5단원. 파이썬 기초 문법

> 대상: 전공자 · 정보처리기사 취득자 | 목적: 인프라 직무(네트워크·클라우드·시스템·보안) 기술 면접 대비 중요도: ★★★ 최빈출 · ★★ 자주 출제 · ★ 기본 개념

> **Q1.** 파이썬의 기본 자료형(**list, tuple, dict, set**)의 차이를 설명하세요. ★★★

답안 리스트(**list**)는 순서가 있고 중복을 허용하며 변경 가능(**mutable**)한 자료형으로 대괄호 `[]`로 표현하고, 원소 추가·삭제·변경이 자유로워 가장 범용적으로 사용됩니다. 튜플(**tuple**)은 리스트와 비슷하게 순서가 있지만 한 번 생성되면 변경 불가능**(immutable)**하다는 점이 핵심 차이이며, 이 덕분에 딕셔너리의 키로 사용할 수 있고 함수에서 여러 값을 안전하게 묶어 반환할 때 자주 쓰입니다. 딕셔너리(**dict**)는 키-값 쌍으로 데이터를 저장하는 해시 기반 자료구조로 키를 통해 O(1)에 가까운 평균 시간복잡도로 값에 접근할 수 있으며, 파이썬 3.7 이상에서는 삽입 순서를 유지합니다. 집합(**set**)은 중복을 허용하지 않고 순서가 없는 자료형으로 합집합·교집합·차집합 같은 집합 연산과 중복 제거에 유용합니다. 변경 가능 여부(mutable/immutable)가 어떤 자료형을 딕셔너리 키나 집합 원소로 쓸 수 있는지를 결정한다는 점이 자주 출제됩니다.

관련 개념 mutable/immutable, 해시 가능(hashable), 딕셔너리 키 제약, 집합 연산

> **Q2.** 파이썬의 제어문(**if, for, while**)과 리스트 컴프리헨션의 기본 문법을 설명하세요. ★

답안 파이썬은 중괄호 대신 들여쓰기(**indentation**)로 코드 블록을 구분하는 것이 문법적 특징이며, `if` / `elif` / `else` 로 조건 분기를 작성합니다. `for` 문은 `for item in iterable:` 형태로 반복 가능한(iterable) 객체를 순회하며, C 언어처럼 인덱스 기반 카운터를 직접 다루기보다 `range()` 나 `enumerate()` 를 함께 활용하는 방식이 파이썬다운(Pythonic) 스타일로 권장됩니다. `while` 문은 조건이 참인 동안 반복하며 `break` / `continue` 로 흐름을 제어합니다. 리스트 컴프리헨션은 `[표현식 for 항목 in 반복가능객체 if 조건]` 형태로 반복문과 조건문을 한 줄로 압축해 새로운 리스트를 생성하는 문법으로, 가독성과 실행 속도 모두에서 일반 for 루프보다 유리한 경우가 많습니다.

관련 개념 들여쓰기 기반 블록, enumerate/range, 리스트 컴프리헨션, Pythonic 스타일

> **Q3.** 파이썬 함수의 가변인자(***args, ****kwargs**)를 설명하세요. ★★

답안 파이썬 함수는 고정된 개수의 매개변수 외에 `*args` 를 사용해 임의 개수의 위치 인자를 튜플 형태로 받을 수 있고, `kwargs` 를 사용해 임의 개수의 키워드 인자를 딕셔너리 형태로 받을 수 있습니다. 예를 들어 `def func(a, b, *args, kwargs):` 로 정의하면 `a` , `b` 는 필수 인자로, 그 이후에 넘어오는 위치 인자들은 `args` 튜플에, `key=value` 형태로 넘어오는 인자들은 `kwargs` 딕셔너리에 모입니다. 이 문법은 인자 개수가 가변적인 유틸리티 함수를 만들거나, 다른 함수에 인자를 그대로 전달(위임)하는 래퍼(wrapper) 함수를 작성할 때 유용합니다. 함수 정의 시 매개변수 순서는 일반 위치 인자, `*args` , 기본값이 있는 키워드 전용 인자, `**kwargs` 순서를 지켜야 문법 오류가 나지 않습니다.

관련 개념 위치 인자 vs 키워드 인자, 인자 언패킹(*, **), 래퍼 함수, 기본값 매개변수

> **Q4.** 파이썬의 예외 처리(**try-except-finally**)를 설명하세요. ★★★

답안 파이썬은 오류 발생 가능성이 있는 코드를 `try` 블록에 작성하고, 특정 예외가 발생했을 때 처리할 로직을 `except` 블록에 작성하는 방식으로 예외를 처리합니다. `except`는 `except ValueError:`처럼 특정 예외 타입을 지정해 세분화된 처리를 할 수 있고, 여러 개의 `except` 블록을 순서대로 배치해 서로 다른 예외를 각각 다르게 처리할 수 있으며, 가능한 한 `except:`처럼 모든 예외를 무차별적으로 잡는 방식은 원인 파악을 어렵게 하므로 지양하는 것이 좋습니다. `else` 블록은 `try` 블록에서 예외가 발생하지 않았을 때만 실행되고, `finally` 블록은 예외 발생 여부와 관계없이 항상 실행되므로 파일 닫기, 네트워크 연결 해제 같은 자원 정리 코드를 넣기에 적합합니다. 예외를 발생시키려면 `raise`를 사용하며, 자신만의 예외 클래스를 만들려면 `Exception`을 상속받아 정의할 수 있습니다.

관련 개념 raise, 사용자 정의 예외, else 블록, finally를 통한 자원 정리, with문(context manager)

> **Q5.** 파이썬 가상환경(**venv**)이 왜 필요한지 설명하세요. ★★

답안 파이썬 프로젝트마다 필요한 패키지와 그 버전이 다를 수 있는데, 시스템에 전역으로 패키지를 설치하면 프로젝트 간에 의존성 버전 충돌이 발생하거나 한 프로젝트의 패키지 업그레이드가 다른 프로젝트를 망가뜨릴 수 있습니다. 가상환경 **(virtual environment)**은 프로젝트별로 독립된 파이썬 인터프리터와 패키지 설치 공간을 만들어 이런 충돌을 방지하는 기법으로, 표준 라이브러리인 `venv` 모듈을 이용해 `python -m venv myenv` 명령으로 생성하고 `source myenv/bin/activate`로 활성화해 사용합니다. 활성화된 가상환경 안에서 `pip install`로 패키지를 설치하면 시스템 전역이 아니라 해당 가상환경 디렉토리 안에만 설치되어 격리성이 보장됩니다. 이 외에도 `virtualenv`, `conda`, `pipenv`, `poetry` 등 유사한 목적의 도구들이 있으며, 인프라 자동화 스크립트를 배포할 때도 실행 환경을 재현 가능하게 만들기 위해 가상환경 사용이 권장됩니다.

관련 개념 python -m venv, activate/deactivate, 의존성 충돌, conda/poetry

---

> **Q6. pip과 requirements.txt를 이용한 패키지 관리 방법을 설명하세요.** ★★

답안 **`pip`**은 파이썬의 표준 패키지 관리 도구로, PyPI(Python Package Index)에 등록된 패키지를 `pip install 패키지명`으로 설치하거나 `pip uninstall`로 제거할 수 있습니다. 프로젝트가 의존하는 패키지 목록과 버전을 **`requirements.txt`** 파일에 `패키지명==버전` 형태로 명시해 두면, 다른 개발자나 서버에서 `pip install -r requirements.txt` 한 줄로 동일한 의존성 환경을 재현할 수 있어 "내 컴퓨터에서는 되는데" 문제를 줄이는 데 도움이 됩니다. 현재 설치된 패키지 목록을 파일로 저장하려면 `pip freeze > requirements.txt`를 사용합니다. 버전을 정확히 고정(`==`)하면 재현성은 높아지지만 보안 패치를 놓칠 수 있고, 범위 지정(`>=`)은 유연하지만 예상치 못한 호환성 문제가 생길 수 있어 운영 환경에서는 보통 고정 버전을 사용하는 경우가 많습니다.

관련 개념 pip freeze, PyPI, 버전 고정 vs 범위 지정, 의존성 재현성

> **Q7. 파이썬의 얕은 복사(shallow copy)와 깊은 복사(deep copy)의 차이를 설명하세요.** ★★

답안 얕은 복사는 객체의 최상위 컨테이너만 새로 만들고, 그 안에 담긴 내부 객체(중첩된 리스트, 딕셔너리 등)는 원본과 동일한 참조를 그대로 공유하는 방식으로, `list.copy()`, `list(원본)`, 슬라이싱(`원본[:]`), `copy.copy()` 등으로 수행할 수 있습니다. 이 때문에 중첩된 리스트를 얕은 복사한 뒤 내부 리스트의 원소를 변경하면 원본도 함께 바뀌는 예상치 못한 부작용(side effect)이 발생할 수 있습니다. 깊은 복사는 `copy.deepcopy()`를 사용해 최상위 객체뿐 아니라 내부의 모든 중첩 객체까지 재귀적으로 완전히 새로 복제하므로 원본과 복사본이 완전히 독립적으로 동작합니다. 단순한 1차원 리스트나 불변(immutable) 원소만 담긴 자료구조라면 얕은 복사로도 충분하지만, 리스트 안에 리스트가 있는 것처럼 중첩 구조를 다룰 때는 의도에 맞게 깊은 복사를 선택해야 버그를 예방할 수 있습니다.

관련 개념 copy.deepcopy, 참조 공유 부작용, 중첩 자료구조, 슬라이싱 복사

> **Q8.** 파이썬의 모듈과 패키지 구조를 설명하세요. ★

답안 파이썬에서 모듈(**module**)은 함수, 클래스, 변수 등을 담고 있는 하나의 `.py` 파일을 의미하며, `import 모듈명`으로 다른 파일에서 불러와 재사용할 수 있습니다. 패키지(**package**)는 여러 모듈을 디렉토리 단위로 묶은 것으로, 해당 디렉토리에 `__init__.py` 파일이 있으면(파이썬 3.3 이후로는 없어도 네임스페이스 패키지로 인식되지만 명시적으로 두는 것이 일반적) 패키지로 인식되어 `from 패키지명 import 모듈명`처럼 계층적으로 임포트할 수 있습니다. `__init__.py`는 패키지가 임포트될 때 실행되는 초기화 코드를 담을 수 있고, 패키지의 공개 API를 정의하는 용도로도 활용됩니다. 이런 구조 덕분에 대규모 프로젝트를 기능 단위로 나누어 체계적으로 관리하고, 다른 프로젝트에서도 재사용 가능한 라이브러리 형태로 배포할 수 있습니다.

관련 개념 __init__.py, import/from import, 네임스페이스 패키지, 상대 임포트

> **Q9.** 파이썬의 **GIL(Global Interpreter Lock)**이 무엇인지 설명하세요. ★★

답안 **GIL**은 CPython 인터프리터에서 한 순간에 오직 하나의 스레드만 파이썬 바이트코드를 실행할 수 있도록 강제하는 전역 락으로, 파이썬 객체의 메모리 관리(참조 카운팅)를 여러 스레드가 동시에 건드려 발생할 수 있는 경합 상태를 막기 위해 도입되었습니다. 이 때문에 멀티스레딩을 사용해도 **CPU** 연산이 많은(**CPU-bound**) 작업은 여러 코어를 동시에 활용하지 못해 성능 향상이 제한적이며, 오히려 스레드 간 락 전환 오버헤드로 단일 스레드보다 느려지는 경우도 있습니다. 반대로 파일 입출력이나 네트워크 대기처럼 **I/O** 바운드 작업은 대기 시간 동안 GIL이 해제되므로 멀티스레딩이 여전히 효과적입니다. CPU 바운드 작업에서 진짜 병렬성이 필요하다면 GIL의 영향을 받지 않는 별도 프로세스를 사용하는 **`multiprocessing`** 모듈을 사용하는 것이 일반적인 해결책입니다.

관련 개념 CPU-bound vs I/O-bound, multiprocessing, 참조 카운팅, CPython 한정 특성

> **Q10.** 파이썬에서 == 연산자와 **is** 연산자의 차이를 설명하세요. ★

답안 `==`는 두 객체의 값(내용)이 같은지를 비교하는 연산자로, 리스트나 문자열처럼 서로 다른 객체라도 담고 있는 내용이 동일하면 `True`를 반환합니다. **`is`**는 두 변수가 동일한 객체(메모리 상 동일한 **identity**)를 참조하고 있는지를 비교하는 연산자로, 내용이 같아도 서로 다른 객체라면 `False`를 반환합니다. 예를 들어 `a = [1,2,3]; b = [1,2,3]`일 때 `a == b`는 `True`이지만 `a is b`는 서로 다른 리스트 객체이므로 `False`입니다. 다만 작은 정수나 문자열 리터럴처럼 파이썬이 내부적으로 캐싱하거나 인터닝(interning)하는 값들은 `is` 비교에서도 `True`가 나올 수 있어 혼동을 주기 쉬운데, 이런 최적화에 의존하지 말고 값 비교에는 `==`를, `None` 비교에는 관례적으로 `is None`을 사용하는 것이 올바른 방식입니다.

관련 개념 identity vs equality, 정수/문자열 인터닝, is None 관례, id() 함수
