

생성형AI

Day 3

Python Programming I



목차

1. 파이썬 기본 문법 복습
2. 자료형
3. 함수
4. 클래스와 객체지향 프로그래밍
5. 제너레이터와 이터레이터
6. 파일 입출력
7. Database
8. 모듈과 패키지
9. 예외 처리



파이썬 기본 문법 복습 - 변수와 자료형

변수

- 데이터를 저장하는 메모리 공간을 명명한 것

자료형

: 변수가 갖는 데이터의 종류

- 숫자형: 정수(int), 실수(float)
- 문자열: 문자(str)
- 불린형: bool(True, False)

자료형 변환

- 정수형으로 변환: int()
- 실수형으로 변환: float()
- 문자열로 변환: str()

파이썬 기본 문법 복습 - 기본 연산 및 조건문

기본연산

- 산술연산: +, -, *, /, //, %
- 비교연산: ==, !=, >, <, >=, <=
- 논리연산: and, or, not

조건문

- 특정조건에 따라 코드블록을 실행하는 구조로 프로그램의 흐름을 제어
- if -> 조건이 참인 경우에만 실행
- if - else -> 조건이 참인 경우와 거짓인 경우 각각 다른 코드블록을 실행
- if - elif - else -> 여러 조건을 검사하여 각각의 조건에 맞는 코드블록을 실행

파이썬 기본 문법 복습 - 반복문

반복문

- 코드블록을 조건 혹은 시퀀스에 따라 여러 번 실행하는 구조
- for문 -> 시퀀스 자료형 요소를 순회
- for문 + range() -> range로 지정한 범위 내에서 반복
- while문 -> 조건이 참인 동안 반복
- break -> 반복문 종료
- continue -> 현재 반복을 종료하고 다음 반복으로 이동

함수

함수

- 재사용 가능한 코드블록으로 코드의 가독성과 유지보수성을 높임
- 코드의 중복을 줄이고, 모듈화된 코드를 작성
- 복잡한 시스템을 구성하는 기본 단위

‘def’ 키워드를 사용해서 정의

```
def 함수이름(매개변수) :
```

 코드 블록

 return 반환값

함수호출

- 함수 이름과 괄호를 사용해서 호출

```
result = greet("Bob")
```

함수

람다함수

- 한 줄로 작성 가능한 익명 함수
- 간단한 연산이나 데이터 변환에 유용

‘lambda’ 키워드를 사용하여 정의.

lambda 매개변수: 반환값

```
square = lambda x: x ** 2
```

왜 필요할까요?

- 코드 간결화
- 일회성 함수가 필요한 경우

함수

*args

- 가변인자
- 여러 개의 인자를 함수에 전달할 때 사용
- 튜플 형태로 전달

```
def 함수이름(*args):  
    코드 블록
```

**kwargs

- 키워드 가변 인자
- 키워드 인자를 함수에 전달할 때 사용
- 딕셔너리 형태로 전달

```
def 함수이름(**kwargs):  
    코드 블록
```

왜 매개변수를 명시하지 않고 가변인자를 사용할까요?

- 유연한 함수 설계 가능
- 다양한 함수 호출 패턴을 지원

함수

함수형 프로그래밍

- 함수를 일급 객체로 사용하여 코드를 구성, 상태 변화와 부작용을 최소화하는 프로그래밍 방법
- map, filter, reduce와 같은 고차 함수 사용

map 함수

- 모든 요소에 함수를 적용하여 새로운 리스트 반환

filter 함수

- 조건에 맞는 요소만 걸러내어 새로운 리스트 반환

reduce 함수

- 모든 요소를 누적하여 단일 값을 반환
- functools 모듈에서 제공

왜?

- 코드를 더 간결하고 읽기 쉽게 만들
- 대규모 데이터 처리에 유용하며, 실무에서 데이터 파이프라인 구축 시 자주 사용

클래스와 객체지향 프로그래밍

객체지향 프로그래밍

객체(Object)를 중심으로 프로그램을 구성하는 프로그래밍 패러다임

추상화 (Abstraction):

- 중요한 정보만을 표현하고 불필요한 세부 사항을 숨기는 것.

캡슐화 (Encapsulation):

- 데이터와 메서드를 하나의 단위로 묶고 외부로부터 숨기는 것.

상속 (Inheritance):

- 기존 클래스(부모 클래스)를 기반으로 새로운 클래스(자식 클래스)를 정의하는 것.

다형성 (Polymorphism):

- 동일한 인터페이스를 사용하여 서로 다른 데이터 타입의 객체를 다루는 것.

객체지향 프로그래밍의 장점

- 코드 재사용성: 기존 코드를 재사용하여 새로운 프로그램을 쉽게 개발.
- 코드 유지보수성: 모듈화된 코드 구조로 인해 유지보수가 용이.
- 확장성: 새로운 기능 추가 시 기존 코드를 수정하지 않고 확장 가능.

클래스와 객체지향 프로그래밍

클래스

객체를 정의하는 데 사용되는 청사진

속성(attributes)과 메서드(methods)로 구성

객체지향 프로그래밍(OOP)의 기본 단위로, 코드의 재사용성, 확장성, 유지보수성을 높임

‘class’ 키워드를 사용하여 정의

인스턴스

- 클래스는 객체를 생성하기 위한 템플릿
- 인스턴스는 클래스의 실제 예제
- 클래스를 사용하여 여러 객체를 효율적으로 생성하고 관리할 수 있음

class 클래스이름:

```
def __init__(self, 매개변수):  
    self.속성 = 매개변수
```

```
def 메서드이름(self, 매개변수):  
    코드 블록
```

클래스와 객체지향 프로그래밍

상속 (Inheritance)

- 기존 클래스(부모 클래스)를 기반으로 새로운 클래스(자식 클래스)를 정의하는 것
- 코드의 재사용성을 높이고, 계층적 관계를 명확히 함

```
class 부모클래스: # 부모 클래스의 속성과 메서드 정의
```

```
class 자식클래스(부모클래스): # 자식 클래스의 속성과 메서드 정의
```

다형성 (Polymorphism)

- 동일한 인터페이스를 사용하여 서로 다른 데이터 타입의 객체를 다룰 수 있는 능력
- 코드의 유연성과 확장성을 높임

클래스와 객체지향 프로그래밍

매직 메서드 (Magic Methods)

- 특별한 이름을 가진 메서드로, 파이썬이 내부적으로 사용하는 메서드
- 객체의 특정 동작을 사용자 정의할 수 있음
- 대표적인 매직 메서드:
 - `__init__`: 객체 초기화 메서드
 - `__str__`: 객체의 문자열 표현을 반환
 - `__repr__`: 객체의 공식 문자열 표현을 반환

연산자 오버로딩 (Operator Overloading)

- 파이썬의 기본 연산자를 사용자 정의 클래스에서 사용할 수 있도록 메서드를 정의
- 객체 간의 연산을 직관적으로 표현 가능

대표적인 매직 메서드:

- `__add__`: + 연산자
- `__sub__`: - 연산자
- `__mul__`: * 연산자
- `__truediv__`: / 연산자

제너레이터와 이터레이터

이터레이터 (Iterator)

- `__iter__()`와 `__next__()` 메서드를 구현한 객체
- 반복자: 반복 가능한 객체에서 값을 순차적으로 꺼내는 역할
- 상태 유지: 현재 위치를 기억하여 `next()` 호출 시 다음 값을 반환

제너레이터 (generator)

- 일련의 값을 생성하는 이터레이터의 일종으로 함수처럼 정의되지만, 값을 반환할 때 `return` 대신 `yield` 키워드를 사용
- 지연 평가 (Lazy Evaluation): 필요한 시점에 값을 생성.
- 상태 유지: 마지막 실행 지점에서 멈추고 상태를 기억하여 다음 호출 시 그 지점부터 재개.
- 메모리 효율성: 한 번에 하나의 값만 생성하므로 메모리 사용을 최소화.
- 이터레이터의 특별한 형태: 제너레이터도 이터레이터이므로 `__iter__()`와 `__next__()`를 자동으로 구현.
- 코드 간결화: 복잡한 이터레이터 코드를 간단히 작성 가능

제너레이터와 이터레이터

구현 방식:

- 이터레이터: 클래스 형태로 `__iter__()`와 `__next__()` 메서드를 구현
- 제너레이터: 함수 형태로 `yield` 키워드를 사용

사용 용이성:

- 이터레이터: 상대적으로 복잡한 구조
- 제너레이터: 간단한 코드로 구현 가능

메모리 효율성:

- 이터레이터: 모든 값을 메모리에 저장
- 제너레이터: 값을 필요할 때마다 생성

제너레이터와 이터레이터

yield 키워드

- 제너레이터 함수에서 값을 반환하고 함수의 실행 상태를 일시 중지
- 자연 평가: yield 키워드를 만나면 값을 반환하고 함수의 실행 상태를 저장
- 상태 유지: 다음 호출 시 저장된 상태에서 다시 시작

필요성

- 메모리 절약: 한 번에 하나의 값만 생성하여 메모리 사용을 최소화
- 복잡한 흐름 제어: 제너레이터를 사용하여 복잡한 반복 작업을 간단히 구현

활용

- 대량 데이터 처리: 대규모 파일 읽기, 대규모 데이터베이스 쿼리 결과 처리 등
- 스트리밍 데이터 처리: 실시간 데이터 스트리밍에서 유용

파일 입출력

파일 입출력의 개념

- 파일 시스템을 통해 데이터를 읽고 쓰는 작업.
- 데이터 영속성 유지, 외부 데이터 소스와의 연동, 대용량 데이터 처리 등에 필수적

with 문

- 자원 관리가 필요한 코드 블록에서 사용되는 구문
- 코드의 가독성과 안전성을 높이며, 자원 해제를 자동으로 처리하여 코드의 안정성을 보장
- 파일을 열고 닫는 작업을 자동으로 처리
- 예외 발생 시에도 파일이 제대로 닫히도록 보장

Open 함수

- 파일을 열고 파일 객체를 반환하는 함수

파일 입출력

모드

- 'r': 읽기 모드 (기본값)
- 'w': 쓰기 모드 (파일이 존재하면 덮어씀)
- 'a': 추가 모드 (파일 끝에 데이터 추가)
- 'b': 이진 모드 (예: 'rb', 'wb')

읽기 함수

- `read()`: 파일 전체를 읽어들임
- `readline()`: 한 줄씩 읽어들임
- `readlines()`: 파일 전체를 읽어 각 줄을 요소로 하는 리스트 반환

쓰기 함수

- `write()`: 문자열을 파일에 씀
- `writelines()`: 문자열 리스트를 파일에 씀

파일 입출력

CSV 파일 처리

- Comma-Separated Values, 쉼표로 구분된 값들을 가지는 텍스트 파일 형식
- 데이터 교환의 표준 형식, 다양한 애플리케이션 간의 호환성 때문에 널리 사용

CSV 모듈

- csv.reader: CSV 파일을 읽어들임
- csv.writer: CSV 파일에 씀
- DictReader: 딕셔너리 형태로 CSV 파일을 읽어들임
- DictWriter: 딕셔너리 형태로 CSV 파일에 씀

Pandas 모듈

파일 입출력

JSON 데이터 처리

- JavaScript Object Notation, 가벼운 데이터 교환 형식
- 사람과 기계 모두 읽기 쉬운 형식, 키-값 쌍으로 데이터 구조화
- 데이터 교환: 웹 애플리케이션과 서버 간의 데이터 교환에서 자주 사용
- 구조화된 데이터 저장: 복잡한 데이터 구조를 손쉽게 저장하고 불러올 수 있음

JSON 모듈

- json.dumps(): 파이썬 객체를 JSON 문자열로 변환
- json.loads(): JSON 문자열을 파이썬 객체로 변환
- json.dump(): 파이썬 객체를 JSON 파일로 저장
- json.load(): JSON 파일을 파이썬 객체로 불러옴

Database

데이터베이스

- 구조화된 데이터의 집합으로, 데이터를 효율적으로 저장, 관리, 검색할 수 있도록 설계된 시스템
- 관계형 데이터베이스 (RDBMS): 데이터를 테이블 형식으로 저장. SQL 사용 (예: MySQL, PostgreSQL)
- 비관계형 데이터베이스 (NoSQL): 다양한 데이터 모델을 지원 (예: MongoDB, Cassandra)

데이터베이스의 구성 요소

- 테이블 (Table): 데이터가 저장되는 기본 단위
- 레코드 (Record): 테이블의 한 행(row)
- 필드 (Field): 테이블의 한 열(column)
- 키 (Key): 데이터를 고유하게 식별할 수 있는 필드

Database

관계형 데이터베이스의 특징

- 스키마 (Schema): 데이터베이스 구조와 제약 조건 정의
- SQL (Structured Query Language): 데이터 정의, 조작, 제어를 위한 언어
- 참조 무결성 (Referential Integrity): 데이터의 일관성을 유지하기 위한 제약 조건

이유와 필요성

- 데이터 관리: 대량의 데이터를 효율적으로 저장, 관리, 검색
- 데이터 무결성: 데이터의 정확성과 일관성을 유지
- 데이터 보안: 접근 제어를 통해 데이터 보호

Database

파이썬과 데이터베이스 연동

- 파이썬 애플리케이션에서 데이터베이스에 접속하고, 데이터를 조회, 삽입, 수정, 삭제하는 작업
- 데이터를 효율적으로 처리하고, 데이터 기반 애플리케이션 개발

파이썬 DB-API

- 파이썬에서 데이터베이스와 상호 작용하기 위한 표준 인터페이스
- Connection 객체: 데이터베이스와의 연결을 관리
- Cursor 객체: SQL 문을 실행하고 결과를 관리

주요 데이터베이스 모듈

- SQLite: 내장형 데이터베이스
- MySQL: 오픈 소스 관계형 데이터베이스
- PostgreSQL: 오픈 소스 객체-관계형 데이터베이스

Database

ORM(Object-Relational Mapping)

- 객체 지향 프로그래밍 언어를 사용하여 관계형 데이터베이스를 조작하는 방법
- SQL 문을 직접 작성하지 않고, 객체 지향 방식으로 데이터베이스 작업을 수행할 수 있음
- SQLAlchemy: 파이썬의 대표적인 ORM 라이브러리

ORM의 장점

- 코드 간결화: SQL 문을 직접 작성하지 않고, 파이썬 코드로 데이터베이스 작업을 수행
- 유지보수성 향상: 데이터베이스 스키마 변경 시 코드 수정 최소화
- 객체 지향 설계: 객체 지향 방식으로 데이터베이스 작업을 설계 및 구현
- 웹 애플리케이션 개발: Django, Flask 등 웹 프레임워크에서 ORM을 사용하여 데이터베이스 작업을 효율적으로 수행

모듈과 패키지

모듈

- 파이썬 코드의 논리적인 단위로, 관련된 함수, 클래스, 변수 등을 하나의 파일에 모아놓은 것
- 코드의 재사용성과 유지보수성을 높이며, 코드의 구조를 더 체계적으로 관리
- 하나의 .py 파일이 하나의 모듈

모듈 가져오기

- import 키워드를 사용하여 모듈을 불러옴

모듈에서 특정 항목 가져오기

- from 키워드를 사용하여 특정 함수, 클래스 등을 가져옴

표준 라이브러리와 사용자 정의 모듈

- 표준 라이브러리: 파이썬 설치 시 기본으로 제공되는 모듈 모음 (예: math, datetime, os)
- 사용자 정의 모듈: 사용자가 직접 작성한 모듈

모듈과 패키지

패키지

- 관련된 모듈을 하나의 디렉터리로 묶은 것
- `__init__.py` 파일: 패키지를 나타내는 파일
- pip: 파이썬 패키지 관리 도구
- `pip install 패키지이름`
- `pip uninstall 패키지이름`
- `pip list`

패키지 관리

- `requirements.txt` 파일: 프로젝트에 필요한 패키지 목록을 기록한 파일
- `pip freeze > requirements.txt.` // pip 설치 패키지를 requirements.txt에 저장
- `pip install -r requirements.txt` // requirements.txt에 있는 패키지를 설치

모듈과 패키지

가상환경

- 프로젝트별로 독립적인 파이썬 환경을 제공하여, 패키지 충돌을 방지하고 일관된 개발 환경을 유지
- 서로 다른 프로젝트 간의 패키지 의존성 문제를 해결하고, 테스트 환경을 쉽게 관리

venv 모듈 사용

- python -m venv 가상환경이름
- 가상환경이름\Scripts\activate
- source 가상환경이름/bin/activate

venv 모듈 종료

- deactivate

왜 사용하나요?

- 프로젝트별 가상환경 설정: 각 프로젝트마다 독립적인 가상환경을 설정하여, 패키지 충돌 방지
- 테스트 환경: 개발 환경과 별도의 테스트 환경을 설정하여 안정적인 코드 배포

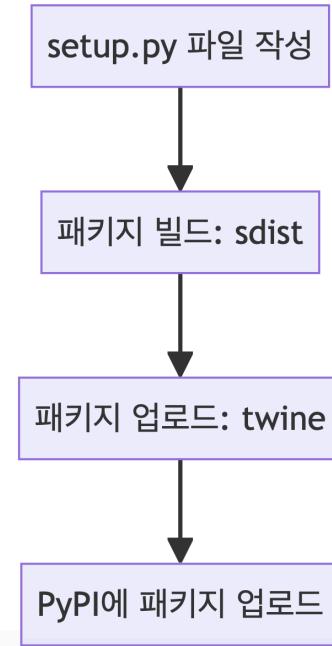
모듈과 패키지

배포

- 작성한 모듈이나 패키지를 다른 개발자와 공유하기 위해 배포하는 과정
- 코드 재사용성 향상, 커뮤니티 기여, 프로젝트 관리
- 조직 내에서 공통으로 사용하는 코드나 라이브러리를 패키지로 배포
- 오픈소스 커뮤니티에 유용한 패키지를 배포하여 기여

배포 방법

- PyPI (Python Package Index): 파일형 패키지를 공유하는 공식 저장소
- 패키지 빌드: `sdist`
- 패키지 업로드: `twine`



예외처리

예외처리의 개념

- 프로그램 실행 중 발생하는 오류를 관리하고 처리하는 기법
- 예외처리를 통해 프로그램의 비정상 종료를 방지하고, 사용자에게 유용한 오류 메시지를 제공하여 프로그램의 안정성과 신뢰성을 높임

예외와 오류의 차이

- 오류 (Error): 프로그램 실행을 중단시키는 심각한 문제
- 예외 (Exception): 프로그램 실행 중 발생할 수 있는 처리 가능한 오류

예외의 종류

- 내장 예외 (Built-in Exceptions): 파이썬에서 미리 정의한 예외 (예: ZeroDivisionError, ValueError)
- 사용자 정의 예외 (User-defined Exceptions): 사용자가 직접 정의한 예외

예외처리

try와 except

- 예외가 발생할 수 있는 코드를 try 블록에 작성하고, 예외 발생 시 except 블록에서 처리

finally

- 예외 발생 여부와 상관없이 항상 실행되는 블록

raise

- 강제로 예외를 발생시키는 구문

예외처리

사용자 정의 예외

- 내장 예외 외에, 특정 상황에서 발생하는 예외를 처리하기 위해 사용자가 직접 정의한 예외
- 도메인 특화 예외를 정의하여 코드의 가독성과 유지보수성을 높임

```
class 예외이름(Exception):  
    def __init__(self, 메시지):  
        self.메시지 = 메시지
```

예외 처리 이유

- 안정성 확보: 예외처리를 통해 예기치 않은 오류로 인한 프로그램 중단을 방지
- 디버깅 용이성: 구체적인 예외 메시지를 제공하여 디버깅을 쉽게 수행
- 사용자 경험 개선: 사용자에게 유용한 오류 메시지를 제공하여 프로그램 사용성을 높임

실습 과제

1. 파이썬 프로그래밍 종합 실습

- <https://colab.research.google.com/drive/1oILuVpnWoSBIGOOFYoEnYuD6AHNsexX-?usp=sharing>

2. 매일 알고리즘 문제 풀기

- 꾸준한 연습을 통해 문제 해결 능력과 코딩 스킬 향상
- 매일 조금씩이라도 알고리즘 문제를 푸는 습관 들이기
- goormlevel

실습 진행