Python Cheatsheet

Basic Data Types and Structures

- Numeric Types: `int`, `float`, `complex` (complex numbers)
- Sequence Types:
  - `str`: Immutable string. `f"name: {name}"` (f-string), `"a" + "b"` (concatenation), `"a" * 3` (repetition).
  - `list`: Mutable list. `[1, "apple", 3.5]`
  - `tuple`: Immutable tuple. `(1, "apple", 3.5)`
- Mapping Types:
  - `dict`: Key-value pair collection. `{"key": "value", "name": "John"}`
- Set Types:
  - `set`: Unordered collection of unique elements. `{1, 2, 3}`. Supports union (`|`), intersection (`&`), and difference (`-`) operations.
  - `frozenset`: Immutable version of a set.

Control Flow

- if-elif-else: Conditional statements.
- for Loops:
  - `for item in iterable: ...`
  - `for i, value in enumerate(my_list): ...`
- while Loops: `while condition: ...`
- Loop Control: `break` (exit), `continue` (skip to next iteration), `else` (executed if the loop completes normally without interruption).
- try-except-else-finally: Exception handling.

```
try:
    # Code to execute
    result = 10 / x
except ZeroDivisionError as e:
    print(f"Error occurred: {e}")
except TypeError:
    print("Type Error!")
else:
    print("Executed successfully without errors.")
finally:
    print("This block always executes.")
```

- with Statement: Context managers. Safely use and automatically release resources like files or locks. `with open("file.txt", "r") as f: ...`

Functions

- Definition: `def func_name(pos_arg, key_arg="default"): ...`
- Argument Types:
  - Positional: Arguments passed in sequence.
  - Keyword: Arguments passed in `name=value` format.
  - Default: Arguments that use a preset value if omitted during call.
  - Variadic Positional (`*args`): Receives multiple positional arguments as a tuple.
  - Variadic Keyword (`**kwargs`): Receives multiple keyword arguments as a dictionary.
- Lambda Functions: Single-line anonymous functions for simple logic. `lambda args: expression`
- Type Hints:

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

- Decorators: Functions used to add or extend functionality of existing functions without modification. Uses `@` syntax.

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Work before function call")
        result = func(*args, **kwargs)
        print("Work after function call")
        return result
    return wrapper

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

Comprehensions and Generators

- List Comprehension: `[expression for item in iterable if condition]`
- Dictionary Comprehension: `{key_expr: val_expr for item in iterable if condition}`
- Set Comprehension: `{expression for item in iterable if condition}`
- Generator Expression: `(expression for item in iterable if condition)`
  - Efficiently uses memory, yielding one item at a time.
- Generator Function: Uses `yield` keyword to create a generator.

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

Classes and Objects (OOP)

- Definition: `class MyClass: ...`
- Constructor: `def __init__(self, ...): ...`
- Instance Method: Receives `self` (the instance itself) as the first argument.
- Inheritance: `class SubClass(SuperClass): ...`
- `super()`: Used to call methods of the parent class.

Modules and Packages

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

Standard Library and Pip

- Major Modules: `os`, `sys`, `datetime`, `math`, `random`, `json`
- Pip (Package Management):
  - Install: `pip install <package_name>`
  - Uninstall: `pip uninstall <package_name>`
  - List: `pip list`
  - Install from requirements: `pip install -r requirements.txt`
  - Save environment settings: `pip freeze > requirements.txt`

Regular Expressions (Regex)

```
import re
<str>   = re.sub(r'<regex>', new, text, count=0)  # Replace matches with 'new'
<list>  = re.findall(r'<regex>', text)
# Return all matches as a list
<list>  = re.split(r'<regex>', text, maxsplit=0)  # Split string based on pattern
<Match> = re.search(r'<regex>', text)
# Search for the first match
<Match> = re.match(r'<regex>', text)
# Check match from the start of string
<iter>  = re.finditer(r'<regex>', text)
# Return all matches as Match iterator
```

- Supports various flags like `re.IGNORECASE`, `re.MULTILINE`, `re.DOTALL`.

Match Object

```
<str>   = <Match>.group()      # Return the whole matched string
<str>   = <Match>.group(1)     # Return part matched by the first parenthetical group
<tuple> = <Match>.groups()     # Return all group matches as a tuple
<int>   = <Match>.start()      # Start index of match
<int>   = <Match>.end()        # End index of match (exclusive)
```

Special Sequences

- `\d`: Digit, same as `[0-9]`.
- `\w`: Word character (alphanumeric, underscore), same as `[a-zA-Z0-9_]`.
- `\s`: Whitespace character (space, tab, newline, etc.).
- Uppercase versions (`\D`, `\W`, `\S`) represent the negation of the lowercase versions.

Enumerations (Enum)

A class to define a set of related constants.

```
from enum import Enum, auto

class <enum_name>(Enum):
    <member_name> = auto()      # Assign auto-incrementing values starting from 1
```

```
    <member_name> = <value>     # Explicitly assign values (can be duplicates)
```

- Member Access: `<enum>.<member_name>`, `<enum>['<member_name>']`, `<enum>(<value>)`
- Member Attributes: `<member>.name`, `<member>.value`

Duck Typing

Python's implicit typing system where if an object defines a certain set of methods, it is considered that type.

Comparable

- `__eq__(self, other)`: Defines `==` operator behavior. Default is `self is other`.

Hashable

- Requires both `__hash__` and `__eq__`, and the object's hash value must remain constant.

Sortable

- Using `@functools.total_ordering` decorator allows defining `__eq__` and one comparison method (`__lt__`, `__gt__`, etc.) to automatically generate others.

Iterator

- An object with `__next__` and `__iter__` methods.
- `__next__()` returns the next item or raises `StopIteration`; `__iter__()` returns the iterator itself.

Context Manager

- Objects defining `__enter__` and `__exit__` methods can be used with `with` statements.
- `__enter__()` acquires resources, and `__exit__()` releases them.

System and Data Processing

Paths

```
import os, glob
from pathlib import Path

# Get current working directory
path_str = os.getcwd()
path_obj = Path.cwd()

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

# List files and directories
file_list = os.listdir(path_str)
path_iter = path_obj.iterdir()
```

JSON

```
import json
<str>  = json.dumps(<list/dict>)    # Convert Python object to JSON string
<coll> = json.loads(<str>)          # Convert JSON string to Python object
```

Pickle

Binary serialization format for storing Python objects.

```
import pickle
<bytes>  = pickle.dumps(<object>)   # Convert object to bytes
<object> = pickle.loads(<bytes>)    # Convert bytes to object
```

CSV

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

SQLite

Built-in database engine that requires no server setup.

```
import sqlite3
conn = sqlite3.connect('example.db') # Open or create file
cursor = conn.execute('SELECT * FROM stocks')
rows = cursor.fetchall()
conn.close()
```

Advanced Topics

Logging

```
import logging
logging.basicConfig(level=logging.INFO, filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')
logging.warning('A warning message recorded in the log file.')
```

Threading

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

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

Coroutines / Asyncio

```
import asyncio

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

asyncio.run(main())
```

Advanced Decorator Patterns

```
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"Execution time for {func.__name__}: {end - start:.4f}s")
        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():
    # Perform risky task
    pass
```

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_Object"

# Guaranteed that both instances are the same object.
db1 = Database()
db2 = Database()
print(db1 is db2)  # True
```

Advanced Context Managers

```
from contextlib import contextmanager, ExitStack

@contextmanager
def file_manager(filename, mode):
    file = open(filename, mode)
    try:
```

```
        yield file
    finally:
        file.close()

# Useful for managing multiple resources simultaneously.
@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
```

Advanced Dataclasses and Type Hints

```
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"

@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.")

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

Advanced Async Programming

```
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)

# Async generator example
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 data: {value}")
```

Properties and Descriptors

```
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 cannot be 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

# Descriptor example
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
```

Functional Programming Patterns

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

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

# Pipeline configuration example
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

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

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

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

Performance Optimization Techniques

```
# 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)

# Improve memory efficiency with generator expressions
def process_large_file(filename):
    with open(filename, 'r') as file:
        # Sequential processing line by line with generator expression
        processed_lines = (line.strip().upper() for line in file if line.strip())
        return sum(len(line) for line in processed_lines)
```

```python
# Memory optimization using __slots__
class Point:
    __slots__ = ['x', 'y']

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

Google Style Guide

Naming Conventions

- Modules: `lower_with_under.py`
- Packages: `lower_with_under`
- Classes: `CapWords` (UpperCamelCase)
- Functions and Methods: `lower_with_under()`
- Variables (local/global): `lower_with_under`
- Constants: `CAPS_WITH_UNDER`
- Internal: Prefix with a single underscore (`_single_leading_underscore`)

Formatting

- Indentation: 4 spaces (no tabs)
- Line Length: Max 80 characters
- Semicolons: Do not use at the end of statements
- Parentheses: Avoid unnecessary parentheses (recommended only for implicit line joining)
- Blank Lines: 2 lines between top-level definitions, 1 line between methods

Programming Practices

- Imports: Import packages and modules only (avoid direct import of functions/classes). Always use full paths.
- Exceptions: Use for flow control. Avoid catch-all `except:` (prefer `Exception`).
- Global State: Avoid mutable global state.
- Type Annotations: Recommended for public APIs.
- Docstrings: Required for all functions, classes, and modules (`"""..."""` format).

Essential Do's & Don'ts

- Do: Use default iterators and operators (`if x:` preferred over `if len(x) != 0:`).
- Do: Use `with` statements for file/socket handling.
- Don't: Use mutable objects (List, Dict, etc.) as default function arguments.
- Don't: Use `assert` for core logic verification (can be ignored during optimization).
- Don't: Use lambda functions if they become complex; define a regular function instead.