2026.01.21.수

생성일	@2026년 1월 21일 오전 9:22
태그	Python

목차

1. 매개변수
2. 람다
3. 데코레이터
4. OS모듈
5. JSON 데이터 처리
6. datetime

▼ 1) 매개변수

```
# 필수 매개변수
def greet(name):
    return f"안녕하세요, {name}님!"

# 함수 호출 (name은 필수이므로 반드시 제공해야 함)
print(greet("철수"))  # 출력: 안녕하세요, 철수님!
# print(greet())     # 오류 발생: name 매개변수가 필요함

# 기본값 매개변수
def greet_with_time(name, time="아침"):
    return f"{time}에 만나서 반가워요, {name}님!"

# 함수 호출 (time은 기본값이 있으므로 생략 가능)
print(greet_with_time("영희"))          # 출력: 아침에 만나서 반가워요, 영희님!
print(greet_with_time("민수", "저녁"))  # 출력: 저녁에 만나서 반가워요, 민수님!

# 가변 매개변수 (*args)
def sum_all(*numbers):
    """여러 숫자의 합을 계산하는 함수"""
    # 전달된 인자들은 'numbers'라는 튜플(tuple) 형태로 묶여 들어옵니다.
    return sum(numbers)

# 함수 호출 (인자 개수 제한 없음)
# 3개의 인자를 전달하는 경우
print(sum_all(1, 2, 3))       # 출력: 6

# 5개의 인자를 전달하는 경우
print(sum_all(1, 2, 3, 4, 5))  # 출력: 15

# 인자를 하나도 전달하지 않는 경우
# 빈 튜플의 합은 0이므로 오류 없이 0을 반환합니다.
print(sum_all())              # 출력: 0 (인자가 없어도 오류 없음)

# 복합 예제: 가변 매개변수와 일반 매개변수 함께 사용
def make_pizza(size, *toppings):
    """피자 주문 정보를 출력하는 함수"""
    # 첫 번째 인자는 size 매개변수에 할당됩니다.
    print(f"{size}인치 피자를 만듭니다.")
```

```
    # 두 번째 인자부터는 toppings라는 튜플로 묶여 처리됩니다.
    for topping in toppings:
        print(f"- {topping} 추가")

# 함수 호출
# 12는 size로, 나머지 문자열들은 toppings로 전달됩니다.
make_pizza(12, "페퍼로니", "치즈", "올리브")

# 출력 결과:
# 12인치 피자를 만듭니다.
# - 페퍼로니 추가
# - 치즈 추가
# - 올리브 추가

# 키워드 매개변수 (**kwargs)
def create_profile(name, age, **details):
    """사용자 프로필을 생성하는 함수"""
    # 기본 정보를 담은 딕셔너리를 생성합니다.
    profile = {"name": name, "age": age}

    # **details로 전달된 키워드 인자들을 기존 딕셔너리에 추가합니다.
    profile.update(details)

    return profile

# 함수 호출 (추가 정보는 키=값 형태로 전달)
# '직업'과 '취미'가 details 딕셔너리에 담겨 전달됩니다.
print(create_profile("지원", 25, 직업="개발자", 취미="독서"))
# 출력: {'name': '지원', 'age': 25, '직업': '개발자', '취미': '독서'}

# 더 많은 정보나 리스트 형태의 데이터도 자유롭게 전달 가능합니다.
print(create_profile("민준", 30, 전공="컴퓨터공학", 거주지="서울", 언어=["Python", "Java"]))
# 출력: {'name': '민준', 'age': 30, '전공': '컴퓨터공학', '거주지': '서울', '언어': ['Python', 'Java']}

# 키워드 매개변수 (**kwargs)
def create_profile(name, age, **details):
    """사용자 프로필을 생성하는 함수"""
    # 이름과 나이를 포함한 기본 프로필 딕셔너리를 생성합니다.
    profile = {"name": name, "age": age}

    # **details에 담긴 추가 키워드 인자들을 기존 딕셔너리에 합칩니다.
    profile.update(details)

    return profile

# 함수 호출 (추가 정보는 키=값 형태로 전달)
# '직업'과 '취미'가 details 딕셔너리로 묶여 함수 내부로 전달됩니다.
print(create_profile("지원", 25, 직업="개발자", 취미="독서"))
# 출력: {'name': '지원', 'age': 25, '직업': '개발자', '취미': '독서'}

# 문자열 외에도 리스트 등 다양한 자료형을 키워드 인자로 전달할 수 있습니다.
print(create_profile("민준", 30, 전공="컴퓨터공학", 거주지="서울", 언어=["Python", "Java"]))
# 출력: {'name': '민준', 'age': 30, '전공': '컴퓨터공학', '거주지': '서울', '언어': ['Python', 'Java']}

def complex_function(required, optional="기본값", *args, **kwargs):
    """여러 종류의 매개변수를 모두 사용하는 함수"""
    # 필수 매개변수 출력
    print(f"필수 매개변수: {required}")
    # 기본값이 설정된 선택적 매개변수 출력
    print(f"선택적 매개변수: {optional}")
```

```
    # 전달된 모든 위치 인자들을 튜플로 출력
    print(f"가변 위치 매개변수: {args}")
    # 전달된 모든 키워드 인자들을 딕셔너리로 출력
    print(f"가변 키워드 매개변수: {kwargs}")

# 함수 호출
# 순서대로: required, optional, *args(1,2,3), **kwargs(키워드1, 키워드2)
complex_function("필수값", "선택값", 1, 2, 3, 키워드1="값1", 키워드2="값2")

# 출력 결과:
# 필수 매개변수: 필수값
# 선택적 매개변수: 선택값
# 가변 위치 매개변수: (1, 2, 3)
# 가변 키워드 매개변수: {'키워드1': '값1', '키워드2': '값2'}

# 올바른 매개변수 순서
# 필수 매개변수 -> 기본값 매개변수 -> 가변 위치 매개변수(*args) -> 가변 키워드 매개변수(**kwargs)
def correct_order(normal, default="기본값", *args, **kwargs):
    pass

# 잘못된 매개변수 순서 (오류 발생)
# 가변 키워드 매개변수(**kwargs)가 가변 위치 매개변수(*args)보다 앞에 오면 안 됩니다.
# def wrong_order(**kwargs, *args): # SyntaxError: invalid syntax
#     pass

def function_demo(*args, **kwargs):
    # 각 매개변수의 타입과 실제 값을 출력하여 구조를 확인합니다.
    print(f"args 타입: {type(args)}, 값: {args}")
    print(f"kwargs 타입: {type(kwargs)}, 값: {kwargs}")

    # args 접근 방식: 튜플이므로 인덱스를 사용하여 접근합니다.
    if args:
        print("args의 첫 번째 값:", args[0])

    # kwargs 접근 방식: 딕셔너리이므로 키(key)를 사용하여 접근합니다.
    if 'name' in kwargs:
        print("이름:", kwargs['name'])

# 함수 호출 예시
# 1, 2, 3은 args로, name과 age는 kwargs로 전달됩니다.
function_demo(1, 2, 3, name="홍길동", age=30)

# --- 출력 결과 ---
# args 타입: <class 'tuple'>, 값: (1, 2, 3)
# kwargs 타입: <class 'dict'>, 값: {'name': '홍길동', 'age': 30}
# args의 첫 번째 값: 1
# 이름: 홍길동
```

▼ 2) lambda

```
# 일반 함수
def square(x):
    """입력값을 제곱하여 반환하는 일반적인 함수 정의 방식"""
    return x ** 2

# 람다 함수
# lambda 매개변수: 표현식 형태로 작성하며 이름 없이 한 줄로 정의 가능합니다.
square_lambda = lambda x: x ** 2
```

```
# 결과 확인
print(square(5))        # 출력: 25
print(square_lambda(5))  # 출력: 25

# 1. 조건식을 포함하는 람다 함수
# lambda 매개변수: 식1 if 조건 else 식2
# 두 수 중 더 큰 값을 반환하는 로직입니다.
get_max = lambda a, b: a if a > b else b

print(get_max(10, 5))  # 출력: 10


# 2. 여러 매개변수를 받는 람다 함수
# 여러 개의 매개변수를 쉼표(,)로 구분하여 선언할 수 있습니다.
# 사각형의 넓이를 계산하는 로직 ($width \times height$)입니다.
calc_rectangle_area = lambda width, height: width * height
print(calc_rectangle_area(5, 3)) # 출력: 15

# 1. map과 람다 함수 활용
# numbers 리스트의 모든 요소를 제곱하여 새로운 리스트를 생성합니다.
numbers = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x**2, numbers))

print(squares) # 출력: [1, 4, 9, 16, 25]

# 2. filter와 람다 함수 활용
# numbers 리스트([1, 2, 3, 4, 5])에서 짝수만 골라내어 새로운 리스트를 생성합니다.
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even_numbers) # 출력: [2, 4]

# 3. reduce와 람다 함수 활용
# functools 모듈에서 reduce를 불러와야 사용 가능합니다.
from functools import reduce

# numbers 리스트([1, 2, 3, 4, 5])의 모든 요소를 곱한 결과를 계산합니다.
product = reduce(lambda x, y: x * y, numbers)

print(product) # 출력: 120 (1*2*3*4*5)
```

▼ 3) 데코레이터

```
def multiplier(n):
    """입력받은 n을 곱해주는 람다 함수를 반환합니다."""
    return lambda x: x * n

# multiplier 함수를 이용해 '2배'와 '3배' 계산기 객체를 생성합니다.
double = multiplier(2) # n=2가 고정된 람다 함수 반환
triple = multiplier(3) # n=3이 고정된 람다 함수 반환

# 생성된 함수 객체를 호출하여 결과를 확인합니다.
print(double(5)) # 출력: 10 (5 * 2)
print(triple(5)) # 출력: 15 (5 * 3)

# 클로저
def power_function(n):
    """입력받은 n을 지수로 사용하는 거듭제곱 함수를 반환합니다."""
    def power(x):
        # 외부 함수의 변수 n을 내부 함수인 power가 기억(Capture)합니다.
```

```
        return x ** n
    return power

# 제곱 함수 생성 (n=2인 클로저 인스턴스)
square = power_function(2)
# 세제곱 함수 생성 (n=3인 클로저 인스턴스)
cube = power_function(3)

# 생성된 함수 사용
print(square(4)) # 출력: 16 (4의 제곱)
print(cube(3))   # 출력: 27 (3의 세제곱)

# 데코레이터
def 데코레이터_함수(원본_함수):
    """원본 함수를 감싸서 추가 기능을 부여하는 데코레이터입니다."""

    # 래퍼 함수는 원본 함수의 인자를 유연하게 받기 위해 *args, **kwargs를 사용합니다.
    def 래퍼_함수(*args, **kwargs):
        # 1. 원본 함수 실행 전 수행할 작업 (예: 로그 기록, 권한 확인)
        print("작업 시작: 원본 함수를 실행하기 전입니다.")

        # 2. 원본 함수 실행
        결과 = 원본_함수(*args, **kwargs)

        # 3. 원본 함수 실행 후 수행할 작업 (예: 실행 시간 계산, 결과 가공)
        print("작업 종료: 원본 함수 실행이 완료되었습니다.")

        return 결과

    return 래퍼_함수

def validate_input(func):
    """입력값이 0보다 큰지 검증하는 데코레이터"""
    def wrapper(x, y):
        # 입력값 x 또는 y가 0보다 작으면 ValueError를 발생시킵니다.
        if x < 0 or y < 0:
            raise ValueError("입력값은 0보다 커야 합니다.")
        return func(x, y)
    return wrapper

@validate_input
def divide(x, y):
    """나눗셈을 수행하는 함수"""
    # 분모가 0인 경우에 대한 예외 처리를 수행합니다.
    if y == 0:
        raise ZeroDivisionError("0으로 나눌 수 없습니다.")
    return x / y

# 테스트
try:
    print(divide(10, 2))   # 출력: 5.0
    print(divide(-10, 2))  # ValueError 발생 (데코레이터에서 차단)
except ValueError as e:
    print(e)               # 출력: 입력값은 0보다 커야 합니다.
```

▼ 4) OS모듈

```
# os 모듈 임포트
import os
```

```
# 1. 현재 작업 디렉토리 확인
# getcwd(): Get Current Working Directory의 약자로 현재 폴더 경로를 반환합니다.
cwd = os.getcwd()
print(f"현재 디렉토리: {cwd}")

# 2. 디렉토리 변경
# chdir('..'): 상위 디렉토리로 이동합니다.
os.chdir('../')

# 3. 디렉토리 생성 및 삭제
# mkdir: 단일 디렉토리를 생성합니다.
os.mkdir('new_folder')

# makedirs: 중첩된 디렉토리 구조를 한 번에 생성합니다.
# exist_ok=True: 이미 해당 경로가 존재해도 오류를 발생시키지 않습니다.
os.makedirs('path/to/new/folder', exist_ok=True)

# rmdir: 지정한 디렉토리를 삭제합니다.
os.rmdir('new_folder')

# 파일 및 디렉토리 목록
# os.listdir('.'): 지정한 경로(여기서는 현재 폴더 '.')에 있는 모든 파일과 폴더를 리스트로 반환합니다.
files = os.listdir('.')
print(f"현재 디렉토리 파일/폴더: {files}")

# 파일 경로 조작
# os.path.join: 운영체제별 경로 구분자(Windows는 \, Linux/Mac은 /)를 자동으로 고려하여 경로를 결합합니다.
path = os.path.join('folder', 'file.txt')  # os에 맞는 경로 구분자로 결합
dirname = os.path.dirname(path)  # 디렉토리 경로 (결과: 'folder')
basename = os.path.basename(path)  # 파일명 (결과: 'file.txt')

# 파일/디렉토리 상태 확인 (불리언 값 반환)
exists = os.path.exists(path)  # 해당 파일이나 디렉토리가 실제로 존재하는지 여부
is_file = os.path.isfile(path)  # 해당 경로가 '파일'인지 여부
is_dir = os.path.isdir(path)  # 해당 경로가 '디렉토리'인지 여부
```

▼ 5) JSON 데이터 처리

```
import json

# 1. 딕셔너리를 JSON 문자열로 변환 (Serialization)
person = {
    'name': '홍길동',
    'age': 30,
    'skills': ['Python', 'JavaScript'],
    'active': True
}

# json.dumps(): 파이썬 객체를 JSON 형태의 문자열로 바꿉니다.
# ensure_ascii=False: 한글이 깨지지 않고 정상적으로 표시되게 합니다.
# indent=2: 가독성을 위해 2칸 들여쓰기를 적용합니다.
json_str = json.dumps(person, ensure_ascii=False, indent=2)

print(f"JSON 문자열:\n{json_str}")

# 2. JSON 문자열을 딕셔너리로 변환 (Deserialization)
# json.loads(): JSON 형식의 문자열을 다시 파이썬 딕셔너리로 바꿉니다.
person_dict = json.loads(json_str)
```

```
print(f"이름: {person_dict['name']}, 나이: {person_dict['age']}")

import json

# 1. JSON 파일 저장
# 'w' 모드(쓰기 모드)로 파일을 생성하거나 엽니다.
with open('person.json', 'w', encoding='utf-8') as f:
    # json.dump(): 파이썬 객체를 파일 객체(f)에 JSON 형식으로 직접 저장합니다.
    # ensure_ascii=False: 한글 문자가 깨지지 않도록 설정합니다.
    # indent=2: 데이터 구조를 파악하기 쉽게 들여쓰기를 추가합니다.
    json.dump(person, f, ensure_ascii=False, indent=2)

# 2. JSON 파일 로드
# 'r' 모드(읽기 모드)로 파일을 엽니다.
with open('person.json', 'r', encoding='utf-8') as f:
    # json.load(): 파일에서 JSON 데이터를 읽어 파이썬 객체로 변환합니다.
    loaded_person = json.load(f)

print(f"파일에서 로드한 사람: {loaded_person}")
```

▼ 6) datetime

```
from datetime import datetime, date, time, timedelta

# 1. 현재 날짜와 시간
# datetime.now(): 시스템의 현재 날짜와 시간을 마이크로초 단위까지 반환합니다.
now = datetime.now()
print(f"현재: {now}")

# 2. 특정 날짜/시간 생성
# 연, 월, 일, 시, 분, 초 순서대로 인자를 전달하여 객체를 만듭니다.
specific_date = datetime(2023, 12, 31, 23, 59, 59)
print(f"특정 일시: {specific_date}")

# 3. 문자열 형식 변환 (Date → String)
# strftime (string format time): 날짜 객체를 지정한 포맷의 문자열로 바꿉니다.
date_str = now.strftime("%Y-%m-%d %H:%M:%S")
print(f"형식화된 날짜: {date_str}")

# 4. 문자열을 날짜로 파싱 (String → Date)
# strptime (string parse time): 문자열 데이터를 분석하여 날짜 객체로 변환합니다.
parsed_date = datetime.strptime("2023-01-15", "%Y-%m-%d")
print(f"파싱된 날짜: {parsed_date}")

# 1. 날짜 연산
# timedelta(days=1): 현재 시간에 1일을 더하여 내일 날짜를 계산합니다.
tomorrow = now + timedelta(days=1)
print(f"내일: {tomorrow}")

# timedelta(weeks=1): 현재 시간에서 1주일(7일)을 빼서 과거 날짜를 계산합니다.
last_week = now - timedelta(weeks=1)
print(f"일주일 전: {last_week}")

# 2. 날짜 비교
# 파이썬의 비교 연산자(>, <, ==)를 사용하여 날짜의 선후 관계를 판별할 수 있습니다.
if now > parsed_date:
    print("현재가 파싱된 날짜보다 이후입니다.")
```

```
# 3. 날짜와 시간 분리
# date.today(): 시간 정보 없이 오늘 날짜(연-월-일)만 가져옵니다.
today = date.today()
# time(): 특정 시점의 시, 분, 초 정보만 추출하여 별도의 객체로 만듭니다.
current_time = time(now.hour, now.minute, now.second)
print(f"오늘 날짜: {today}, 현재 시간: {current_time}")
```