

지역변수: 함수 내에서 선언된 변수는 함수 내에서만 접근 가능
전역변수: 함수 밖에서 선언된 변수들은 어디서든 접근 가능

지역변수를 함수 밖에서 사용하는 경우
- 전역변수는 함수 밖에서 사용 가능
- 지역변수를 함수 밖에서 사용하려 할 경우 에러 발생

함수 내에서 전역변수에 값을 할당하는 경우 -> 동일한 이름의 지역변수를 새로 생성한 것
함수의 바디 내에서 동일한 이름의 전역변수는 함수 내에서 사용 불가능

함수 내에서 선언된 변수만이 지역변수, 제어문이나 반복문에서 생성된 변수는 전역변수
- if 문 내에서 생성된 변수는 해당 조건이 만족되지 않을 경우 생성되지 않음

동일한 변수명을 각기 다른 함수에서 사용하는 경우는 가능
- conflict 없이 각각 독립적으로 지역변수가 생성되기 때문

**Pass by Value**

Passing arguments by reference values
- 파이썬의 모든 데이터는 객체이고, 변수들은 이러한 객체를 참조하고 있음
- arguments와 함께 함수가 호출될 때에, 함수의 파라미터는 arguments가 참조하고 있는 개체를 참조 -> 즉, 변수가 참조하고 있는 값 (객체) 자체를 전달 : pass by value

Pass by Value - immutable objects
- 숫자형 데이터의 경우 함수 내의 파라미터에 전역변수가 참조하고 있는 객체가 할당
- 함수 내에서 연산을 수행한 경우, 객체 자체가 수정되지 못하고 새로운 객체가 생성
- 전역변수가 참조하고 있는 객체는 그대로
- 문자열 데이터의 경우도 마찬가지

Pass by Value - mutable objects
- 리스트 데이터의 경우 함수 내의 파라미터에 전역변수가 참조하고 있는 객체가 할당
- 함수 내에서 연산을 수행한 경우, 객체 자체가 수정
- 전역변수는 동일한 객체를 참조하고 있으므로, 함께 수정됨
- 인덱싱을 이용하여 함수 내에서 연산을 수행한 경우, 객체 자체가 수정되지만, 전역 변수가 참조하고 있는 객체는 그대로 유지
- 함수 내에서 파라미터에 x[:]를 이용하여 새로운 객체를 생성해도 같은 결과
- 슬라이싱을 통한 새로운 객체를 생성하더라도 리스트 객체 내의 아이템 자체가 mutable objects인 경우 (예를 들어 리스트 내에 int가 아닌 list가 존재) recursively (재귀적으로) 완전한 복제가 이루어지지 않음
- copy.deepcopy() 함수를 통해 재귀적으로 완전히 새로운 객체를 얻는 것이 가능

**Positional & Keyword Arguments**

Keyword Arguments

- 함수의 파라미터 이름을 이용하여 argument가 어느 parameter에 할당되는지 명시적으로 나타낼 수 있음

Arguments 대입 방법

Positional Arugments의 방법은 말 그대로 매개변수의 위치상 순차적으로 대입

Keyword Arguments의 방법은 순서 변경도 가능하며, 파라미터의 이름을 지정해주는 방식

- 한번 keyword 방식으로 argument를 대입한 경우, 그 이후의 매개변수는 모두 keyword 방식을 사용해야 함
- keyword 방식에서 순서가 보장되지 않으므로, 이후의 argument와 parameter 간의 매칭이 확실하지 않기 때문

**Default Argumetns**

함수 선언 시에 파라미터가 가질 default value를 명시하면, arguments가 주어지지 않았을 시 해당 값을 사용

함수 선언 시, default value가 주어지는 파라미터와 주어지지 않는 파라미터가 혼재된 경우, default value가 주어지지 않은 파라미터가 먼저 등장해야함

-> 즉 파라미터를 default로 지정하는 것은 뒤에서부터 가능

# Python 프로그래밍 고급 <클래스와 객체>

**객체지향과 절차지향**

절차지향 프로그래밍

- 동사 중심의 프로그래밍 방식
- 절차지향에서 전체 과정을 나누어 처리하는 단위를 함수
- 문제를 여러 가지 작은 함수로 나누어 문제를 해결하는 방식
- 고객이 자판기에서 제품을 구매하는 일련의 과정을 작은 단위의 함수로 구분하여 나열
- 자판기 이용에 필요한 기능을 함수로 모듈화
- 100명에 대해 각각 금액, 구매한 제품, 구매시각을 나타내는 전역변수를 모두 생성

객체지향 프로그래밍

- 명사 중심의 프로그래밍 방식
- 문제를 구성하는 객체 (instance)를 만들어 이 객체들 간의 메시지 교환으로 문제를 해결
- 고객과 자판기를 객체로 설정
- 고객 객체는 돈, 제품 2가지 속성, 돈을 넣고, 돈을 받고, 제품을 받는 3개의 행동 가능
- 자판기 객체는 돈, 제품 2가지 속성, 돈을 받고, 돈을 검사하고, 잔돈을 주고, 제품을 주는 4가지 행동이 가능
- 자판기에 필요한 기능 뿐만 아니라, 데이터까지도 모듈화
- 100명을 각각 객체로 생성하여, 금액, 구매한 제품, 구매 시각 등은 객체 내부에 저장

**객체지향 프로그래밍의 특성**

추상화

- 객체들의 공통적인 속성 (attributes)과 기능 (method)을 묶어 이름 붙이는 것
- 프로그램에 필요한 정보만 간추려 구성
- 객체지향 프로그래밍에서 클래스를 정의하는 부분

상속

- 자식 클래스는 부모 클래스의 속성과 기능을 그대로 물려 받음
- 자식 클래스는 부모 클래스에는 없는 추가적인 속성과 기능을 가질 수 있음
- 부모 클래스를 활용하여 자식 클래스를 쉽게 개발하고, 코드의 중복을 줄여주는 코드의 재사용성이 높아짐
- 부모 클래스를 유지 보수함으로써 자식 클래스들을 한번에 수정이 가능하다는 유지 보수의 편리성

다형성

- 하나의 함수명으로 정의된 기능이, 상속 과정에서 각기 다른 방식으로 작동할 수 있게 함
- 각기 다른 기능이지만, 공통의 부모클래스를 갖고 있기 때문에 같은 함수명으로 실행
- 예를 들어 동물 클래스 내에서 울음소리에 대한 메서드 -> 동물들마다 다른 방식으로 실행

캡슐화

- 세부 작동 방식을 외부에 드러내지 않고, 내부로 감추는 것을 의미
- 외부에서는 인터페이스에만 접근할 수 있음
- 예를 들어 자동차에서 엔진의 구체적인 작동과정은 모르지만, 핸들, 페달 등의 인터페이스만을 이용해 기능을 수행

프로그래밍 내에서의 캡슐화

- 데이터와 기능을 클래스 내부로 감춤 -> 클래스를 정의하는 것 자체가 일종의 캡슐화
- 사용자가 객체를 다룰 때 필요한 인터페이스 (함수)만을 제공하고, 나머지 변수 및 함수들은 사용자가 접근할 수 없으며, 객체 내부에서만 접근 가능하도록 함

**클래스와 객체**

클래스

- 객체의 공통적인 특징을 기술
- 객체의 속성 (arrtibute)과 기능 (method)를 포함

객체

- 클래스를 구체화하여 생성된 물리적 공간을 갖는 구체적인 실체
- 같은 클래스로부터 생성된 객체들이더라도 서로 독립적으로 존재
- 한 객체의 속성이 변하더라도 나머지 객체들은 영향을 받지 않음
- 객체는 클래스의 인스턴스를 의미
- 예를 들어 자동차는 클래스, 실제 도로 위에 존재하는 차량들은 객체

**클래스의 구성**

attributes

- 클래스로부터 생성된 객체의 속성 정보를 나타냄
- 변수와 변수에 할당된 값으로 표현
- properties, data fields, 멤버 변수 등 다양하게 불림

methods

- 클래스로부터 생성한 객체가 수행할 수 있는 함수들

**클래스 정의하기**

헤더

- class 클래스명: 으로 정의 // 클래스명은 식별자의 형태로 관례적으로 CamelCase 형태

바디

- 한 단계 들여쓰기하여 작성, 클래스로부터 생성된 객체들이 사용할 함수가 정의

클래스 내부에 정의된 함수 (method)

- 일반적인 함수들과 달리 첫 번째 파라미터가 self이어야 함

self 파라미터

- self 파라미터는 객체 스스로를 참조하는 파라미터
- 모든 method에 self 파라미터가 존재하므로, 객체의 attributes와 methods에 접근 가능

__init__ 함수

- 객체를 초기화하는 생성자 함수를 말하며, 객체 생성 시 자동으로 호출
- self 이외의 파라미터를 통해 객체를 생성할 때 특정한 값을 받아서 사용 가능
- attributes는 가능하면 생성자 함수 내에서 선언
- 선언된 attributes들은 다른 methods에서도 self 파라미터를 이용하여 접근 가능

**객체 생성하기**

객체 생성 방법

- 클래스명과 () 기호를 통해 객체 생성
- 생성한 객체는 변수에 할당하여 활용 가능
- 클래스의 __init__ 함수에 self 이외의 파라미터가 정의되어 있을 경우, 객체 생성 시 argument를 대입해주어야 함

객체 생성과정

- 객체 생성 후 __init__ 함수가 호cnfehlauss서 생성된 객체와 입력 받은 파라미터를 전달
- 객체 내에 파라미터로 입력 받은 attribute가 생성, 파라미터가 가리키는 값으로 초기화
- 나머지 attribute들도 초기화된 이후에 객체가 반환
- 변수는 반환된 객체를 참조하는 방식

Accessing attributes and methods
접근 방법
- 객체가 할당된 변수 뒤에 . 기호와 함께 접근하고자 하는 attribute명, method명을 통해 접근 가능
- method를 실행하고자 할 때는 ()와 함께 적절하게 argument를 대입
- 객체 내의 atttribute에 새로운 값을 할당할 수도 있음 // ex) self.att = 'hi'
- self 파라미터에도 역시 객체가 할당되어 있기 때문에 같은 방법으로 접근 가능

**method 실행 시, arguments의 전달**
- 객체 스스로 self 파라미터에 전달하며, 이는 명시적으로 arguments로 대입하지 않더라도 자동적으로 수행
- 나머지 파라미터에 대해서는 일반적인 파라미터와 마찬가지로 arguments가 전달됨

**클래스 내 변수의 scope**
지역변수의 경우는 self를 사용하지 않고, attributes의 경우에는 self가 붙어 있음
함수의 파라미터 혹은 함수의 바디에서 생성된 변수는 지역변수로 함수 내에서만 활용 가능
attributes는 self.attribute_name으로 객체 내에서 어디서든 접근 가능
- 객체를 담고 있는 지역변수 self가 모든 함수에 파라미터로 정의되어 있기 때문

**self 파라미터가 활용되지 않은 method**
self 파라미터를 통해 객체 내의 attribute 또는 method에 접근하지 않은 함수도 정의 가능
하지만 이러한 method는 굳이 클래스 내에서 정의할 이유가 없음

**without __init__ method**
클래스 내에서 정의할 attributes가 없을 때는 __init__ 함수 생략 가능
__init__ method가 아닌 다른 method 내에서도 attribute를 선언할 수 있음
- attribute를 선언하는 문장이 실행되기 전 다른 method에서 해당 attribute에 접근시 에러
- attribute 값의 변경은 다른 method에서 하더라도, 최초의 선언은 __init__에서

**attributes, methods 숨기기**
attribute, method의 이름 앞부분에 '_' 기호를 두 번 연속으로 적음
▶ 이를 통해 객체 외부에서 직접 조회를 할 수 없도록 할 수 있음 ex) self.__att = att
▶ 접근을 원할 경우, get_attribute_name(), set_attribute_name()을 통해 접근 가능
하지만 '__' 을 통한 접근 제한은 권장되지 않음
▶ attribute의 이름을 수정하는 방법일 뿐, 완벽한 의미에서는 private 성질을 제공 X
▶ 본래의 목적은 클래스 간에 attribute 명이 겹치는 것을 방지하기 위한 기능
'_' 기호를 통해 객체 내부에서만 접근 가능하도록 함
▶ private 변수, method 임을 나타내는 관례적 용법
▶ 실제로는 외부에서 접근 가능
▶ 관례적 용법이지만 해당 방법을 통해 private임을 나타내는 것 권장 ex) self._att = att

**Passing Objects Arguments to Function**

일반적으로 정의한 클래스는 mutable한 객체를 생성

- arguments로 대입한 객체의 attributes를 함수 내에서 변경한 경우, 객체 자체가 수정되어 전역 변수도 수정된 객체를 참조

# Python 프로그래밍 고급 <상속과 다형성>

**상속**

상위 클래스의 속성과 기능을 하위 클래스에 물려주는 것

슈퍼 클래스

- 상속을 통해 속성과 기능을 물려주는 클래스
- 서브 클래스보다 더 추상화되는 클래스
- 상위 클래스, 부모 클래스 등으로 불림

서브 클래스

- 상속을 통해 속성과 기능을 물려 받는 클래스
- 슈퍼 클래스를 더 구체화한 클래스
- 슈퍼 클래스에는 없는 속성과 기능을 추가할 수 있음
- 슈퍼 클래스에서 물려 받은 기능을 수정할 수 있음
- 하위 클래스, 자식 클래스 등으로 불림

필요성

- 코드의 재사용성이 높아짐

▶ 슈퍼 클래스를 활용하여 서브 클래스 쉽게 개발

▶ 코드의 중복을 줄여줌 (상속을 활용하지 않을 경우, 동일한 attribute, method 중복)

▶ 파이썬에서 제공해주는 클래스 혹은 다른 사람이 구현해 놓은 클래스를 상속받아 커스터마이징할 수 있음

- 유지 보수의 편리성

▶ 슈퍼 클래스를 수정함으로써 서브 클래스들을 한번에 수정할 수 있음

**상속을 활용한 클래스 정의**

- 서브 클래스들에 공통적으로 등장할 attribute와 method를 포함하는 슈퍼 클래스를 정의
- 서브 클래스 정의 시, 슈퍼 클래스에 존재하지 않는 attribute, method 추가

**서브 클래스 정의하기**

서브 클래스의 헤더

- 슈퍼 클래스를 괄호 안에 넣어서 상속 관계를 나타냄

서브 클래스의 생성자 함수

- 반드시 슈퍼 클래스의 생성자 함수를 호출해야 함
- super()를 이용하여 슈퍼 클래스에 정의된 method에 접근 가능

▶ self 파라미터에 대응되는 argument는 자동으로 전달

**상속을 활용한 클래스의 객체 생성 과정**

- 클래스의 객체가 생성
- 클래스 내의 __init__ 함수가 자동으로 호출되며, 객체가 self 파라미터 argument로 대입
- super().__init__()에 의해 슈퍼 클래스의 __init__ 함수 호출, self 파라미터 argument로 대입
- 부모 클래스 attribute 생성, 자식 클래스 attribute 생성, 클래스 변수는 객체를 참조

**다형성**

- 하나의 함수명으로 정의된 기능이 상속 과정에서 각기 다른 방식으로 작동할 수 있도록 함
- 메서드 오버라이딩을 통해 슈퍼 클래스에 구현된 메서드를 재구현 가능

메서드 오버라이딩

- 슈퍼 클래스에 정의된 메서드와 동일한 이름의 함수를 서브 클래스 내에서 새롭게 정의
- 메서드의 헤더는 슈퍼 클래스의 헤더와 동일하며, 바디 부분만을 새롭게 정의

**Object - The base class**

- Object는 파이썬 내 모든 클래스의 최상위 클래스

▶ 클래스의 헤더에 상속받을 클래스를 명시하지 않을 경우, 자동적으로 Object 클래스 상속받음

- Object 클래스에는 몇 가지 특별한 메서드들이 정의, 파이썬 내 모든 클래스는 이를 상속받음

▶ __init__()

- 생성자 함수, 클래스를 정의할 때마다 object class 내 정의된 __init__ 메서드 오버라이딩

▶ __new__()

- 객체를 생성하고, 그 후 자동적으로 __init__() 호출

▶ __str__()

- 객체를 문자열로 형변환 하였을 때 호출되는 메서드
- Object Class에서는 클래스의 이름과 메모리 주소를 16진법으로 나타낸 값을 문자열로 반환

**동적 바인딩**

- 동일한 이름의 메서드가 오버라이딩에 의해 여러 클래스에서 각각 정의되어 있음
- 파이썬은 어떤 메서드를 실행할 것인지를 실행 시간에 결정

**연산자 오버로딩** ex) a+b = a.__add__(b)

메서드 오버로딩

- 동일한 이름의 메서드에 대해서 파라미터의 개수와 데이터 타입을 달리하여 정의
- 다양하게 arguments가 대입되었을 때, 이를 모두 처리 가능하도록 하여 편리성 향상
- 파이썬에서는 지원 X

▶ default 파라미터를 통해 aruguments 수를 조절하여 대입이 가능

▶ 데이터 타입을 강제하지 않기 때문에 메서드의 바디 내에서 다양한 데이터 타입에 대한 처리방식을 구현하는 것이 가능

연산자 오버로딩

- 연산자가 기본적으로 처리할 수 있는 데이터 타입 이외에 새로운 데이터 타입들을 처리
- 새롭게 정의한 클래스 내의 연산자에 대응되는 스페셜 메서드 재정의
- 새롭게 정의한 클래스의 객체에 대해서도 +, - 연산을 수행 가능하도록 함

**오버로딩 vs 오버라이딩**

- == 연산자에 대응되는 __eq__ 메서드는 __str__메서드와 마찬가지로 object class에 정의
- 새로운 클래스를 정의할 때, __eq__ 메서드를 재정의하는 것은 오버라이딩
- __add__ 메서드 같은 경우 object 클래스에 정의되어 있지 않음

▶ __add__ 메서드를 정의하는 것은 오버라이딩이 아님

▶ 그러나 연산자 ==, + 입장에서는 새롭게 정의한 클래스를 처리할 수 있도록 메서드가 재구현된 것이기 때문에 오버로딩이라는 공통적인 개념으로 설명할 수 있음

▶ 따라서 오버라이딩이 아닌 연산자 오버로딩이라고 부름

▶ 부르는 명칭은 다르나, __str__ 메서드를 오버라이딩 하는 것과 동일한 방식으로 스페셜 메서드를 재정의

▶ 새로 정의한 클래스의 객체에 대해서 다양한 연산을 직접 수행할 수 있음

# Python 프로그래밍 고급 <추상클래스, 모듈, 패키지>

**객체지향 프로그래밍의 특성**

상속, 다형성, 캡슐화, 추상화

**추상화**

- 객체들의 공통적인 속성 (attribute)과 기능 (method)을 묶어 이름을 붙이는 것
- 프로그램에 필요한 정보만을 간추려 구성
- 객체 지향 프로그래밍에서 클래스를 정의하는 것

슈퍼 클래스는 일종의 인터페이스만을 제공하고, 서브 클래스에서는 미구현된 메서드를 오버라이드하여 다양하게 활용하기 때문에 슈퍼 클래스에서는 메서드가 완벽히 구현되어 있을 필요는 없고 틀만 제공해도 됨

**슈퍼 클래스의 미구현 메서드**

- 미구현된 메서드를 서브 클래스에서 오버라이드하여 다양하게 활용
- 미구현된 메서드를 오버라이드하지 않고 사용해도 에러가 발생하지는 않음

▶ 복잡한 프로그램에서 이러한 미구현 메서드들을 모두 관리하기 어려우며, 추후 문제가 생길 여지는 존재

- 서브 클래스에서의 메서드 오버라이딩을 강제

▶ 보다 엄격하게 상속 클래스들의 메서드들을 관리할 수 있음

▶ raise NotImplementedError() : 엄격하게 상속 클래스의 메서드 관리가 가능

▶ 그러나 미구현 메서드가 호출되기 전에 미리 파악할 수 없음

**추상 클래스**

- abc 모듈의 ABC 클래스를 상속 받음 (abstract base class)

추상 메서드

- abc 모듈에 정의되어 있는 abstractmethod 함수를 decorator 방식으로 적용

추상 메서드가 존재하는 추상 클래스는 객체를 생성할 수 없음

추상 클래스가 아닌 클래스에서의 추상 메서드는 가능

- 추상 메서드가 없는 추상 클래스, 추상 클래스가 아닌 클래스에서의 추상 메서드

▶ 객체 생성과 메서드 사용에 제약이 없으며, 다른 일반적인 경우와 동일하게 활용 가능

**추상 클래스의 상속**

추상 클래스를 상속받은 클래스도 추상 클래스

- 추상 메서드를 일반 메서드로 오버라이드하지 않으면, 여전히 추상 메서드를 포함
- 추상 메서드를 일반 메서드로 오버라이드함으로써, 객체 생성이 가능해짐

▶ 슈퍼 클래스를 정의해야 할 필요가 있을 경우, 코드가 길어지더라도 잊지 않고 관리할 수 있도록 하기 위해서 추상 메서드를 사용한다 -> Overring을 강제

**추상 클래스를 이용한 인터페이스 제공**

- 슈퍼 클래스에서는 메서드를 구체적으로 구현하지 않아도 됨
- 서브 클래스에서는 슈퍼 클래스에서의 미구현 메서드를 오버라이드하여 다형성을 제공
- 슈퍼 클래스를 추상 메서드를 가진 추상 클래스로 정의

▶ 서브 클래스 역시 추상 클래스이므로, 추상 메서드를 더 추가할 수도 있음

▶ 서브 클래스의 객체를 생성하기 위해서는 추상 메서드를 반드시 일반 메서드로 오버라이드

▶ 보다 엄격하게 코드의 유지 보수 및 관리를 할 수 있게 됨

**모듈**
- 관련성이 깊은 함수나 변수 또는 클래스들을 모아 놓은 파일
- .py 확장자를 갖는 일반적인 하나의 파이썬 파일

여러 모듈로 구성된 프로그램
- 여러 모듈로 프로그램을 구성하는 것이 강제되는 것은 아님
- 복잡한 프로그램의 경우, 하나의 모듈에 모든 코드를 작성하기보다는 여러 모듈에 나누어 작성하는 것이 관리에 이점이 있음

▶ 코드가 크고, 방대하고 복잡한 경우 관련성 있는 것들끼리 모아야 쉽고 빠르게 탐색 가능

- 특정 모듈에 작성되어 있는 클래스, 함수, 변수 등을 다른 모듈에서 불러와 사용 가능

**다른 파이썬 파일에 있는 클래스를 사용하는 방법**
- 모듈 자체를 불러오기

▶ import 모듈명

▶ 모듈 내에 있는 클래스와 함수, 변수에 접근하기 위해 모듈명을 접두어로 사용해야함

▶ ex) 모듈명.클래스명, 모듈명.함수명, 모듈명.변수명

- 모듈 내의 특정 식별자들을 불러오기

▶ from 모듈명 import 식별자1, 식별자2, ....

▶ 모듈명을 접두어로 사용하지 않고 식별자를 그대로 사용 가능

- 모듈 내의 모든 식별자들을 불러오기

▶ from 모듈명 import *

▶ 모듈명을 접두어로 사용하지 않고 식별자를 그대로 사용 가능

**import 문법 정리**
import 모듈명.식별자 불가
dogs.py 모듈 내에 선언된 Dog 클래스를 한번에 import 하는 것은 불가

▶ from 없이 import만 사용된 경우에는 import 뒤에는 모듈명

▶ 모듈 내에 선언된 개별 식별자에 접근하기 위해서는 from 모듈명이 필요

**모듈 import 시 주의할 점**
- 식별자 중복

▶ 다른 모듈로부터 Dog라는 식별자를 갖고 있는 클래스를 import하였으나, 새롭게 정의한 클래스의 이름으로 Dog를 사용한 경우, Dog라는 식별자는 새로 정의한 클래스

▶ 클래스 뿐만 아니라 변수, 함수에 대해서도 동일

▶ 모듈 import에서만 나타나는 문제는 아님

▶ 한 모듈 내에서 동일한 식별자를 여러 번 사용한 경우에도 마지막에 실행된 문장만이 효력

**여러 모듈 간 import**
import 된 함수, 변수, 클래스 등은 import를 수행한 모듈에서 새로 선언한 것과 동일하게 동작

**여러 모듈로 구성된 프로그램의 실행**

프로그램의 실행 과정

- 특정 모듈을 실행하면, 해당 모듈 내에 들여쓰기 되지 않은 문장들이 순차적으로 실행
- 특정 모듈을 import하면, 해당 모듈에 있는 모든 들여쓰기 되지 않은 문장 모두 실행

__name__ = "__main__"의 활용

- 특정 모듈 파일을 직접 실행한 경우에만 만족하는 조건문
- __name__은 파이썬에서 내부적으로 사용되는 변수명
- 직접 실행한 경우 "__main__"이 할당
- import를 통해 간접적으로 실행된 경우에는, __name__ 변수에 모듈의 이름이 할당

임의의 경로에 존재하는 모듈을 import 하는 방법

- sys.py를 import한 뒤에 sys.path에 새로운 경로를 append한 뒤에 import 가능

**패키지**

dot(.)을 이용하여 파이썬 모듈을 계층적으로 관리할 수 있게 해주는 도구
파이썬 모듈들이 들어있는 디렉토리 (.py 파이썬 파일들을 담고 있는 폴더를 의미)
많은 파일들을 관리할 때에 관련성이 큰 것끼리 서브 디렉토리에 모으는 것처럼
파이썬 모듈들도 계층화하여 관리

**패키지 내부로의 접근**

패키지명 뒤에 dot(.)을 사용하여 접근
특정 패키지를 import할 경우 __init__ 모듈이 자동으로 실행
패키지를 모듈처럼 불러온 경우, 하위모듈을 사용 불가
최상단 계층의 패키지, 모듈을 import 할 경우, dot(.) 필요 없음
패키지 내부의 모듈은 패키지명.모듈명으로 접근 가능
from 없이 import만 쓴 경우에는 import 뒤에 모듈명 존재
▶ 패키지명이 올 경우, __init__.pt 실행하고 모듈로 취급
▶ ex) import 패키지명.패키지명.모듈명
from을 사용한 경우에는 import 뒤에 dot(.) 사용 불가
▶ from 패키지명.패키지명.모듈명 import 식별자

**__init__.py**

일반 디렉토리가 아닌 파이썬 패키지임을 나타냄
python 3.3부터 __init__.py이 없어도 패키지로 인식하나, 호환을 위해 넣어주는 것이 안전
패키지를 import 했을 시에 자동으로 __init__.py 모듈이 실행
__init__.py 내부에서 모듈들을 import
패키지를 import함으로써 패키지 내 모듈을 활용 가능해짐

**__init__.py를 이용한 패키지 내 모듈 import**

__init__.py 내부에 __all__ 변수를 이용하여 모듈의 이름 문자열을 아이템으로 갖는 리스트 할당 -> import 문장 실행 시 __all__에 명시된 모듈들이 import 됨

# 코드 버전 관리 및 협업을 위한 Git / Github
# <Git Github 소개>

**버전관리**

- 내가 원하는 시점으로 이동할 수 있게 하는 것
- 팀 프로젝트 진행 시, 어느 파일이 최종 업데이트 파일인지 확인하기 어려움 -> 버전관리

버전 관리 시스템

- 버전 관리를 도와주는 Tool

Git

- 소스코드의 버전 사이를 오가는 기능을 제공하는 버전 관리 시스템
- 버전 사이를 자유롭게 이동하며 새로운 소스코드를 추가, 삭제, 변경

Git 원격 저장소 (호스팅 사이트)

- Git으로 관리하는 프로젝트를 올려둘 수 있는 저장소 & 사이트 ex) GitHub, GitLab

# 코드 버전 관리 및 협업을 위한 Git / Github
# <CLI 활용 기초>

로컬저장소
- 실제로 Git을 통해 버전 관리가 이뤄질 내 컴퓨터에 있는 폴더
원격저장소
- 다른 사람과 공유되는 저장소

Git bash
- Command Line Interface (CLI) : 명령어를 하나씩 입력하는 방식

**로컬 저장소 생성**
- git init을 통해 Git 초기화 -> 명령어 실행 후 git이라는 폴더 자동 생성
▶ [.git] 폴더에는 Git으로 생성한 버전들의 정보와 원격저장소 주소 등이 저장
▶ [.git] 폴더를 로컬 저장소라 지칭

**첫 번째 커밋 만들기**
커밋: 생성된 각각의 버전
- git config –global user.email “ ”, git config –global user.name “ ”
▶ 해당 명령어를 통해 Email, Username 등록
- git add README.txt를 통하여 커밋
- git commit -m “ ”을 통해 커밋에 상세설명 추가 // [-m]은 message의 약자
▶ 상세설명을 통하여 파일을 수정한 이유, 파일의 생성이유를 알 수 있음
▶ 해당 버전을 찾아서 그 버전으로 코드를 교체하기도 수월

**다른 커밋으로 버전 이동**
개발을 하다가 요구사항이 바뀌면, Git을 이용하여 이전 커밋으로 되돌아감
- git log 명령어를 통해 현재까지 만든 커밋 확인 가능
- git checkout [커밋 아이디]를 통해 해당 커밋으로 코드를 돌릴 수 있음
▶ 커밋 아이디의 경우 앞 7자리만 입력해도 가능
▶ git checkout - 의 경우 최신버전의 커밋으로 이동 // ‘-’은 최신버전을 의미

**Github 원격저장소에 커밋 올리기**
.gitignore: 작업할 때 굳이 GitHub 원격저장소에 올릴 필요가 없는 파일을 업로드 X
- git remote add origin “ ~ ” 을 통해 로컬저장소에 원격저장소 주소를 입력
- 로컬 저장소에 있는 커밋들을 git push 명령어를 통해 원격저장소에 올림
▶ git push origin mater
- git clone “원격저장소주소”를 통해 내 컴퓨터의 로컬저장소로 내려받음
▶ 맨 뒤에 한 칸 띄고, 마침표를 입력하면 현재 폴더에 클론을 받음
- git pull origin master를 통해 원격저장소의 새로운 커밋을 로컬저장소로 내려 받음

**정리**

Git : 버전 관리 시스템 중 하나
Github : Git으로 관리하는 프로젝트를 올려둘 수 있는 사이트이자 원격저장소
CLI : Command Line Interface의 약어, 명령어를 하나씩 입력하는 방식의 인터페이스
Git Bash : CLI 방식으로 Git을 사용할 수 있는 환경을 의미
Commit (커밋) : 버전 관리를 통해 생성된 파일, 혹은 그 행위를 의미
Git log (로그 명령어) : 지금까지 만든 커밋을 모두 확인하는 명령어
체크아웃 : 원하는 지점으로 파일을 되돌리는 기능
로컬저장소 : Git으로 버전 관리하는 내 컴퓨터 안의 폴더
원격저장소 : Github에서 협업하는 공간 (폴더)
레포지토리 : 원격저장소를 의미
푸시 : 로컬저장소의 커밋을 원격저장소에 올리는 것
풀 : 원격저장소의 커밋을 로컬저장소에 내려받는 것

# 코드 버전 관리 및 협업을 위한 Git / Github
# <GUI 활용 소스트리>

**소스트리**

- Git 사용을 도와주는 Graphic User Interface (GUI) 프로그램
- 커맨드 라인을 입력, 명령을 실행하는 CLI 방식과 달리, 버튼 클릭을 통해 필요한 명령을 실행하기 때문에 편리
- Git의 핵심인 커밋, 푸시, 브랜치 등을 눈으로 쉽게 확인할 수 있어서 개념을 이해하는데 도움

**소스트리 화면 구성**

Local: 로컬저장소 목록
- 컴퓨터에 저장되어 있는 Git 로컬저장소 목록을 보여줌
- Git으로 관리되는 모든 저장소가 자동으로 표시되는 것은 아님
- Add를 통해 소스트리에 추가

Remote: 원격저장소 목록
- 원격저장소 계정에 연결되어 있는 모든 원격저장소를 볼 수 있음
- 예를 들어 Github 계정을 소스트리에 추가하면, Github에서 등록한 원격저장소 목록 확인

Clone: 원격저장소 클론
- 원격 서버에 올라와 있는 Git 저장소를 내 로컬 컴퓨터에 다운로드 및 연동

Add: 로컬저장소 추가
- 내 컴퓨터에 만들어져 있는 로컬저장소를 소스트리에서 관리할 수 있도록 추가

Create: 로컬저장소 생성
- 내 컴퓨터에 있는 일반 폴더를 Git으로 버전 관리할 수 있도록 로컬저장소를 생성
- 모든 폴더는 자동으로 Git으로 버전 관리되지 않고, 해당 폴더 내에 로컬저장소를 생성해야 버전 관리가 가능

+: 새탭
- 새로운 탭을 열어서 다른 저장소 관리 가능

**로컬저장소를 소스트리에 불러오기**

[Clone], [Add], [Create] 차이
- [Clone]: 원격저장소를 내 컴퓨터에 받아오고 (로컬저장소 자동 생성), 소스트리에도 추가
- [Add]: 내 컴퓨터에서 이미 만든 로컬저장소를 소스트리에 추가
- [Create]: 내 컴퓨터의 폴더에 새로운 로컬저장소 생성하기 (git init)

Git 기초 원리의 이해

커밋은 차이점이 아니라 스냅사진
- SVN과 같은 시스템과 Git의 가장 큰 차이는, Git이 커밋에 바뀐 것만 저장X

▶ 전체 코드를 저장

# 코드 버전 관리 및 협업을 위한 Git / Github
# <GUI 활용 브랜치 병합>

**원격저장소에서 협업**

- Git이 커밋을 관리하는 방식: 줄줄이 기차

▶ 커밋: 코드의 변경사항을 묶어 하나의 덩어리 (버전)으로 생성
▶ 새로 만든 커밋은 기존 커밋 다음에 시간 순으로 쌓이게 됨
▶ 한 명이 작업한다면, 한 줄로 계속 커밋으 쌓아가면 됨
▶ 두 명이 협업을 한다면, 특정 기준에서 줄기를 나누어 작업할 수 있는 기능 브랜치 필요
▶ 만약 브랜치를 만들지 않고, 특정 기준으로 커밋을 만들 경우 에러 발생

[master]는 Git에서 제공하는 기본 브랜치를 의미
첫 번째 커밋을 하면 자동으로 'master'라는 이름의 브랜치가 커밋
브랜치는 단순한 포인터

▶ 새로 커밋할 때마다 [master] 브랜치의 포인터가 최신 커밋을 가리킴
▶ 브랜치를 생성하면, 특정 기준점을 모두 가리키고, 해당 브랜치에 커밋 시 이동
▶ 만약 브랜치 사이를 넘나들고 싶다면 [HEAD]라는 특수한 포인터 활용
▶ [HEAD]는 브랜치 혹은 커밋을 가리키는 포인터
▶ 브랜치의 최신 커밋이 아닌 과거 커밋으로도 [HEAD]를 이동할 수 잇음
▶ 해당 경우에는 [master]와 브랜치 포인터와 [HEAD] 브랜치 포인터가 떨어짐
▶ 분리된 HEAD 상태

**브랜치 실습 기본**

1. 동시에 작업하다가 꼬일 수 있으므로 [master] 브랜치에는 직접 커밋을 올리지 않는다.
▶ 해당 경우에 master 브랜치는 최종버전의 브랜치라고 간주, 개발 후 최종 검증까지 완료
2. 기능 개발을 하기 전에 [master] 브랜치를 기준으로 새로운 브랜치를 만든다.
▶ 확정된 브랜치 상태인 master 브랜치에서 기준을 생성하고 브랜치 생성 후 작업
3. 브랜치의 이름은 [feature/기능이름] 형식으로 하고 한 명만 커밋을 올린다.
▶ 한 사람 당 하나의 branch를 관리하고, 한 명만 commit 하도록
4. [feature/기능이름] 브랜치에서 기능 개발이 끝나면 [master] 브랜치에서 이를 합친다.
▶ 이와 같은 방식을 통해 결국 작업한 모든 코드가 [master] 브랜치에서 합쳐짐
▶ 직접 [master] 브랜치에는 커밋을 진행하지 않음

**병합 (Merge)**

두 버전의 합집합을 구해서 버전을 구성

- 병합 커밋 (Merge Commit)

▶ 베이스로부터 변경된 부분이 겹치지 않으면 두 커밋을 합쳐 저장

- 빨리 감기 병합 (Fast-forward)

▶ 합친 결과물이 합치기 전 두 커밋 중 하나와 같을 경우, 새로 상태를 만들 필요 없이 이전 상태로 바꿔주면 됨

- 충돌 상태 (Conflict)

▶ 베이스로부터 변경된 부분에 겹친 부분이 있을 경우

▶ 무엇을 남길지 수동으로 선택

**브랜치를 합치는 예의바른 방법: 풀 리퀘스트**

풀 리퀘스트

- 협업자에게 브랜치 병합을 요청하는 메시지를 보내는 것