Python 치트시트

1. 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. `{"key": "value", "name": "John"}`
- 집합 타입:
  - `set`: 중복 없는 순서 없는 컬렉션. `{1, 2, 3}`. 합집합(`|`), 교집합(`&`), 차집합(`-`).
  - `frozenset`: 불변 집합.

2. 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` (건너뛰기), `else` (루프가 정상적으로 완료됐을 때 실행).
- try-except-else-finally: 예외 처리.

```
try:
    # 실행할 코드
    result = 10 / x
except ZeroDivisionError as e:
    print(f"Error: {e}")
except TypeError:
    print("Type error!")
else:
    print("No errors occurred.")
finally:
    print("This always runs.")
```

- with 문: 컨텍스트 관리자. 파일, 락 등 자원을 안전하게 사용하고 자동 해제. `with open("file.txt", "r") as f: ...`

3. 함수

- 정의: `def func_name(pos_arg, key_arg="default"): ...`
- 인수 종류:
  - 위치 인수 (Positional): 순서대로 전달.
  - 키워드 인수 (Keyword): `name=value` 형태로 전달.
  - 기본값 인수 (Default): 호출 시 생략 가능.
  - 가변 위치 인수 (`*args`): 여러 위치 인수를 튜플로 묶어 받음.
  - 가변 키워드 인수 (`**kwargs`): 여러 키워드 인수를 딕셔너리로 묶어 받음.
- 람다 함수 (Lambda): 한 줄로 된 익명 함수. `lambda args: expression`
- 타입 힌트 (Type Hints):

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

- 데코레이터 (Decorators): 함수를 수정하지 않고 기능을 추가하는 함수. `@` 구문 사용.

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

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

4. 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 사용. 한 번에 하나의 항목만 생성.
- 제너레이터 함수: `yield` 키워드를 사용하여 함수를 제너레이터로 만듦.

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

5. 클래스와 객체 (OOP)

- 정의: `class MyClass: ...`
- 생성자: `def __init__(self, ...): ...`
- 인스턴스 메서드: 첫 인수로 `self`를 받음.
- 상속: `class SubClass(SuperClass): ...`
- `super()`: 부모 클래스의 메서드 호출.

6. 모듈과 패키지

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

7. 표준 라이브러리 및 Pip

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

8. 정규 표현식 (Regex)

```
import re
<str>   = re.sub(r'<regex>', new, text, count=0)  # 모든 발생을 'new'로 대체
<list>  = re.findall(r'<regex>', text)
# 패턴의 모든 발생을 반환
<list>  = re.split(r'<regex>', text, maxsplit=0)  # 일치 항목을 유지하려면 정규식 주위에 괄호 추가
<Match> = re.search(r'<regex>', text)
# 패턴의 첫 번째 발생 또는 None
<Match> = re.match(r'<regex>', text)
# 텍스트 시작 부분에서만 검색
<iter>  = re.finditer(r'<regex>', text)
# 모든 발생을 Match 객체로 반환
```

- `re.IGNORECASE`, `re.MULTILINE`, `re.DOTALL` 플래그 사용 가능.

Match 객체

```
<str>   = <Match>.group()      # 전체 일치 항목 반환
<str>   = <Match>.group(1)     # 첫 번째 괄호 안의 부분 반환
<tuple> = <Match>.groups()     # 모든 괄호 부분을 튜플로 반환
<int>   = <Match>.start()      # 전체 일치 항목의 시작 인덱스
<int>   = <Match>.end()        # 전체 일치 항목의 끝 인덱스 (제외)
```

특수 시퀀스

- `\d`: 숫자 `[0-9]`
- `\w`: 단어 문자 `[a-zA-Z0-9_]`
- `\s`: 공백 문자 `[ \t\n\r\f\v]`
- 대문자(`\D`, `\W`, `\S`)는 각각의 부정을 의미.

9. 열거형 (Enum)

이름이 지정된 상수들의 클래스입니다.

```
from enum import Enum, auto

class <enum_name>(Enum):
    <member_name> = auto()      # 이전 숫자 값에서 1 증가 또는 1
    <member_name> = <value>     # 값은 해시 가능하거나 고유할 필요 없음
```

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

10. 덕 타이핑 (Duck Types)

특정 특수 메서드 집합을 규정하는 암시적 타입. 해당 메서드가 정의된 모든 객체는 해당 덕 타입의 멤버로 간주됩니다.

비교 가능 (Comparable)

- `__eq__(self, other)`: `==` 연산자. 정의되지 않으면 `self is other`를 반환.

해시 가능 (Hashable)

- `__hash__` 및 `__eq__` 메서드가 필요하며 해시 값은 변경되지 않아야 합니다.
- 동일하게 비교되는 해시 가능한 객체는 동일한 해시 값을 가져야 합니다.

정렬 가능 (Sortable)

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

이터레이터 (Iterator)

- `__next__`와 `__iter__` 메서드가 있는 모든 객체.
- `__next__()`는 다음 항목을 반환하거나 `StopIteration` 예외를 발생시켜야 합니다.
- `__iter__()`는 이터레이터 자신을 반환해야 합니다.

컨텍스트 관리자 (Context Manager)

- `with` 문은 `__enter__`와 `__exit__` 특수 메서드가 있는 객체에서만 작동합니다.
- `__enter__()`는 리소스를 잠그고 선택적으로 객체를 반환해야 합니다.
- `__exit__()`는 리소스를 해제해야 합니다.

11. 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/디렉토리 목록
file_list = os.listdir(path_str)
path_iter = path_obj.iterdir()
```

JSON

```
import json
<str>  = json.dumps(<list/dict>)     # 컬
렉션을 JSON 문자열로 변환
<coll> = json.loads(<str>)           #
JSON 문자열을 컬렉션으로 변환
```

Pickle

파이썬 객체 저장을 위한 바이너리 형식.

```
import pickle
<bytes>  = pickle.dumps(<object>)    # 객
체를 바이트 객체로 변환
<object> = pickle.loads(<bytes>)     # 바
이트 객체를 객체로 변환
```

CSV

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

SQLite

서버가 없는 데이터베이스 엔진.

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

12. 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('This will get logged to
a file')
```

스레딩 (Threading)

```
import threading
def worker():
    print('Worker')

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__} took
{end - start:.4f} seconds")
        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
{attempt + 1} failed: {e}")
            return None
        return wrapper
    return decorator

@timing_decorator
@retry(max_attempts=3)
def risky_function():
    # Some risky operation
    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 =
"database_connection"

# 두 인스턴스는 동일한 객체
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 리소스 관리
@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 = "pending"
    RUNNING = "running"
    COMPLETED = "completed"
    FAILED = "failed"
```

```python
@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("Task name
cannot be empty")

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

비동기 프로그래밍 심화

```python
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
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"Received: {value}")
```

프로퍼티와 디스크립터

```python
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("Temperature below absolute
zero")
        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 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"Invalid
value: {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

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

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

# 파이프라인 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
def multiply(x, y):
    return x * y

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

# 그룹화
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

```python
# 메모이제이션
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
def process_large_file(filename):
    with open(filename, 'r') as file:
        # 제너레이터 표현식으로 한 번에 하나
씩 처리
        processed_lines =
(line.strip().upper() for line in file
if line.strip())
        return sum(len(line) for line in
processed_lines)

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

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