My Notes in Python

Index

Variable Types

- When an object is initiaded, it is assigned a unique object id. It’s type is defined at runtime and once set can never change, however it’s state can be changed if it is mutable. Simple put, a mutable object can be changed after it is created; immutable object can’t.
- Mutable Object: Objects that can change after creation. (list, set, dict)
- Immutable Object: Objects that can’t change after creation. (int, float, bool, str, tuple, unicode)
- Immutable Table:

Usage of Asterisk - Send Multiple Parameters to Function with *Args and **Kwargs

- *args: Non-Keyword Arguments
  - Using the *, the variable that we associate with the * becomes an iterable meaning you can do things like iterate over it, run some higher-order functions such as map and filter, etc.
- **kwargs: Keyword Arguments
  - A keyword argument is where you provide a name to the variable as you pass it into the function.

- We can unpack variables with using args and kwargs.
- single asterisk (*) is using for unpack iterables. — result return as tuple.
- double asterisk (**) is using for unpack dictionaries. — result return as dict.

Unpacking With args or kwargs

```
# unpacking.py

new_array = [6, 7, 8, 9, 10, 11]
print("Our List: ", new_array, "\n")
print("Unpack our list with * -> Asterisk")
print(*new_array, "\n")

a, *b, c = new_array
print(f"Unpack {new_array} with multiple *args")
print(a, b, c)
```

Output

```
Our List:  [6, 7, 8, 9, 10, 11]

Unpack our list with * -> Asterisk
6 7 8 9 10 11

Unpack [6, 7, 8, 9, 10, 11] with multiple *args
6 [7, 8, 9, 10] 11
```

- We can merge items with using args or kwargs (depends on data type)

Merging items with Asterisk Operator

```
# merging.py

first_list = [5, 6, 7]
second_list = [5, 66, 77]
print(f"Two lists: {first_list} and {second_list}")
merged_list = [*first_list, *second_list]
print(f"Merged List with Asterisk: {merged_list} \n\n")

first_dict = {
    "a": 2,
    "b": 3,
    "c": 4
}

second_dict = {
    "c": 6,
    "d": 41,
    "e": 19
}

print(f"Two dicts: {first_dict} and {second_dict}")
merged_dict = {**first_dict, **second_dict}
print(f"Merged Dict With Asterisk: {merged_dict} \n\n")
```

Output

```
Two lists: [5, 6, 7] and [5, 66, 77]
Merged List with Asterisk: [5, 6, 7, 5, 66, 77]


Two dicts: {'a': 2, 'b': 3, 'c': 4} and {'c': 6, 'd': 41, 'e': 19}
Merged Dict With Asterisk: {'a': 2, 'b': 3, 'c': 6, 'd': 41, 'e': 19}
```

- args or kwargs names are just names. We can change this names when we are using. Important part is the asterisk(*)
- We can send parameters to a function without using args or kwargs, of course. But sometimes we wouldn't know how many parameters come to our function. In that cases, we can use args or kwargs.

Multiple Parameters Wihtout Args/Kwargs Examples

```
# multiple_parameters_one.py

def sum_and_print(integers_list):
    result = 0
    for i in range(len(integers_list)):
        print(f"{i+1}. element:" , integers_list[i])
        result = result + integers_list[i]

    print(f"Sumamry of Integer List: ", result)
    return result

def sum_and_print_two(first_number, second_number):
    result = 0
    print(f"Numbers are {first_number} and {second_number}")
    result = first_number + second_number
    print(f"Summary of Two Numbers: ", result)
    return result

list_of_integers = [6, 7, 8, 9]
sum_and_print(list_of_integers)
print("**********************")
a, b = 14, 15
sum_and_print_two(a, b)
```

Output

```
1. element: 6
2. element: 7
3. element: 8
4. element: 9
Sumamry of Integer List:  30
**********************
Numbers are 14 and 15
Summary of Two Numbers:  29
```

- We can use *args for list type variables.

*Multiple Parameters with *args*

```
# multiple_parameters_two.py

def args_function(*numbers):
    print("*args seems like: ", numbers)
    print("*args type is: ", type(numbers))
    for i in numbers:
        print(i)

def sum_and_print(*numbers):
    result = 0
    print("*args seems like: ", numbers)
    print("*args type is: ", type(numbers))
    enumerated_args = enumerate(numbers, 1)
    for count, item in enumerated_args:
        print(f"{count}. element: {item}")
        result = result + item
    print("Summary of Integers: ", result)

list_of_integers = 6, 7, 8, 9
args_function(list_of_integers)
print("**********************")
sum_and_print(6, 7, 8, 9)
```

Output

```
*args seems like:  ((6, 7, 8, 9),)
*args type is:  <class 'tuple'>
(6, 7, 8, 9)
**********************
*args seems like:  (6, 7, 8, 9)
*args type is:  <class 'tuple'>
1. element: 6
2. element: 7
3. element: 8
4. element: 9
Summary of Integers:  30
```

- We can use **kwargs for dictionary type variables.

*Multiple Parameters with **kwargs*

```
# multiple_parameters_three.py

def sum_and_print(**numbers):
    print("**kwargs seems like: ", numbers)
    print("**kwargs type is: ", type(numbers))
    result = 0
    for key in numbers:
        print(f"{key}:{numbers[key]}")
        result = result + numbers[key]
    print("Sumamry of Numbers: ", result)
    return result

sum_and_print(a=6, b=7, c=8, d= 9)
# first_dict = {
#     "a": 2,
#     "b": 3,
#     "c": 4
# }

# second_dict = {
#     "c": 6,
#     "d": 41,
#     "e": 19
# }
# sum_and_print(first_dict, second_dict)
```

Output

```
**kwargs seems like:  {'a': 6, 'b': 7, 'c': 8, 'd': 9}
**kwargs type is:  <class 'dict'>
a:6
b:7
c:8
d:9
Sumamry of Numbers:  30
```

- When defining a function, every parameter has its own order.

*Defining Function with * Operator*

```
# Correct Usage of Functions

def function_one(first_p, second_p, *args, **kwargs):
    # Correct Usage
    pass

def function_two(first_p, second_p, **kwargs, *args):
    # Correct Usage
    pass
```

Design Patterns

- Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a sibstanial period of time.
- Why we need?
- Gang of Four
- Common Python Patterns.
- Common Usages
- Our Examples
- Some Principles to Follow
    - Never create things that shouldn't be created: Your classes should follow the single responsibility principle; the idea that a class should only do one thing.

    - Keep constructors simple: Constructors should be kept simple. The constructor of a class shouldn't be doing any work — that is, they shouldn't be doing anything other than checking for null, creating creatables, and storing dependencies for later use. They shouldn't include any coding logic.
    - Don't assume anything about the implementation: Interfaces are, of course, useless without an implementation. However, you, as a developer, should never make any assumptions about what that implementation is.

Behavioral Patterns

Iterators

- What is iterators?
- Examples

Observers

- An object, called the Subject (Observable), manages a list of dependents, called Observers, and notifies them automatically of any internal state changes by calling one of their methods.
- The Observer pattern follows the publish/subscribe concept. A subscriber, subscribes to a publisher. The publisher then notifies the subscribers when necessary.
- The observer stores state that should be consistent with the subject. The observer only needs to store what is necessary for its own purposes.
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

Observer Example

```python
# observer_1.py

"""
Define a one-to-many dependency between objects so that when one object
changes state, all its dependents are notified and updatedautomatically.
"""

import abc


class Subject:
    """
    Know its observers. Any number of Observer objects may observe a
    subject.
    Send a notification to its observers when its state changes.
    """

    def __init__(self):
        self._observers = set()
        self._subject_state = None

    def attach(self, observer):
        observer._subject = self
        self._observers.add(observer)

    def detach(self, observer):
        observer._subject = None
        self._observers.discard(observer)

    def _notify(self):
        for observer in self._observers:
            observer.update(self._subject_state)

    @property
    def subject_state(self):
        return self._subject_state

    @subject_state.setter
    def subject_state(self, arg):
        self._subject_state = arg
        self._notify()


class Observer(metaclass=abc.ABCMeta):
    """
    Define an updating interface for objects that should be notified of
    changes in a subject.
    """

    def __init__(self):
        self._subject = None
        self._observer_state = None

    @abc.abstractmethod
    def update(self, arg):
        pass


class ConcreteObserver(Observer):
    """
    Implement the Observer updating interface to keep its state
    consistent with the subject's.
    Store state that should stay consistent with the subject's.
    """

    def update(self, arg):
        self._observer_state = arg
        # ...


def main():
    subject = Subject()
    concrete_observer = ConcreteObserver()
    subject.attach(concrete_observer)
    subject.subject_state = 123


if __name__ == "__main__":
    main()
```

Output

Structural Patterns

Decorators

- Decorators provide simple syntax for calling high order functions.
- High Order Function: In mathematics and computer science, a high-order function is a function that does at least one of the following:
    - take one or more functions as arguments (i.e procedural parameters),
    - returns a function as its result.
- All the other functions are first-order functions.
- By definition, a decorator is a function that takes another function and extends the behaviour of the latter function without explicitly modfying it. And we use decorator a lot :D
- I wrote decorators.py and added two decorators in it. Then I used these decorators in another file.

Decorators

```
# decorators.py
''' DocString'''

from functools import wraps
from time import time

def exception_handler():
    ''' Exception Handler Decorator. If any exception occurs, don't stop just print exception. '''

    def decorator_exception_handler(func):
        @wraps(func)
        def func_exception_handler(*args, **kwargs):

            try:
                return func(*args, **kwargs)
            except Exception as exception: # pylint: disable=broad-except
                print("Exception Occured! -> ", str(exception))

        return func_exception_handler
    return decorator_exception_handler

def measure_response_time():
    ''' Measue response time decorator.'''

    def decorator_measure_response_time(func):
        wraps(func)
        def func_measue_respone_time(*args, **kwargs):
            start_time = time()
            result = func(*args, **kwargs)
            end_time = time()
            elapsed_time = (end_time - start_time)

            print(f"Elapsed Time: {elapsed_time} seconds")

            return result
        return func_measue_respone_time
    return decorator_measure_response_time
```

Usage of Decorators

```
''' Example python file'''

from decorators import exception_handler
from decorators import measure_response_time
import time

@exception_handler()
def example_function(a):
    ''' DocString '''
    return 66/a

example_function(317)
example_function(0)
example_function("any_type_of_str")

@measure_response_time()
def example_function_2(last_words):
    ''' DocString '''
    time.sleep(3)
    print(last_words)
    return last_words

example_function_2("I'm only sleeping")
```

Output

```
Exception Occured! ->  division by zero
Exception Occured! ->  unsupported operand type(s) for /: 'int' and 'str'
I'm only sleeping
Elapsed Time: 3.005300998687744 seconds
```

Facade

- According to GoF, Facade design pattern is defind as: Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.
- Facade can be recognized in a class that has a simple interface, but delegates most of the work to other classes. Usually, facades manage the full life cycle of objects they use.
- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

Example Facade Pattern for Complex Coffee Machine

```
# fasaad design pattern py file

# facade class
class ComplexCoffeMachine():
    """ Complex Coffee Machine Facade class. It calls other complex classes. """

    def __init__(self):
        """ initialization """

        self.grinder = _Grinder()
        self.kettle = _Kettle()
        self.pressure = _PressureUnit()
        self.distiller = _Distiller()
        self.water_tank = 0
        self.coffee_tank = 0

    def add_water(self, amount_of_water):
        """ Add Water to water tank. """
        self.water_tank = self.water_tank + amount_of_water

    def get_amount_of_water(self):
        """ Get amount of water of water tank. """
        return self.water_tank

    def add_coffee_beans(self, amount_of_coffee):
        """ Add coffee beans to coffee tank. """
        self.coffee_tank = self.coffee_tank + amount_of_coffee

    def get_amount_coffee_beans(self):
        """ Get amount of coffee beans. """
        return self.coffee_tank

    def make_filter_coffee(self, cup_count):
        """ Make Filter Coffee """
```

```python
        """ Make Filter Coffee """
        water = cup_count * 250
        coffee = cup_count * 12
        self.grinder.change_coffee_bean_size(4)
        ground_coffee = self.grinder.grind_coffee(coffee)
        temp_water = self.kettle.boil_water(water, 94)
        self.distiller.distill(ground_coffee, temp_water)

        print(f"Filter Coffee Is ready for {cup_count} cups.")


class _Grinder():
    """ Grinder Unit Class """

    def __init__(self):
        """ initialization """
        self.turn_per_second = 30
        self.coffe_bean_size = 4

    def change_turn_per_second(self, turn_count):
        """ Change the blades turn count. max 10-30 """
        self.turn_per_second = turn_count

    def change_coffee_bean_size(self, bean_size):
        """ Change the coffee bean size. 1-10 """
        self.coffe_bean_size = bean_size

    def grind_coffee(self, amount_of_coffee):
        """ Start the coffee grinder. 30gr coffee bean -> 15sec. """
        time = (amount_of_coffee * self.coffe_bean_size / self.turn_per_second) * 1.8
        print(f"Coffee Grind Operation Completed! It tooks {time} seconds")

        return amount_of_coffee


class _Kettle():
    """ Kettle Unit Class """

    def __init__(self):
        """ initialization """
        self.amount_of_water = 0

    def boil_water(self, amount_of_water, tempature):
        """ Boil water to desired tempature. """
        time = (tempature * amount_of_water) / 450
        print(f"Water is ready! It tooks {time} seconds. ")
        return (amount_of_water * 98) / 100


class _PressureUnit():
    """ Pressure Unit Class """

    def __init__(self):
        """ initialization """


class _Distiller():
    """ Pressure Unit Class """

    def __init__(self):
        pass

    def distill(self, coffee, water):
        """ Distill coffee """
        time = 150 + (coffee * water) / 200
        print(f"Distill Operation Completed! It tooks {time} seconds.")


def main():
    """ Main Function Of Facade. It is client of Facade Pattern. """

    hikmet_bey = ComplexCoffeMachine()
    hikmet_bey.add_coffee_beans(1000)
    hikmet_bey.add_water(2000)
    hikmet_bey.make_filter_coffee(3)

if __name__ == "__main__":
    main()
```

Output

```
Coffee Grind Operation Completed! It tooks 8.64 seconds
Water is ready! It tooks 156.66666666666666 seconds.
Distill Operation Completed! It tooks 282.3 seconds.
Filter Coffee Is ready for 3 cups.
```

Creational Patterns

Singleton

- Why singleton?
- Give examples.

Dependency Injection

- Dependency Injection is "D" for SOLID Principles.
- Dependency Injection is an object-oriented software design principle that creates less fragile code and makes writing tests easier by decoupling lower-level classes from higher-level classes.
- Minimalistic dependencies — As the dependencies are clearly defined, easier to eliminate/reduce unnecessary dependencies.
- Code with reduced module complexity, increased module reusability.
- Instantiating mock objects and integrating with class definitions is easier. (Easy To Test :D)
- Flexibility of configurable components.

Without Dependency Injection Principles

```
from datetime import datetime

class MessageFormatter:
    def success(self, message):
        now = datetime.now().strftime("%d-%m-%Y %H:%M:%S")
        return f"[{now}] -> {message}"


class MessageWriter:
    def __init__(self):
        self.message_formatter = MessageFormatter()

    def write(self, message):
        print(self.message_formatter.success(message))


def main():
    message_writer = MessageWriter()
    message_writer.write("Joy Division Concert at Concert Hall")


if __name__ == "__main__":
    main()
```

Output

```
[13-06-2021 20:39:25] -> Joy Division Concert at Concert Hall
```

With Dependency Injection Principles

```
from datetime import datetime

class MessageFormatter:
    def success(self, message):
        now = datetime.now().strftime("%d-%m-%Y %H:%M:%S")
        return f"[{now}] -> {message}"


class MessageWriter:
    def __init__(self, message_formatter):
        self.message_formatter = message_formatter

    def write(self, message):
        print(self.message_formatter.success(message))


def main():
    message_formatter = MessageFormatter()
    message_writer = MessageWriter(message_formatter)
    message_writer.write("Joy Division Concert at Concert Hall")


if __name__ == "__main__":
    main()
```

Output

```
[13-06-2021 20:39:25] -> Joy Division Concert at Concert Hall
```

- Benefits of Dependency Injection
  - Maintainability: Probably the main benefit of dependency injection is maintainability. If your classes are loosely coupled and follow the single responsibility principle — the natural result of using DI — then your code will be easier to maintain.
  - Testability: Along the same lines as maintainability is testability. Code that is easy to test is tested more often. More testing means higher quality.
  - Readability: Code that uses DI is more straightforward. It follows the single responsibility principle and thus results in smaller, more compact, and to-the-point classes.
  - Flexibility: Loosely coupled code — yet again, the result of using dependency injection — is more flexible and usable in different ways. Small classes that do one thing can more easily be reassembled and reused in different situations
  - Extensibility: Code that uses dependency injection results in a more extendable class structure. By relying on abstractions instead of implementations, code can easily vary a given implementation.
  - Team-ability: If you are on a team and that team needs to work together on a project (when is that not true?), then dependency injection will facilitate team development.

Context Managers

- Context managers allow you to allocate and release resources precisely when you want to. The most widely used example of context managers is the "with" statement.
- Suppose you have two related operations which you'd like to execute as a pair, with a block of code in between. Context managers allow you to do specifically that.

Context Manager 101

```
""" Context Manager Example"""

with open('red_hot_chili_peppers.txt', 'r') as opened_file:
    data = opened_file.read()

splitted_data = data.split()
print(splitted_data[0], splitted_data[1])


# Write with 'with'
with open('new_file.txt', 'w') as opened_file:
    opened_file.write("This is a new file! ")

# Same as Write with 'with'
file = open('new_file.txt', 'w')
try:
    file.write("Different Write Method")
finally:
    file.close()
```

- We can implement Context Manager as class.
- 1. The with statement stores the *exit* method of the File class.
- 2. It calls the *enter* method of the File class.
- 3. The *enter* method opens the file and returns it.
- 4. The opened file handle is passed to opened_file.
- 5. We write to the file using .write().
- 6. The with statement calls the stored *exit* method.
- 7. The *exit* method closes the file.

Implementing a Context Manager as a Class

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        self.file_obj.close()

with File('new_demo.txt', 'w') as opened_file:
    opened_file.write('We created new class and used with Context Manager!')
```

- Handling Exception: Type, value and traceback arguments of the *exit* method. Between the 4th and 6th step, if an exception occurs, Python passes the type, value and traceback of the exception to the *exit* method. It allows the *exit* method to decide how to close the file and if any further steps are required.
- 1. It passes the type, value and traceback of the error to the *exit* method.
- 2. It allows the *exit* method to handle the exception.
- 3. If *exit* returns True then the exception was gracefully handled.
- 4. If anything other than True is returned by the *exit* method then the exception is raised by the with statement.
- Our *exit* method returned True, therefore no exception was raised by the with statement.

Exception Handling in Context Manager

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        print("Exception has been handled")
        self.file_obj.close()
        return True

with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function()

# Output: Exception has been handled
```

- Context manager can be used as generator.
- 1. Python encounters the yield keyword. Due to this it creates a generator instead of a normal function.
- 2. Due to the decoration, contextmanager is called with the function name (open_file) as its argument.
- 3. The contextmanager decorator returns the generator wrapped by the GeneratorContextManager object.
- 4. The GeneratorContextManager is assigned to the open_file function. Therefore, when we later call the open_file function, we are actually calling the GeneratorContextManager object.

Implement Context Manager as Decorator

```
from contextlib import contextmanager

@contextmanager
def open_file(name):
    f = open(name, 'w')
    try:
        yield f
    finally:
        f.close()
```

Multithread

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.
- A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.
- We have a function that send request to 3rd Party API and it may be take a few seconds.
- Let's write a program without multithread and convert to multithread and compare programs run times.

Program Without Multithread

```
# program without multithread

import re
import json
import time
from datetime import datetime

def prepare_report_links(build:dict):
    """ export report links from build """
    build_task_log = dict()
    build_task_log["build_id"] = build["build_id"]
    build_task_log["jobs"] = list()

    for job in build.get("jobs", []):
        job_dict = dict()
        job_dict["name"] = job.get("name")
```

```python
        job_dict["tasks"] = list()

        for task in job.get("tasks", []):
            task_dict = dict()
            task_dict["name"] = task.get("name", "")
            task_dict["report_link"] = task.get("report_link", "")
            job_dict["tasks"].append(task_dict)

        build_task_log["jobs"].append(job_dict)

    return build_task_log

def get_data_from_link(link):
    """ Mock Request to 3rd Party API Part. """

    today = datetime.today().strftime("%Y-%M-%d %H:%M:%S")
    number = int(link.split("/")[-1])
    string = f'''
    [f{today}] [sarpine_LOG] copy input aposto/{number}{number}/{number}{number} -> aposto/{number}
    [f{today}] [sarpine_LOG] copy output aposto/{number+10}{number+10}/{number+10}{number+10} -> aposto/{number+10}
    [f{today}] [sarpine_LOG] copy input aposto/{number+20}{number+20}/{number+20}{number+20} -> aposto/{number+20}
    '''
    time.sleep(2)
    return string

def parse_link_content(link_content):
    """ Parse Link Conntent from 3rd Party API """

    result = dict()
    result["dep_in"] = []
    result["dep_out"] = []

    matches_out = re.findall(r"copy output .* -> .*", link_content)
    matches_in = re.findall(r"copy input .* -> .*", link_content)

    for dep_in in matches_in:
        result["dep_in"].append(dep_in.split(" ")[2])
    for dep_out in matches_out:
        result["dep_out"].append(dep_out.split(" ")[2])

    return result

def prepare_document(build:dict):
    """ Create new document from raw data. """

    build_id = build.get("build_id")
    job_tasks_with_report_links = prepare_report_links(build)
    result = dict()
    result["build_id"] = build_id
    result["deps"] = list()

    for job in job_tasks_with_report_links.get("jobs", []):
        job_dict = dict()
        job_dict["job_name"] = job.get("name")
        job_dict["tasks"] = list()
        for task in job.get("tasks", []):
            task_dict = dict()
            task_dict["task_name"] = task["name"]
            report_link = task.get("report_link")
            link_content = get_data_from_link(report_link)
            task_content_output = parse_link_content(link_content)
            task_dict["dep_in"] = task_content_output["dep_in"]
            task_dict["dep_out"] = task_content_output["dep_out"]
            job_dict["tasks"].append(task_dict)
        result["deps"].append(job_dict)

    return result


def main():
    """ Main Function. """
    data = ""
    with open("build.json", "r") as file_2:
        data = json.load(file_2)
    prepare_document(data)
    # print(prepare_document(data))

if __name__ == "__main__":
    start_time = time.time()
    main()
    print(f"It takes {time.time() - start_time} seconds.")
```

Output

```
It takes 32.048429012298584 seconds
```

- We separate functions and data because program runs with threads. We must know what each threads do on their lifetime.

Same Program With Multithread

```
# program with multithread

import re
import json
import time
import concurrent.futures
from datetime import datetime


def separate_data(build):
    """ Separate build data to run multithread. """

    part_of_build = list()
    for job in build.get("jobs"):
        for task in job.get("tasks"):
            part_of_build.append((job["name"], task["name"], task["report_link"]))
    return part_of_build, build["build_id"]

def parse_link_content(link_content):
    """ Parse Link Conntent from 3rd Party API """

    result = dict()
    result["dep_in"] = []
    result["dep_out"] = []

    matches_out = re.findall(r"copy output .* -> .*", link_content)
    matches_in = re.findall(r"copy input .* -> .*", link_content)

    for dep_in in matches_in:
        result["dep_in"].append(dep_in.split(" ")[2])
    for dep_out in matches_out:
        result["dep_out"].append(dep_out.split(" ")[2])

    return result

def get_data_from_link(link):
    """ Mock Request to 3rd Party API Part. """

    today = datetime.today().strftime("%Y-%M-%d %H:%M:%S")
    number = int(link.split("/")[-1])
    string = f'''
    [f{today}] [sarpine_LOG] copy input aposto/{number}{number}/{number}{number} -> aposto/{number}
    [f{today}] [sarpine_LOG] copy output aposto/{number+10}{number+10}/{number+10}{number+10} -> aposto/{number+10}
    [f{today}] [sarpine_LOG] copy input aposto/{number+20}{number+20}/{number+20}{number+20} -> aposto/{number+20}
    '''
    time.sleep(2)
    return string

def run_separated_build_data_with_threads(separated_data, build_id):
    # Do Multithread Operation

    max_workers = 10
    print(f"Multithreading With {max_workers} workers.")
    with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
        results = executor.map(create_deps_data_from_task, separated_data)

    # group by job name
    job_group = dict()

    for result in results:
        job_name = result[1]
        output = result[0]
        job_tasks_list = job_group.get(job_name, [])
        job_tasks_list.append(output)
        job_group[job_name] = job_tasks_list

    # create result data from grouped multithread data
    result_data = dict()
    result_data["build_id"] = build_id
    deps = list()
    for i in job_group:
        temp_dict = dict()
        temp_dict["job_name"] = f"{i}"
        temp_dict["tasks"] = job_group[i]
        deps.append(temp_dict)
    result_data["deps"] = deps

    return result_data
```

```
def create_deps_data_from_task(item):
    """ Create Data to use in collection. """

    job_name, task, link = item
    link_content = get_data_from_link(link)
    content_output = parse_link_content(link_content)
    task_dict = dict()
    task_dict["task_name"] = task
    task_dict["dep_in"] = content_output.get("dep_in", [])
    task_dict["dep_out"] = content_output.get("dep_out", [])

    return task_dict, job_name


def main():
    """ Main Function. """

    data = ""
    with open("build.json", "r") as file_2:
        data = json.load(file_2)
    separated_build_data, build_name = separate_data(data)
    collection_data = run_separated_build_data_with_threads(separated_build_data, build_name)
    # print(collection_data)

if __name__ == "__main__":
    start_time = time.time()
    main()
    print(f"It takes {time.time() - start_time} seconds.")
```

Output

```
It takes 4.031013011932373 seconds
```

- Disadvantages of Multithreading:
  - On a single processor system, multithreading won't hit the speed of computation. The performance may downgrade due to the overhead of managing threads.
  - Multithreading increases the complexity of the program, thus also making it difficult to debug.
  - It raises the possibility of potential deadlocks.
  - It may cause starvation when a thread doesn't get regular access to shared resources. The application would then fail to resume its work.

Garbage Collector/Collection

- The Garbage Collector is keeping track of all objects in memory. A new object starts its life in the first generation of the garbage collector. If Python executes a garbage collection process on a generation and an object survives, it moves up into second, older generation.
- How Python implements garbage collection There are two aspects to memory management and garbage collection in CPython:
  - Reference counting: At a very basic level, a Python object's reference count is incremented whenever the object is referenced, and it's decremented when an object is dereferenced. If an object's reference count is 0, the memory for the object is deallocated.
  - Generational garbage collection: There are two key concepts to understand with the generational garbage collector.
    - The first concept is that of a generation.
    - The second key concept is the threshold.
  - The garbage collector is keeping track of all objects in memory. A new object starts its life in the first generation of the garbage collector. If Python executes a garbage collection process on a generation and an object survives, it moves up

    into a second, older generation. The Python garbage collector has three generations in total, and an object moves into an older generation whenever it survives a garbage collection process on its current generation.
  - For each generation, the garbage collector module has a threshold number of objects. If the number of objects exceeds that threshold, the garbage collector will trigger a collection process. For any objects that survive that process, they're moved into an older generation.
  - Unlike the reference counting mechanism, you may change the behavior of the generational garbage collector in your Python program. This includes changing the thresholds for triggering a garbage collection process in your code. Additionally, you can manually trigger a garbage collection process, or disable the garbage collection process altogether.

GC(Garbage Collection) Module Get Methods.

```
# gc_module.py

import gc

# check the configured thresholds of your garbage collector
print("Threshoold: ", gc.get_threshold())
# (youngest_generation, next_generation, oldest_generation)
print("Number Of Objects in Generation: ", gc.get_count())
# (youngest_generation, next_generation, oldest_generation)

print(gc.get_count())
print(gc.collect())
print(gc.get_count())
```

Output

```
Threshoold:  (700, 10, 10)
Number Of Objects in Generation:  (544, 4, 1)
(544, 4, 1)
0
(1, 0, 0)
```

- Running a garbage collection process cleans up a huge amount of objects—there are 544 objects in the first generation and 5 (4, 1) more in the older generations.
- We can change thresholds.

GC(Garbage Collection) Module Set Methods.

```
# gc_module_2.py

import gc

# In the example above, we increase each of our thresholds from their defaults.
# Increasing the threshold will reduce the frequency at which the garbage collector runs.
# This will be less computationally expensive in your program at the expense of keeping dead objects around longer.
print("Threshoold: ", gc.get_threshold())
# (youngest_generation, next_generation, oldest_generation)
gc.set_threshold(1000, 20, 30)
print("Threshoold: ", gc.get_threshold())
```

Output

```
Threshoold:  (700, 10, 10)
Threshoold:  (1000, 20, 30)
```

- In the example above, we increase each of our thresholds from their defaults. Increasing the threshold will reduce the frequency at which the garbage collector runs. This will be less computationally expensive in your program at the expense of keeping dead objects around longer.

NOTE

> Don’t change garbage collector behavior: As a general rule, you probably shouldn’t think about Python’s garbage collection too much. One of the key benefits of Python is it enables developer productivity. Part of the reason for this is because it’s a high-level language that handles a number of low-level details for the developer.

- EXAMPLE: The Instagram (Django) team disabled the garbage collector module by setting the thresholds for all generations to zero. This change led to their web applications running 10% more efficiently.

Performance Improvement

- Why we need?
- Performance Tips and Usage.
- Maybe game or demo examples. (DEMO CULTURE)
- Common Usages
- Our Examples

Street Coding

- What is Street Coding?
- Why we need them / Advantages
- Common Usages
- Our Examples