Imperative Programming

Imperative programming is a programming paradigm in which programs are written in a sequence of instructions where the programs' state changes from each resultant instruction. The order of execution of these instructions is critical as each instruction directly changes the programs' state. The code describes how a program operates, the instructions the computer must make to accomplish a certain goal e.g. algorithms.

Object Oriented Programming

Object Oriented programming is a programming paradigm where everything is an object, objects are an instance of a class and each object has its own set of attributes and methods that only itself can access and change. Objects interact with each other and themselves to accomplish a certain goal. Object oriented programming has numerous features such as:

- Encapsulation - Allows data to be hidden from the outside world (inaccessible apart from by itself)
- Inheritance - Classes can inherit data and methods from a parent class
- Polymorphism - Allows inheritance to occur while also allowing child classes to change an inherited method to perform a variation of its originally intended use

Comparison of implementations

I chose to implement both the Imperative and Object Oriented style in Python 3, as python naturally supports both of these programming paradigms. Both programs begin by establishing a valid command from the users input.

Imperative

The program is started by calling the main() function which initiates the program loop in the form of a while loop. The user is asked which command they want to use, if the command is not valid the program will loop and ask them again until a valid command is entered which will then run the break instruction to break out of the inner loop.

```
def main():
    while True:
        todo_type = ""
        while True:
            print("Enter Task or Event   |   quit, remove and view commands
are also available")
            todo_type = lower(strip(input()))
            if todo_type in ["task", "event", "quit", "remove", "view"]:
                break
```

Object Oriented

Todo loop is initiated by the Todo().start() object method call, this creates a todo object and runs the start method. The program begins as in the imperative by asking the user to input a command and the program then verifies that command, displaying an invalid command error message and looping until a valid command is entered. At which point the getattr() function finds the function with the same name as the input command. This function is then given to the thread which initialises a thread object which executes the given function parameter.

```
    def start(self):
        while self.quit != True:
            threads = []
            print("Enter Task or Event   |   quit, remove and view commands
are also available")
            cmd = input().strip().lower()

            if cmd in self.cmds_allowed:

                myfunc = getattr(self, cmd

                t = Thread(target=myfunc)
                threads.append(t)

                self.handle_threading(threads)

            else:
                print("{} is not a valid command\n".format(cmd))
```

Both programs then parse the input in the exact same way, taking input and parsing, the only difference is once all needed data is acquired the imperative approach simply adds it all to a string and then puts it in the queue. Alternatively the Object Oriented program creates an object with the acquired data and then puts it in a queue. The object oriented approach is much simpler as all the data can be accessed by a simple .<attribute> where <attribute> represents the attribute you want, e.g. Event.title. in the imperative if you wanted to get just the title of the event from the string in the queue you would need to manually parse the string which would be more difficult.

Imperative

```
            if todo_type == "task":
                print("\nEnter task duration")
                duration = input()
                print("\nEnter people assigned to the task separated by a
comma \",\"")
                people = input()
                output = "\nTitle : " + title + "\nDate : " + date +
"\nTime : " + time + "\nDuration : " + duration + "\nPeople : " + people +
"\n"
                queue.put(output)
            else:
                print("\nEnter event location")
                location = input()
                output = "\nTitle : " + title + "\nDate : " + date +
"\nTime : " + time + "\nLocation : " + location + "\n"
                queue.put(output)
```

Object Oriented

```
    def task(self):
        title, date, time = self.get_details("task")
        print("\nEnter task duration")
        duration = input().strip()
        print("\nEnter people assigned to the task separated by a comma
\",\"")
        people = input().strip()

        task = Task(title, date, time, duration, people)
        self.add(task)

    def event(self):
        title, date, time = self.get_details("event")
        print("\nEnter event location")
        location = input().strip()

        event = Event(title, date, time, location)
        self.add(event)
```

After both programs loop and repeat the instructions explained above unless an alternative command is entered. If the user enters ‘view’ both programs will display the reminder to the terminal window in the exact same way.

Imperative

The control of flow is determined mainly by if statements, when the user enters view
The elif statement below evaluates to True and the code is executed. If the queue is empty an error message is displayed, else the str from the queue is printed.

```
        elif todo_type == "view":
            if queue.empty():
                print("\nNo reminders available to view\n")
            else:
                while not queue.empty():
                    print(queue.get())
```

Object Oriented

The control of flow is determined mainly by methods, when the user enters ‘view’ the getattr() function finds the view() function in the Todo class and the code is executed. As in the imperative version if the queue is empty an error message is displayed, else, the object retrieved from the queue is printed. To allow the printing of an object the object must have a __str__ method, this allows to print() method to return a string representation of the given object.

```
    def view(self):
        q = self.queue
        if q.empty():
            print("\nNo reminders available to view\n")
        else:
            while not q.empty():
                print(q.get())
```

The remaining commands remove and quit operate in essentially the same way across both implementations.

Conclusion

In my opinion, both the imperative and object oriented programming paradigm have their own advantages and disadvantages.

- The imperative approach of this Todo program is significantly less code than the object oriented approach if u minus the additional helper functions implemented.
- Thought the object oriented code is much longer, it is much easier to implement new functionality into the program, you could add a new todo class variant with ease but in the imperative approach eventually the code will become too long and complex that the addition of new functionality will be very difficult. This becomes exponentially more difficult in a large scale project setting with multiple people working on the same code base, while a whole team could work on a new class definition without touching any of the already existing code which they shouldn't modify unless needed
- classes can inherit from parent classes reducing the need to rewriting the same thing over and over and increasing code reusability
- Encapsulation, although not available in python, can be used in an object oriented paradigm to hide data, which you cannot do in an imperative approach

In summary, an imperative paradigm has the advantage of being quick and simple to write and possibly easier to read up to a certain point. An object oriented paradigm has the advantage of quick and easy data access through objects reducing parsing and features such as Encapsulation, Inheritance and Polymorphism to increase security and code reusability.

Overall I think any small project or algorithm should be designed with an imperative approach over object oriented programming, and any medium to large scale project should implement an object oriented approach over imperative as an object oriented approach allows the code to scale well without creating too much additional complexity and is more reusable and maintainable.