Python 치트시트

기본 데이터 타입 및 구조

- 숫자형: `int`, `float`, `complex` (복소수)
- 시퀀스 타입:
  - `str`: 불변(immutable) 문자열. `f"name: {name}"` (f-string), `"a" + "b"` (연결), `"a" * 3` (반복).
  - `list`: 가변(mutable) 리스트. `[1, "apple", 3.5]`
  - `tuple`: 불변(immutable) 튜플. `(1, "apple", 3.5)`
- 매핑 타입:
  - `dict`: 키-값 쌍 컬렉션. `{"key": "value", "name": "John"}`
- 집합 타입:
  - `set`: 중복이 없고 순서가 없는 컬렉션. `{1, 2, 3}`. 합집합(`|`), 교집합(`&`), 차집합(`-`) 연산 지원.
  - `frozenset`: 내용을 변경할 수 없는 불변 집합.

제어 흐름

- if-elif-else: 조건문.
- for 루프:
  - `for item in iterable: ...`
  - `for i, value in enumerate(my_list): ...`
- while 루프: `while condition: ...`
- 루프 제어: `break` (종료), `continue` (다음 반복으로 건너뛰기), `else` (루프가 중단 없이 정상적으로 완료되었을 때 실행).
- try-except-else-finally: 예외 처리.

```
try:
    # 실행할 코드
    result = 10 / x
except ZeroDivisionError as e:
    print(f"에러 발생: {e}")
except TypeError:
    print("타입 에러!")
else:
    print("에러 없이 정상 실행되었습니다.")
finally:
    print("이 블록은 항상 실행됩니다.")
```

- with 문: 컨텍스트 관리자. 파일이나 락(lock) 같은 자원을 안전하게 사용하고 자동으로 해제합니다. `with open("file.txt", "r") as f: ...`

함수

- 정의: `def func_name(pos_arg, key_arg="default"): ...`
- 인수 종류:
  - 위치 인수 (Positional): 순서대로 전달되는 인수.
  - 키워드 인수 (Keyword): `name=value` 형태로 전달되는 인수.
  - 기본값 인수 (Default): 호출 시 생략하면 설정된 기본값을 사용하는 인수.
  - 가변 위치 인수 (`*args`): 여러 개의 위치 인수를 튜플로 한꺼번에 받습니다.
  - 가변 키워드 인수 (`**kwargs`): 여러 개의 키워드 인수를 딕셔너리로 한꺼번에 받습니다.
- 람다 함수 (Lambda): 간단한 로직을 위한 한 줄 익명 함수. `lambda args: expression`
- 타입 힌트 (Type Hints):

```
def greet(name: str) -> str:
    return f"Hello, {name}"
```

- 데코레이터 (Decorators): 기존 함수를 수정하지 않고 기능을 추가하거나 확장할 때 사용하는 함수입니다. `@` 구문을 사용합니다.

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("함수 호출 전 작업 실행")
        result = func(*args, **kwargs)
        print("함수 호출 후 작업 실행")
        return result
    return wrapper

@my_decorator
def say_hello():
    print("안녕하세요!")
```

컴프리헨션 및 제너레이터

- 리스트 컴프리헨션: `[expression for item in iterable if condition]`
- 딕셔너리 컴프리헨션: `{key_expr: val_expr for item in iterable if condition}`
- 집합 컴프리헨션: `{expression for item in iterable if condition}`
- 제너레이터 표현식: `(expression for item in iterable if condition)`
  - 메모리를 효율적으로 사용하며, 한 번에 하나의 항목만 생성합니다.
- 제너레이터 함수: `yield` 키워드를 사용하여 함수를 제너레이터로 만듭니다.

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1
```

클래스와 객체 (OOP)

- 정의: `class MyClass: ...`
- 생성자: `def __init__(self, ...): ...`
- 인스턴스 메서드: 첫 번째 인수로 인스턴스 자신인 `self`를 받습니다.
- 상속: `class SubClass(SuperClass): ...`
- `super()`: 부모 클래스의 메서드를 호출할 때 사용합니다.

모듈과 패키지

- `import module_name`
- `from module_name import function_name`
- `from module_name import function_name as fn`
- `import package_name.module_name`

표준 라이브러리 및 Pip

- 주요 모듈: `os`, `sys`, `datetime`, `math`, `random`, `json`
- Pip (패키지 관리):
  - 설치: `pip install <package_name>`
  - 제거: `pip uninstall <package_name>`
  - 목록 확인: `pip list`
  - 요구사항 파일로 설치: `pip install -r requirements.txt`
  - 현재 환경 설정 저장: `pip freeze > requirements.txt`

정규 표현식 (Regex)

```
import re
<str>   = re.sub(r'<regex>', new, text, count=0)  # 패턴과 일치하는 부분을 'new'로 교체
<list>  = re.findall(r'<regex>', text)
# 일치하는 모든 부분을 리스트로 반환
<list>  = re.split(r'<regex>', text, maxsplit=0)  # 패턴을 기준으로 문자열 분리
<Match> = re.search(r'<regex>', text)
# 패턴과 일치하는 첫 번째 부분 탐색
<Match> = re.match(r'<regex>', text)
# 문자열 시작 부분부터 일치 여부 확인
<iter>  = re.finditer(r'<regex>', text)
# 모든 일치 항목을 Match 객체 반복자로 반환
```

- `re.IGNORECASE`, `re.MULTILINE`, `re.DOTALL` 등 다양한 플래그를 지원합니다.

Match 객체

```
<str>   = <Match>.group()      # 전체 일치 문자열 반환
<str>   = <Match>.group(1)     # 첫 번째 괄호 그룹에 해당되는 부분 반환
<tuple> = <Match>.groups()     # 모든 그룹 해당 부분을 튜플로 반환
<int>   = <Match>.start()      # 일치 시작 인덱스
<int>   = <Match>.end()        # 일치 종료 인덱스 (포함되지 않음)
```

특수 시퀀스

- `\d`: 숫자 `[0-9]`와 동일.
- `\w`: 단어 문자(영문, 숫자, 밑줄) `[a-zA-Z0-9_]`와 동일.
- `\s`: 공백 문자(스페이스, 탭, 줄바꿈 등)와 동일.
- 대문자(`\D`, `\W`, `\S`)는 각각 소문자 버전의 부정을 의미합니다.

열거형 (Enum)

서로 연관된 상수들의 집합을 정의하는 클래스입니다.

```
from enum import Enum, auto

class <enum_name>(Enum):
    <member_name> = auto()      # 1부터 자동으로 증가하는 값을 할당
    <member_name> = <value>     # 명시적으로 값을 할당 (중복 가능)
```

- 멤버 접근: `<enum>.<member_name>`, `<enum>['<member_name>']`, `<enum>(<value>)`
- 멤버 속성: `<member>.name`, `<member>.value`

덕 타이핑 (Duck Types)

특정 메서드 집합이 정의되어 있다면 해당 타입으로 간주하는 파이썬의 암시적 타입 시스템입니다.

비교 가능 (Comparable)

- `__eq__(self, other)`: `==` 연산자 동작 정의. 기본값은 `self is other`입니다.

해시 가능 (Hashable)

- `__hash__`와 `__eq__` 메서드가 모두 필요하며, 객체의 해시 값은 변하지 않아야 합니다.

정렬 가능 (Sortable)

- `functools.total_ordering` 데코레이터를 사용하면 `__eq__`와 하나의 비교 메서드(`__lt__`, `__gt__` 등)만 정의해도 나머지 비교 연산자가 자동으로 생성됩니다.

이터레이터 (Iterator)

- `__next__`와 `__iter__` 메서드가 있는 객체입니다.
- `__next__()`는 다음 항목을 반환하거나 `StopIteration` 예외를 발생시켜야 하며, `__iter__()`는 이터레이터 자신을 반환합니다.

컨텍스트 관리자 (Context Manager)

- `__enter__`와 `__exit__` 메서드가 정의된 객체에서 `with` 문을 사용할 수 있습니다.
- `__enter__()`는 자원을 획득하고, `__exit__()`는 자원을 해제하는 역할을 합니다.

시스템 및 데이터 처리

경로 (Paths)

```
import os, glob
from pathlib import Path

# 현재 작업 디렉토리 가져오기
path_str = os.getcwd()
path_obj = Path.cwd()

# 경로 결합
full_path = os.path.join(path_str,
'dir', 'file.txt')
full_path_obj = path_obj / 'dir' /
'file.txt'

# 파일 및 디렉토리 목록 확인
file_list = os.listdir(path_str)
path_iter = path_obj.iterdir()
```

JSON

```
import json
<str>  = json.dumps(<list/dict>)    # 파
이썬 객체를 JSON 문자열로 변환
<coll> = json.loads(<str>)          #
JSON 문자열을 파이썬 객체로 변환
```

Pickle

파이썬 객체를 저장하기 위한 바이너리 직렬화 형식입니다.

```
import pickle
<bytes>  = pickle.dumps(<object>)   # 객
체를 바이트로 변환
<object> = pickle.loads(<bytes>)    # 바
이트를 객체로 변환
```

CSV

```
import csv
with open('data.csv', newline='',
encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

SQLite

서버 설정이 필요 없는 내장 데이터베이스 엔진입니다.

```
import sqlite3
conn = sqlite3.connect('example.db') #
파일 열기 또는 생성
cursor = conn.execute('SELECT * FROM
stocks')
rows = cursor.fetchall()
conn.close()
```

고급 주제

로깅 (Logging)

```
import logging
logging.basicConfig(level=logging.INFO,
filename='app.log', filemode='w',
                    format='%(name)s -
%(levelname)s - %(message)s')
logging.warning('로그 파일에 기록될 경고 메
시지입니다.')
```

스레딩 (Threading)

```
import threading
def worker():
    print('작업 스레드 실행 중')

t = threading.Thread(target=worker)
t.start()
```

코루틴 (Coroutines) / Asyncio

```
import asyncio

async def main():
    print('Hello')
    await asyncio.sleep(1)
    print('World')

asyncio.run(main())
```

고급 데코레이터 패턴

```
from functools import wraps
import time

def timing_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} 함수 실행
시간: {end - start:.4f}초")
        return result
    return wrapper

def retry(max_attempts=3):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in
range(max_attempts):
                try:
                    return func(*args,
**kwargs)
                except Exception as e:
                    if attempt ==
max_attempts - 1:
                        raise e
                    print(f"{attempt +
1}차 시도 실패: {e}")
            return None
        return wrapper
    return decorator

@timing_decorator
@retry(max_attempts=3)
def risky_function():
    # 위험한 작업 수행
    pass
```

메타클래스 (Metaclasses)

```
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] =
super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Database(metaclass=SingletonMeta):
    def __init__(self):
        self.connection = "데이터베이스_연
결_객체"

# 두 인스턴스는 동일한 객체임을 보장합니다.
db1 = Database()
db2 = Database()
print(db1 is db2)  # True
```

컨텍스트 관리자 심화

```
from contextlib import contextmanager,
ExitStack

@contextmanager
def file_manager(filename, mode):
    file = open(filename, mode)
    try:
        yield file
    finally:
        file.close()

# 여러 리소스를 한꺼번에 관리할 때 유용합니다.
@contextmanager
def multi_resource_manager():
    with ExitStack() as stack:
        file1 =
stack.enter_context(open('file1.txt',
'r'))
        file2 =
stack.enter_context(open('file2.txt',
'w'))
        yield file1, file2
```

데이터 클래스와 타입 힌트 심화

```
from dataclasses import dataclass, field
from typing import List, Optional,
Union, Dict, Any
from enum import Enum
```

```
class Status(Enum):
    PENDING = "대기"
    RUNNING = "실행 중"
    COMPLETED = "완료"
    FAILED = "실패"

@dataclass
class Task:
    id: int
    name: str
    status: Status = Status.PENDING
    dependencies: List[int] =
field(default_factory=list)
    metadata: Optional[Dict[str, Any]] =
None

    def __post_init__(self):
        if not self.name:
            raise ValueError("작업 이름은
비어 있을 수 없습니다.")

# 사용 예제
task = Task(
    id=1,
    name="데이터 처리",
    dependencies=[2, 3],
    metadata={"priority": "high"}
)
```

비동기 프로그래밍 심화

```
import asyncio
import aiohttp
from typing import List

async def fetch_url(session:
aiohttp.ClientSession, url: str) -> str:
    async with session.get(url) as
response:
        return await response.text()

async def fetch_multiple_urls(urls:
List[str]) -> List[str]:
    async with aiohttp.ClientSession()
as session:
        tasks = [fetch_url(session, url)
for url in urls]
        return await
asyncio.gather(*tasks)

# 비동기 제너레이터 예제
async def async_generator():
    for i in range(5):
        await asyncio.sleep(0.1)
        yield i

async def consume_async_generator():
    async for value in
async_generator():
        print(f"수신 데이터: {value}")
```

프로퍼티와 디스크립터

```
class Temperature:
    def __init__(self):
        self._celsius = 0

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("온도는 절대
영도보다 낮을 수 없습니다.")
        self._celsius = value

    @property
    def fahrenheit(self):
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) *
5/9

# 디스크립터(Descriptor) 예제
class ValidatedAttribute:
    def __init__(self, validator):
        self.validator = validator
        self.name = None

    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, instance, owner):
        return
instance.__dict__.get(self.name)

    def __set__(self, instance, value):
        if not self.validator(value):
            raise ValueError(f"유효하지
않은 값입니다: {value}")
        instance.__dict__[self.name] =
value

def positive_number(value):
    return isinstance(value, (int,
float)) and value > 0

class Product:
    price =
ValidatedAttribute(positive_number)

    def __init__(self, price):
        self.price = price
```

함수형 프로그래밍 패턴

```
from functools import reduce, partial
from itertools import chain, groupby
from operator import itemgetter

# 함수 합성(Composition)
def compose(*functions):
    return reduce(lambda f, g: lambda x:
f(g(x)), functions, lambda x: x)

# 파이프라인 구성 예제
def add_one(x): return x + 1
def multiply_by_two(x): return x * 2
def square(x): return x ** 2

pipeline = compose(square,
multiply_by_two, add_one)
result = pipeline(3)  # ((3 + 1) * 2) **
2 = 64

# 부분 적용(Partial Application)
def multiply(x, y):
    return x * y

double = partial(multiply, 2)
triple = partial(multiply, 3)

# 그룹화 예제
data = [('A', 1), ('B', 2), ('A', 3),
('B', 4)]
grouped = {k: list(v) for k, v in
groupby(sorted(data),
key=itemgetter(0))}
```

성능 최적화 기법

```
# 메모이제이션(Memoization)
from functools import lru_cache

@lru_cache(maxsize=128)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) +
fibonacci(n-2)

# 제너레이터 표현식으로 메모리 효율성 향상
def process_large_file(filename):
    with open(filename, 'r') as file:
        # 제너레이터 표현식으로 한 줄씩 순차
처리
        processed_lines =
(line.strip().upper() for line in file
if line.strip())
        return sum(len(line) for line in
processed_lines)

# __slots__를 사용한 메모리 최적화
class Point:
    __slots__ = ['x', 'y']

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

Google Style Guide

명명 규칙 (Naming)

- 모듈: `lower_with_under.py`
- 패키지: `lower_with_under`
- 클래스: `CapWords` (UpperCamelCase)
- 함수 및 메서드: `lower_with_under()`
- 변수 (로컬/전역): `lower_with_under`
- 상수: `CAPS_WITH_UNDER`
- 비공개(Internal): 앞부분에 언더스코어 하나 추가 (`_single_leading_underscore`)

포매팅 (Formatting)

- 들여쓰기: 공백 4개 (탭 사용 금지)

- 줄 길이: 최대 80자
- 세미콜론: 문장 끝에 세미콜론(;) 사용 금지
- 괄호: 불필요한 괄호 사용 자제 (단, 줄 바꿈을 위한 묵시적 줄 결합에는 권장)
- 빈 줄: 최상위 정의(클래스/함수) 사이에는 2줄, 메서드 사이에는 1줄

프로그래밍 관례

- Imports: 패키지와 모듈만 import (함수나 클래스 직접 import 지양). 항상 전체 경로(Full Path) 사용.
- Exceptions: 예외를 사용하여 흐름 제어. `except:` 처럼 모든 예외를 잡는 구문 지양 (`Exception` 명시 권장).
- Global State: 가변 전역 상태 사용 지양.
- Type Annotations: 공용 API에는 타입 힌트 추가 권장.
- Docstrings: 모든 함수, 클래스, 모듈에 작성 (`"""...."""` 형식).

주요 Do's & Don'ts

- Do: 기본 이터레이터와 연산자 사용 (`if x:` 는 `if len(x) != 0:` 보다 선호됨).
- Do: 파일/소켓 처리는 `with` 문 사용.
- Don't: 가변 객체(List, Dict 등)를 함수의 기본 인자로 사용 금지.
- Don't: `assert` 문을 핵심 로직 검증에 사용 금지 (최적화 시 무시될 수 있음).
- Don't: 람다 함수가 복잡해지면 일반 함수로 정의.