

# 쉬운 파이썬

파이썬을 배우기 위한 실용 가이드

실행 가능한 예제와 출력 포함

# 목차

제1장: 파이썬 기초

제2장: 제어 흐름

제3장: 데이터 구조

제4장: 함수

제5장: 모듈과 패키지

제6장: 객체지향 프로그래밍

제7장: 오류 처리

제8장: 파일과 데이터 처리

제9장: 고급 파이썬 주제

제10장: 테스트와 모범 사례

제11장: 동시성과 비동기

제12장: 실용적인 응용

# 제1장: 파이썬 기초

파이썬에 오신 것을 환영합니다! 이 장에서는 파이썬 프로그래밍 언어의 기본 구성 요소를 다룹니다.

## Hello World

### 첫 번째 파이썬 프로그램

print() 함수는 화면에 텍스트를 표시합니다. 정보를 출력하는 가장 간단한 방법입니다.

```
print("안녕하세요, 세계!")  
print("쉬운 파이썬에 오신 것을 환영합니다!")
```

출력:

안녕하세요, 세계!

쉬운 파이썬에 오신 것을 환영합니다!

## 변수와 데이터 타입

### 기본 변수

변수는 데이터를 저장합니다. 파이썬은 할당된 값에 따라 자동으로 타입을 결정합니다.

```
# 정수
age = 25
print(f"나이: {age}, 타입: {type(age).__name__}")

# 실수
price = 19.99
print(f"가격: {price}, 타입: {type(price).__name__}")

# 문자열
name = "파이썬"
print(f"이름: {name}, 타입: {type(name).__name__}")

# 불리언
is_active = True
print(f"활성: {is_active}, 타입: {type(is_active).__name__}")
```

출력:

```
나이: 25, 타입: int
가격: 19.99, 타입: float
이름: 파이썬, 타입: str
활성: True, 타입: bool
```

## 타입 변환

쉬운 파이썬  
int(), float(), str(), bool()을 사용하여 타입 간 변환합니다.

```
# 문자열을 정수로
```

```
num_str = "42"
```

```
num_int = int(num_str)
```

```
print(f'{num_str}' -> {num_int} (타입: {type(num_int).__na...
```

```
# 정수를 실수로
```

```
x = 10
```

```
y = float(x)
```

```
print(f'{x}' -> {y})
```

```
# 숫자를 문자열로
```

```
pi = 3.14159
```

```
pi_str = str(pi)
```

```
print(f'문자열 버전: '{pi_str}')
```

출력:

```
'42' -> 42 (타입: int)
```

```
10 -> 10.0
```

```
문자열 버전: '3.14159'
```

# 기본 연산자

## 산술 연산자

파이썬은 모든 표준 산술 연산과 몫 나눗셈, 거듭제곱을 지원합니다.

```
a, b = 17, 5
```

```
print(f"{a} + {b} = {a + b}") # 덧셈
print(f"{a} - {b} = {a - b}") # 뺄셈
print(f"{a} * {b} = {a * b}") # 곱셈
print(f"{a} / {b} = {a / b}") # 나눗셈 (실수)
print(f"{a} // {b} = {a // b}") # 몫 나눗셈
print(f"{a} % {b} = {a % b}") # 나머지
print(f"{a} ** {b} = {a ** b}") # 거듭제곱
```

출력:

```
17 + 5 = 22
17 - 5 = 12
17 * 5 = 85
17 / 5 = 3.4
17 // 5 = 3
17 % 5 = 2
17 ** 5 = 1419857
```

## 비교 연산자

비교 연산자는 불리언 값(True 또는 False)을 반환합니다.

```
x, y = 10, 20
```

```
print(f"{x} == {y}: {x == y}") # 같음
print(f"{x} != {y}: {x != y}") # 같지 않음
print(f"{x} < {y}: {x < y}") # 작음
print(f"{x} > {y}: {x > y}") # 큼
print(f"{x} <= {y}: {x <= y}") # 작거나 같음
print(f"{x} >= {y}: {x >= y}") # 크거나 같음
```

출력:

```
10 == 20: False
```

```
10 != 20: True
```

```
10 < 20: True
```

```
10 > 20: False
```

```
10 <= 20: True
```

```
10 >= 20: False
```

## 입력과 출력

### 포맷된 출력

f-문자열(포맷 문자열 리터럴)은 문자열에 표현식을 깔끔하게 포함할 수 있게 해줍니다.

## 식으 파이썬

```
name = "철수"
age = 30
height = 1.75

# f-문자열 포매팅
print(f"이름: {name}")
print(f"나이: {age}세")
print(f"키: {height:.2f}미터")
print(f"5년 후, {name}은(는) {age + 5}세가 됩니다")
```

출력:

```
이름: 철수
나이: 30세
키: 1.75미터
5년 후, 철수은(는) 35세가 됩니다
```

## 제2장: 제어 흐름

제어 흐름 문장은 프로그램이 조건에 따라 결정을 내리고 동작을 반복할 수 있게 합니다.

### 조건문

#### If-Elif-Else

if, elif, else를 사용하여 조건에 따라 다른 코드 블록을 실행합니다.

```
score = 85

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"

print(f"점수: {score} -> 등급: {grade}")
```

출력:

```
점수: 85 -> 등급: B
```

## 논리 연산자

and, or, not을 사용하여 조건을 결합합니다.

```
age = 25
has_license = True
has_car = False

can_drive = age >= 18 and has_license
print(f"운전 가능: {can_drive}")

needs_transport = not has_car
print(f"교통수단 필요: {needs_transport}")

can_travel = has_car or has_license
print(f"여행 가능: {can_travel}")
```

출력:

```
운전 가능: True
교통수단 필요: True
여행 가능: True
```

## 반복문

### For 반복문

for 반복문은 리스트, <sup>쉬운 파이썬</sup> 문자열, range와 같은 시퀀스를 순회합니다.

```
# 리스트 순회
fruits = ["사과", "바나나", "체리"]
for fruit in fruits:
    print(f"{fruit}을(를) 좋아합니다")

print()

# range로 반복
for i in range(1, 6):
    print(f"카운트: {i}")
```

출력:

사과을(를) 좋아합니다  
바나나을(를) 좋아합니다  
체리을(를) 좋아합니다

카운트: 1  
카운트: 2  
카운트: 3  
카운트: 4  
카운트: 5

## While 반복문

while 반복문은 조건이 참인 동안 반복합니다.

```
count = 0
while count < 5:
    print(f"카운트는 {count}입니다")
    count += 1

print("반복 완료!")
```

출력:

```
카운트는 0입니다  
카운트는 1입니다  
카운트는 2입니다  
카운트는 3입니다  
카운트는 4입니다  
반복 완료!
```

## 반복문 제어

break로 반복문을 종료하고 continue로 다음 반복으로 건너뛸니다.

```
# break 예제
print("첫 번째 짝수 찾기:")
for n in [1, 3, 5, 6, 7, 8]:
    if n % 2 == 0:
        print(f"찾음: {n}")
        break

print()
```

```
# continue 예제
print("홀수만 출력:")
for n in range(1, 8):
    if n % 2 == 0:
        continue
    print(n, end=" ")
print()
```

출력:

첫 번째 짝수 찾기:

찾음: 6

홀수만 출력:

1 3 5 7

## 컴프리헨션

### 리스트 컴프리헨션

리스트 컴프리헨션 문법으로 리스트를 간결하게 생성합니다.

```
# 전통적인 방법
squares_old = []
for x in range(1, 6):
    squares_old.append(x ** 2)

# 리스트 컴프리헨션
squares = [x ** 2 for x in range(1, 6)]
print(f"제곱: {squares}")

# 조건 포함
evens = [x for x in range(10) if x % 2 == 0]
print(f"짝수: {evens}")
```

출력:

```
제곱: [1, 4, 9, 16, 25]
짝수: [0, 2, 4, 6, 8]
```

## 딕셔너리와 세트 컴프리헨션

비슷한 문법이 딕셔너리와 세트에도 적용됩니다.

```
# 딕셔너리 컴프리헨션
word = "hello"
char_positions = {char: i for i, char in enumerate(word)}
print(f"문자 위치: {char_positions}")

# 세트 컴프리헨션 (고유 값)
numbers = [1, 2, 2, 3, 3, 3, 4]
unique_squares = {x ** 2 for x in numbers}
print(f"고유 제곱: {unique_squares}")
```

출력:

```
문자 위치: {'h': 0, 'e': 1, 'l': 3, 'o': 4}
고유 제곱: {16, 1, 4, 9}
```

## 제3장: 데이터 구조

파이썬은 데이터 컬렉션을 구성하고 조작하기 위한 강력한 내장 데이터 구조를 제공합니다.

### 리스트

#### 리스트 연산

리스트는 모든 타입의 데이터를 담을 수 있는 순서가 있는 변경 가능한 컬렉션입니다.

```
# 리스트 생성
numbers = [1, 2, 3, 4, 5]
mixed = [1, "안녕", 3.14, True]

# 요소 접근
print(f"첫 번째: {numbers[0]}, 마지막: {numbers[-1]}")

# 수정
numbers.append(6)
numbers.insert(0, 0)
print(f"추가 및 삽입 후: {numbers}")

# 슬라이싱
print(f"슬라이스 [1:4]: {numbers[1:4]}")
print(f"2번째마다: {numbers[::2]}")
```

출력:

```
첫 번째: 1, 마지막: 5
추가 및 삽입 후: [0, 1, 2, 3, 4, 5, 6]
슬라이스 [1:4]: [1, 2, 3]
2번째마다: [0, 2, 4, 6]
```

## 리스트 메서드

리스트는 많은 유용한 내장 메서드를 가지고 있습니다.

```
fruits = ["바나나", "사과", "체리", "사과"]

# 정렬
fruits.sort()
print(f"정렬됨: {fruits}")

# 카운트와 찾기
print(f"'사과' 개수: {fruits.count('사과')}")
print(f"'체리' 인덱스: {fruits.index('체리')}")

# 제거
fruits.remove("사과") # 첫 번째 항목 제거
print(f"제거 후: {fruits}")

last = fruits.pop()
print(f"꺼낸 값: {last}, 남은 것: {fruits}")
```

출력:

정렬됨: ['바나나', '사과', '사과', '체리']

'사과' 개수: 2

'체리' 인덱스: 3

제거 후: ['바나나', '사과', '체리']

꺼낸 값: 체리, 남은 것: ['바나나', '사과']

## 튜플

### 튜플 다루기

튜플은 변경 불가능한 시퀀스로, 고정된 항목 컬렉션에 자주 사용됩니다.

# 튜플 생성

```
point = (10, 20)
```

```
person = ("철수", 30, "엔지니어")
```

# 접근

```
print(f"점: x={point[0]}, y={point[1]}")
```

# 언패킹

```
name, age, job = person
```

```
print(f"{name}은(는) {age}세이고, {job}로 일합니다")
```

# 반환 값으로서의 튜플

```
def get_min_max(numbers):
```

```
    return min(numbers), max(numbers)
```

```
data = [5, 2, 8, 1, 9]
```

```
minimum, maximum = get_min_max(data)
```

```
print(f"최소: {minimum}, 최대: {maximum}")
```

출력:

점:  $x=10$ ,  $y=20$

철수은(는) 30세이고, 엔지니어로 일합니다

최소: 1, 최대: 9

## 딕셔너리

### 딕셔너리 기초

딕셔너리는 빠른 조회를 위한 키-값 쌍을 저장합니다.

```
# 딕셔너리 생성
person = {
    "이름": "영희",
    "나이": 25,
    "도시": "서울"
}

# 값 접근
print(f"이름: {person['이름']}")
print(f"나이: {person.get('나이')}")

# 추가 및 업데이트
person["이메일"] = "younghee@example.com"
person["나이"] = 26
print(f"업데이트됨: {person}")

# 기본값으로 안전한 접근
country = person.get("국가", "알 수 없음")
print(f"국가: {country}")
```

출력:

이름: 영희

나이: 25

업데이트됨: {'이름': '영희', '나이': 26, '도시': '서울', '이메일': 'younghe...

국가: 알 수 없음

## 딕셔너리 순회

키, 값, 또는 둘 다 순회합니다.

```
scores = {"철수": 95, "영희": 87, "민수": 92}
```

```
# 키
```

```
print("학생들:", list(scores.keys()))
```

```
# 값
```

```
print("점수들:", list(scores.values()))
```

```
# 둘 다
```

```
print("\n모든 항목:")
```

```
for name, score in scores.items():
```

```
    print(f" {name}: {score}")
```

출력:

학생들: ['철수', '영희', '민수']

점수들: [95, 87, 92]

모든 항목:

철수: 95

영희: 87

민수: 92

## 세트

### 세트 연산

세트는 고유한 요소의 순서 없는 컬렉션으로, 멤버십 테스트에 적합합니다.

```
# 세트 생성
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

print(f"세트 A: {a}")
print(f"세트 B: {b}")

# 세트 연산
print(f"합집합 (A | B): {a | b}")
print(f"교집합 (A & B): {a & b}")
print(f"차집합 (A - B): {a - b}")
print(f"대칭 차집합 (A ^ B): {a ^ b}")

# 멤버십
print(f"3이 A에 있음: {3 in a}")
print(f"7이 A에 있음: {7 in a}")
```

출력:

세트 A: {1, 2, 3, 4}

세트 B: {3, 4, 5, 6}

합집합 (A | B): {1, 2, 3, 4, 5, 6}

교집합 (A & B): {3, 4}

차집합 (A - B): {1, 2}

대칭 차집합 (A ^ B): {1, 2, 5, 6}

30이 A에 있음: True

70이 A에 있음: False

## 제4장: 함수

함수는 코드를 재사용 가능한 블록으로 구성하여 프로그램을 더 모듈화하고 유지보수하기 쉽게 만듭니다.

### 함수 정의

#### 기본 함수

`def`를 사용하여 매개변수와 반환 값이 있는 함수를 정의합니다.

```
def greet(name):  
    """인사 메시지를 반환합니다."""  
    return f"안녕하세요, {name}님!"  
  
def add(a, b):  
    """두 숫자를 더합니다."""  
    return a + b  
  
# 함수 사용  
message = greet("파이썬")  
print(message)  
  
result = add(5, 3)  
print(f"5 + 3 = {result}")
```

출력:

```
안녕하세요, 파이썬님!  
5 + 3 = 8
```

## 기본값과 키워드 인자

매개변수는 기본값을 가질 수 있고, 인자는 이름으로 전달할 수 있습니다.

```
def make_coffee(size="중간", milk=False, sugar=0):
    """커피 주문을 설명합니다."""
    order = f"{size} 커피"
    if milk:
        order += " 우유 추가"
    if sugar > 0:
        order += f" 설탕 {sugar}개"
    return order

# 다양한 호출 방법
print(make_coffee())
print(make_coffee("큰"))
print(make_coffee(milk=True, sugar=2))
print(make_coffee("작은", True, 1))
```

출력:

```
중간 커피
큰 커피
중간 커피 우유 추가 설탕 2개
작은 커피 우유 추가 설탕 1개
```

## 고급 매개변수

## \*args와 \*\*kwargs

가변 개수의 위치 인자와 키워드 인자를 받습니다.

```
def sum_all(*args):
    """모든 인자를 더합니다."""
    return sum(args)

def print_info(**kwargs):
    """키-값 쌍을 출력합니다."""
    for key, value in kwargs.items():
        print(f" {key}: {value}")

# *args 사용
print(f"합계: {sum_all(1, 2, 3, 4, 5)}")

# **kwargs 사용
print("개인 정보:")
print_info(이름="철수", 나이=30, 도시="서울")
```

출력:

합계: 15

개인 정보:

이름: 철수

나이: 30

도시: 서울

## 람다 함수

### 익명 함수

람다 함수는 작은 한 줄짜리 익명 함수입니다.

```
# 간단한 람다
square = lambda x: x ** 2
print(f"5의 제곱: {square(5)}")

# 여러 인자를 가진 람다
add = lambda a, b: a + b
print(f"3 + 4 = {add(3, 4)}")

# 일반적인 사용: 정렬
students = [("철수", 85), ("영희", 92), ("민수", 78)]
students.sort(key=lambda x: x[1], reverse=True)
print("점수순 정렬:")
for name, score in students:
    print(f" {name}: {score}")
```

출력:

5의 제곱: 25

$3 + 4 = 7$

점수순 정렬:

영희: 92

철수: 85

민수: 78

## 스코프

### 변수 스코프

변수는 다른 스코프를 가집니다: 지역, 둘러싸인, 전역, 내장 (LEGB 규칙).

```
global_var = "나는 전역 변수입니다"

def outer():
    enclosing_var = "나는 둘러싸인 변수입니다"

    def inner():
        local_var = "나는 지역 변수입니다"
        print(local_var)
        print(enclosing_var)
        print(global_var)

    inner()

outer()

# 전역 변수 수정
counter = 0

def increment():
    global counter
    counter += 1

increment()
increment()
print(f"카운터: {counter}")
```

출력:

```
나는 지역 변수입니다  
나는 둘러싸인 변수입니다  
나는 전역 변수입니다  
카운터: 2
```

## 제5장: 모듈과 패키지

모듈은 코드를 별도의 파일로 구성하는 데 도움을 줍니다.  
파이썬의 표준 라이브러리는 많은 유용한 모듈을 제공합니다.

### 모듈 가져오기

#### Import 문법

`import`를 사용하여 모듈을 로드하고 함수에 접근합니다.

```
import math
from datetime import datetime, timedelta
from random import randint, choice

# math 모듈 사용
print(f"파이: {math.pi:.4f}")
print(f"16의 제곱근: {math.sqrt(16)}")
print(f"4.2의 올림: {math.ceil(4.2)}")

# datetime 사용
now = datetime.now()
print(f"현재 시간: {now.strftime('%Y-%m-%d %H:%M')}")

# random 사용
print(f"1-10 랜덤 숫자: {randint(1, 10)}")
print(f"랜덤 선택: {choice(['사과', '바나나', '체리'])}")
```

출력:

파이: 3.1416

16의 제곱근: 4.0

4.2의 올림: 5

현재 시간: 2026-02-01 22:29

1-10 랜덤 숫자: 4

랜덤 선택: 체리

## 표준 라이브러리 하이라이트

### Collections 모듈

`collections` 모듈은 특수한 컨테이너 데이터 타입을 제공합니다.

```
from collections import Counter, defaultdict, namedtuple
```

```
# Counter - 요소 세기
```

```
words = ["사과", "바나나", "사과", "체리", "바나나", "사과"]
```

```
count = Counter(words)
```

```
print(f"단어 개수: {dict(count)}")
```

```
print(f"가장 많은 것: {count.most_common(2)}")
```

```
# defaultdict - 기본값이 있는 딕셔너리
```

```
grouped = defaultdict(list)
```

```
for word in words:
```

```
    grouped[len(word)].append(word)
```

```
print(f"길이별 그룹화: {dict(grouped)}")
```

```
# namedtuple - 이름이 있는 필드의 튜플
```

```
Point = namedtuple('Point', ['x', 'y'])
```

```
p = Point(10, 20)
```

```
print(f"점: x={p.x}, y={p.y}")
```

출력:

단어 개수: {'사과': 3, '바나나': 2, '체리': 1}

가장 많은 것: [('사과', 3), ('바나나', 2)]

길이별 그룹화: {2: ['사과', '사과', '체리', '사과'], 3: ['바나나', '바나나']}

점: x=10, y=20

## Itertools 모듈

itertools는 일반적인 패턴을 위한 효율적인 이터레이터를 제공합니다.

```
from itertools import count, cycle, islice, combinations,...
```

```
# 조합
```

```
items = ['A', 'B', 'C']
```

```
print("2개의 조합:")
```

```
for combo in combinations(items, 2):
```

```
    print(f" {combo}")
```

```
# 순열
```

```
print("2개의 순열:")
```

```
for perm in permutations(items, 2):
```

```
    print(f" {perm}")
```

```
# islice - 이터레이터 슬라이스
```

```
from itertools import accumulate
```

```
numbers = [1, 2, 3, 4, 5]
```

```
running_sum = list(accumulate(numbers))
```

```
print(f"누적 합: {running_sum}")
```

출력:

2개의 조합:

('A', 'B')

('A', 'C')

('B', 'C')

2개의 순열:

('A', 'B')

('A', 'C')

('B', 'A')

('B', 'C')

('C', 'A')

('C', 'B')

누적 합: [1, 3, 6, 10, 15]

## 제6장: 객체지향 프로그래밍

OOP는 클래스와 객체를 사용하여 실세계 엔티티를 모델링하고, 코드 재사용과 구성을 촉진합니다.

### 클래스와 객체

#### 클래스 정의

클래스는 속성과 메서드를 가진 객체를 생성하기 위한 청사진입니다.

```
class Dog:
    """간단한 Dog 클래스."""

    # 클래스 속성 (모든 인스턴스가 공유)
    species = "개과"

    def __init__(self, name, age):
        """새 Dog를 초기화합니다."""
        self.name = name # 인스턴스 속성
        self.age = age

    def bark(self):
        """개가 짖게 합니다."""
        return f"{self.name}이(가) 멍멍!"

    def describe(self):
        """개를 설명합니다."""
        return f"{self.name}은(는) {self.age}살입니다"

# 객체 생성
buddy = Dog("버디", 3)
max_dog = Dog("맥스", 5)

print(buddy.describe())
print(buddy.bark())
print(f"종: {buddy.species}")
```

출력:

```
버디은(는) 3살입니다  
버디이(가) 멍멍!  
종: 개과
```

## 상속

### 클래스 확장

상속은 클래스가 부모 클래스의 속성과 메서드를 상속받을 수 있게 합니다.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("서브클래스가 구현해야 합니다")

class Cat(Animal):
    def speak(self):
        return f"{self.name}이(가) 야옹!"

class Dog(Animal):
    def speak(self):
        return f"{self.name}이(가) 멍멍!"

class Cow(Animal):
    def speak(self):
        return f"{self.name}이(가) 음메!"

# 다형성 실행
animals = [Cat("나비"), Dog("버디"), Cow("베시")]
for animal in animals:
    print(animal.speak())
```

출력:

나비이(가) 야옹!  
버디이(가) 멍멍!  
베시이(가) 음메!

## 특수 메서드

### 던더 메서드

특수 메서드(던더 메서드)는 객체가 연산자와 내장 함수에서 어떻게 동작하는지 커스터마이징합니다.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __abs__(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

v1 = Vector(3, 4)
v2 = Vector(1, 2)

print(f"v1 = {v1}")
print(f"v2 = {v2}")
print(f"v1 + v2 = {v1 + v2}")
print(f"v1 * 3 = {v1 * 3}")
print(f"|v1| = {abs(v1)}")
```

출력:

```
v1 = Vector(3, 4)
v2 = Vector(1, 2)
v1 + v2 = Vector(4, 6)
v1 * 3 = Vector(9, 12)
|v1| = 5.0
```

## 제7장: 오류 처리

적절한 오류 처리는 예상치 못한 상황을 우아하게 관리하여 프로그램을 견고하고 사용자 친화적으로 만듭니다.

### Try-Except

#### 기본 예외 처리

try-except 블록을 사용하여 오류를 잡고 처리합니다.

```
def divide(a, b):  
    try:  
        result = a / b  
    except ZeroDivisionError:  
        return "오류: 0으로 나눌 수 없습니다!"  
    else:  
        return f"{a} / {b} = {result}"  
    finally:  
        print("나눗셈 시도됨.")  
  
print(divide(10, 2))  
print()  
print(divide(10, 0))
```

출력:

나눗셈 시도됨.

10 / 2 = 5.0

나눗셈 시도됨.

오류: 0으로 나눌 수 없습니다!

## 여러 예외

쉬운 파이썬  
다른 타입의 예외를 다르게 처리합니다.

```
def process_data(data, index):  
    try:  
        value = data[index]  
        result = 100 / value  
        return f"결과: {result}"  
    except IndexError:  
        return "오류: 인덱스 범위 초과"  
    except ZeroDivisionError:  
        return "오류: 0으로 나눔"  
    except TypeError:  
        return "오류: 잘못된 데이터 타입"  
  
data = [10, 0, 5, "abc"]  
  
print(process_data(data, 0))  
print(process_data(data, 1))  
print(process_data(data, 10))  
print(process_data(data, 3))
```

출력:

결과: 10.0

오류: 0으로 나눔

오류: 인덱스 범위 초과

오류: 잘못된 데이터 타입

## 예외 발생시키기

### 커스텀 예외

예외를 발생시켜 오류를 알리고 커스텀 예외 타입을 만듭니다.

```
class ValidationError(Exception):
    """유효성 검사 오류를 위한 커스텀 예외."""
    pass

def validate_age(age):
    if not isinstance(age, int):
        raise TypeError("나이는 정수여야 합니다")
    if age < 0:
        raise ValidationError("나이는 음수일 수 없습니다")
    if age > 150:
        raise ValidationError("나이가 비현실적입니다")
    return True

# 유효성 검사 테스트
test_ages = [25, -5, 200, "서른"]

for age in test_ages:
    try:
        validate_age(age)
        print(f"나이 {age}: 유효함")
    except (TypeError, ValidationError) as e:
        print(f"나이 {age}: {e}")
```

출력:

나이 25: 유효함

나이 -5: 나이는 음수일 수 없습니다

나이 200: 나이가 비현실적입니다

나이 서른: 나이는 정수여야 합니다

## 제8장: 파일과 데이터 처리

파이썬은 파일과 데이터를 읽고, 쓰고, 처리하기 위한 간단하면서도 강력한 도구를 제공합니다.

### 파일 읽기와 쓰기

#### 텍스트 파일

`open()`과 컨텍스트 매니저를 사용하여 파일을 안전하게 처리합니다.

```
import tempfile
import os

# 데모를 위한 임시 파일 생성
with tempfile.NamedTemporaryFile(mode='w', suffix='.txt',...
    temp_path = f.name
    f.write("안녕하세요, 파이썬!\n")
    f.write("이것은 2번째 줄입니다.\n")
    f.write("이것은 3번째 줄입니다.\n")

# 전체 파일 읽기
with open(temp_path, 'r') as f:
    content = f.read()
    print("전체 내용:")
    print(content)

# 줄별로 읽기
print("줄별로:")
with open(temp_path, 'r') as f:
    for i, line in enumerate(f, 1):
        print(f" 줄 {i}: {line.strip()}")

# 정리
os.unlink(temp_path)
```

출력:

전체 내용:

안녕하세요, 파이썬!

이것은 2번째 줄입니다.

이것은 3번째 줄입니다.

줄별로:

줄 1: 안녕하세요, 파이썬!

줄 2: 이것은 2번째 줄입니다.

줄 3: 이것은 3번째 줄입니다.

## JSON 데이터

### JSON 다루기

JSON은 데이터를 저장하고 교환하기 위한 인기 있는 형식입니다.

```
import json

# 파이썬 객체를 JSON으로
data = {
    "이름": "철수",
    "나이": 30,
    "언어": ["파이썬", "자바스크립트", "고"],
    "활성": True
}

# JSON 문자열로 변환
json_str = json.dumps(data, indent=2, ensure_ascii=False)
print("JSON 문자열:")
print(json_str)

print()

# JSON을 파이썬으로 다시 파싱
parsed = json.loads(json_str)
print(f"이름: {parsed['이름']}")
print(f"언어: {' '.join(parsed['언어'])}")
```

출력:

JSON 문자열:

```
{  
  "이름": "철수",  
  "나이": 30,  
  "언어": [  
    "파이썬",  
    "자바스크립트",  
    "고"  
  ],  
  "활성": true  
}
```

이름: 철수

언어: 파이썬, 자바스크립트, 고

## 경로 처리

### pathlib 사용

pathlib 모듈은 파일 경로에 대한 객체지향적 접근 방식을 제공합니다.

```
from pathlib import Path

# 현재 디렉토리
current = Path.cwd()
print(f"현재 디렉토리: {current.name}")

# 경로 연산
example_path = Path("/home/user/documents/report.pdf")
print(f"이름: {example_path.name}")
print(f"스텝: {example_path.stem}")
print(f"확장자: {example_path.suffix}")
print(f"부모: {example_path.parent}")
print(f"부분: {example_path.parts}")

# 경로 구성
new_path = Path("data") / "2024" / "january" / "report.csv"
print(f"구성된 경로: {new_path}")
```

출력:

## 신의 파일서

현재 디렉토리: test-python-book

이름: report.pdf

스텝: report

확장자: .pdf

부모: /home/user/documents

부분: ('/', 'home', 'user', 'documents', 'report.pdf')

구성된 경로: data/2024/january/report.csv

## 제9장: 고급 파이썬 주제

이러한 고급 기능은 더 우아하고, 효율적이며, 파이썬다운 코드를 작성하는 데 도움을 줍니다.

### 이터레이터

#### 커스텀 이터레이터

이터레이터는 모든 것을 메모리에 로드하지 않고 요소에 하나씩 접근하는 방법을 제공합니다.

```
class Countdown:
    """n부터 1까지 카운트다운하는 이터레이터."""

    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= 0:
            raise StopIteration
        self.n -= 1
        return self.n + 1

# 이터레이터 사용
print("5부터 카운트다운:")
for num in Countdown(5):
    print(num, end=" ")
print()
```

출력:

5부터 카운트다운:

5 4 3 2 1

## 제너레이터

### 제너레이터 함수

제너레이터는 `yield`를 사용하여 값을 지연 생성하여 메모리를 절약합니다.

```
def fibonacci(n):
    """처음 n개의 피보나치 수를 생성합니다."""
    a, b = 0, 1
    count = 0
    while count < n:
        yield a
        a, b = b, a + b
        count += 1

# 제너레이터 사용
print("처음 10개의 피보나치 수:")
for num in fibonacci(10):
    print(num, end=" ")
print()

# 제너레이터 표현식
squares = (x**2 for x in range(1, 6))
print(f"제공: {list(squares)}")
```

출력:

처음 10개의 피보나치 수:

0 1 1 2 3 5 8 13 21 34

제공: [1, 4, 9, 16, 25]

## 데코레이터

### 함수 데코레이터

데코레이터는 원본 함수를 수정하지 않고 동작을 추가하기 위해 함수를 감쌉니다.

```
import time

def timer(func):
    """실행 시간을 출력하는 데코레이터."""
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()
        print(f"{func.__name__}은(는) {end - start:.6f}초 걸림")
        return result
    return wrapper

def logger(func):
    """함수 호출을 로깅하는 데코레이터."""
    def wrapper(*args, **kwargs):
        print(f"{func.__name__}을(를) {args}로 호출")
        return func(*args, **kwargs)
    return wrapper

@timer
@logger
def slow_add(a, b):
    time.sleep(0.01) # 느린 작업 시뮬레이션
    return a + b

result = slow_add(3, 5)
```

```
print(f"결과: {result}")
```

쉬운 파이썬

출력:

```
slow_add을(를) (3, 5)로 호출  
wrapper은(는) 0.012513초 걸림  
결과: 8
```

## 컨텍스트 매니저

### 커스텀 컨텍스트 매니저

컨텍스트 매니저는 `with` 문을 사용하여 설정과 정리를 처리합니다.

```

from contextlib import contextmanager

class Timer:
    """코드 블록의 시간을 측정하는 컨텍스트 매니저."""

    def __enter__(self):
        import time
        self.start = time.perf_counter()
        return self

    def __exit__(self, *args):
        import time
        self.elapsed = time.perf_counter() - self.start
        print(f"경과 시간: {self.elapsed:.6f}초")

# 컨텍스트 매니저 사용
with Timer():
    total = sum(range(100000))
    print(f"합계: {total}")

# contextmanager 데코레이터 사용
@contextmanager
def tag(name):
    print(f"<{name}>")
    yield
    print(f"</{name}>")
    
```

## 쉬운 파이썬

```
with tag("html"):
    with tag("body"):
        print(" 안녕하세요, 세계!")
```

출력:

```
합계: 4999950000
경과 시간: 0.000817초
<html>
<body>
  안녕하세요, 세계!
</body>
</html>
```

## 제10장: 테스트링과 모범 사례

테스트를 작성하면 코드가 올바르게 작동하고 변경 시에도 계속 작동하는지 확인할 수 있습니다.

### 유닛 테스트

#### 테스트 작성

파이썬의 unittest 모듈은 테스트를 작성하고 실행하기 위한 프레임워크를 제공합니다.

```
import unittest

def add(a, b):
    return a + b

def divide(a, b):
    if b == 0:
        raise ValueError("0으로 나눌 수 없습니다")
    return a / b

class TestMathFunctions(unittest.TestCase):
    def test_add_positive(self):
        self.assertEqual(add(2, 3), 5)

    def test_add_negative(self):
        self.assertEqual(add(-1, -1), -2)

    def test_divide(self):
        self.assertEqual(divide(10, 2), 5)

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(10, 0)

# 테스트 실행
suite = unittest.TestLoader().loadTestsFromTestCase(TestM...
```

쉬운 파이썬  
`runner = unittest.TextTestRunner(verbosity=2)`  
`runner.run(suite)`

## 타입 힌트

### 타입 어노테이션 추가

타입 힌트는 코드 가독성을 향상시키고 정적 타입 검사를 가능하게 합니다.

```

from typing import Optional

def greet(name: str) -> str:
    """인사 메시지를 반환합니다."""
    return f"안녕하세요, {name}님!"

def find_max(numbers: list[int]) -> Optional[int]:
    """최대 숫자를 찾거나, 리스트가 비어있으면 None을 반환합니다."""
    if not numbers:
        return None
    return max(numbers)

def process_data(
    data: dict[str, int],
    multiplier: float = 1.0
) -> dict[str, float]:
    """모든 값을 승수로 곱합니다."""
    return {k: v * multiplier for k, v in data.items()}

# 함수 사용
print(greet("파이썬"))
print(f"[1, 5, 3]의 최대값: {find_max([1, 5, 3])}")
print(f"[]의 최대값: {find_max([])}")

data = {"a": 10, "b": 20}
print(f"처리됨: {process_data(data, 1.5)}")

```

출력:

안녕하세요, 파이썬님!

[1, 5, 3]의 최대값: 5

[]의 최대값: None

처리됨: {'a': 15.0, 'b': 30.0}

# 제11장: 동시성과 비동기

파이썬은 동시 작업을 처리하는 여러 방법을 제공합니다:  
스레딩, 멀티프로세싱, `async/await`.

## 스레딩

### 기본 스레딩

스레드는 동시 실행을 허용하며, I/O 바운드 작업에 유용합니다.

```
import threading
import time

def worker(name, delay):
    """작업 태스크를 시뮬레이션합니다."""
    print(f"{name} 시작")
    time.sleep(delay)
    print(f"{name} 완료 ({delay}초 후)")

# 스레드 생성
threads = [
    threading.Thread(target=worker, args=("작업자-1", 0.1)),
    threading.Thread(target=worker, args=("작업자-2", 0.05)),
    threading.Thread(target=worker, args=("작업자-3", 0.08)),
]

# 모든 스레드 시작
start = time.perf_counter()
for t in threads:
    t.start()

# 모두 완료될 때까지 대기
for t in threads:
    t.join()

elapsed = time.perf_counter() - start
```

<sup>쉬운 파이썬</sup>  
`print(f"모든 작업자 완료 ({elapsed:.2f}초)에")`

출력:

```
작업자-1 시작
작업자-2 시작
작업자-3 시작
작업자-2 완료 (0.05초 후)
작업자-3 완료 (0.08초 후)
작업자-1 완료 (0.1초 후)
모든 작업자 완료 (0.11초)에
```

## Async/Await

### 비동기 프로그래밍

async/await는 I/O 바운드 작업의 효율적인 처리를 제공합니다.

```
import asyncio

async def fetch_data(name, delay):
    """데이터 가져오기를 시뮬레이션합니다."""
    print(f"{name} 가져오는 중...")
    await asyncio.sleep(delay)
    return f"{name} 데이터"

async def main():
    # 동시에 태스크 실행
    tasks = [
        fetch_data("사용자", 0.1),
        fetch_data("제품", 0.08),
        fetch_data("주문", 0.05),
    ]

    start = asyncio.get_event_loop().time()
    results = await asyncio.gather(*tasks)
    elapsed = asyncio.get_event_loop().time() - start

    print(f"결과: {results}")
    print(f"총 시간: {elapsed:.2f}초")

# 비동기 main 함수 실행
asyncio.run(main())
```

출력:

사용자 가져오는 중...

제품 가져오는 중...

주문 가져오는 중...

결과: ['사용자 데이터', '제품 데이터', '주문 데이터']

총 시간: 0.10초

## 제12장: 실용적인 응용

배운 것을 적용하여 실용적이고 현실적인 예제를 만들어 봅시다.

### 데이터 처리

#### CSV 유사 데이터 처리

일반적인 작업은 구조화된 데이터를 처리하는 것입니다.

# 샘플 데이터 (보통 CSV에서)

```
sales_data = [
    {"제품": "위젯", "수량": 100, "가격": 9.99},
    {"제품": "가젯", "수량": 50, "가격": 24.99},
    {"제품": "위젯", "수량": 75, "가격": 9.99},
    {"제품": "기즈모", "수량": 30, "가격": 49.99},
    {"제품": "가젯", "수량": 25, "가격": 24.99},
]
```

# 제품별 합계 계산

```
from collections import defaultdict
```

```
totals = defaultdict(lambda: {"수량": 0, "매출": 0})
```

```
for item in sales_data:
```

```
    product = item["제품"]
```

```
    totals[product]["수량"] += item["수량"]
```

```
    totals[product]["매출"] += item["수량"] * item["가격"]
```

# 결과 표시

```
print("판매 요약:")
```

```
print("-" * 40)
```

```
for product, data in sorted(totals.items()):
```

```
    print(f"{product:10} | 수량: {data['수량']:4} | ${data['매출']:10.2f}")
```

```
total_revenue = sum(d["매출"] for d in totals.values())
```

```
print("-" * 40)
```

```
print(f"{'합계':10} | | ${total_revenue:,.2f}")
```

쉬운 파이썬

출력:

판매 요약:

가젯 | 수량: 75 | \$1,874.25

기즈모 | 수량: 30 | \$1,499.70

위젯 | 수량: 175 | \$1,748.25

합계 | | \$5,122.20

## 간단한 API 클라이언트

### 간단한 클래스 기반 클라이언트 구축

기능을 깔끔하고 재사용 가능한 클래스로 캡슐화합니다.

```

from dataclasses import dataclass
from typing import Optional
from datetime import datetime

@dataclass
class Task:
    id: int
    title: str
    completed: bool = False
    created_at: datetime = None

    def __post_init__(self):
        if self.created_at is None:
            self.created_at = datetime.now()

class TaskManager:
    def __init__(self):
        self._tasks: dict[int, Task] = {}
        self._next_id = 1

    def add(self, title: str) -> Task:
        task = Task(id=self._next_id, title=title)
        self._tasks[self._next_id] = task
        self._next_id += 1
        return task

```

```

쉬운 파이썬
def complete(self, task_id: int) -> Optional[Task]:
    if task_id in self._tasks:
        self._tasks[task_id].completed = True
        return self._tasks[task_id]
    return None

def list_all(self) -> list[Task]:
    return list(self._tasks.values())

```

```

# TaskManager 사용
manager = TaskManager()
manager.add("파이썬 기초 배우기")
manager.add("연습 문제 풀기")
manager.add("프로젝트 만들기")

```

```

manager.complete(1)

```

```

print("할 일 목록:")
for task in manager.list_all():
    status = "[x]" if task.completed else "[ ]"
    print(f" {status} {task.id}. {task.title}")

```

출력:

할 일 목록:

- ☒ 1. 파이썬 기초 배우기
- ☐ 2. 연습 문제 풀기
- ☐ 3. 프로젝트 만들기