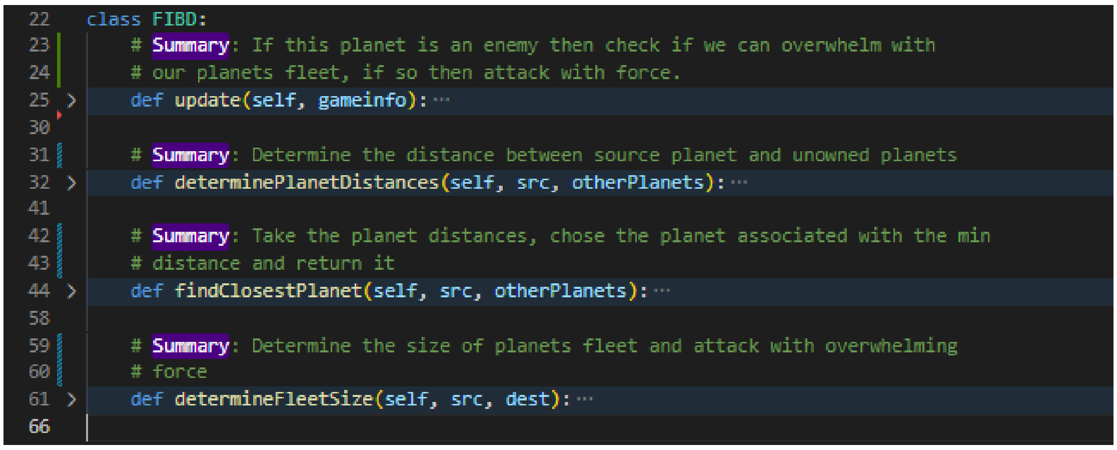
# Tasks Undertaken

## Tactical Analysis Bot

My bot’s overall strategy was to take control of the closest available planets with the most resources. This resulting in the bot valuing strong control over dense clusters of planets as-well as rapid initial growth (not wasting any time on more advanced inputs such as scouting).

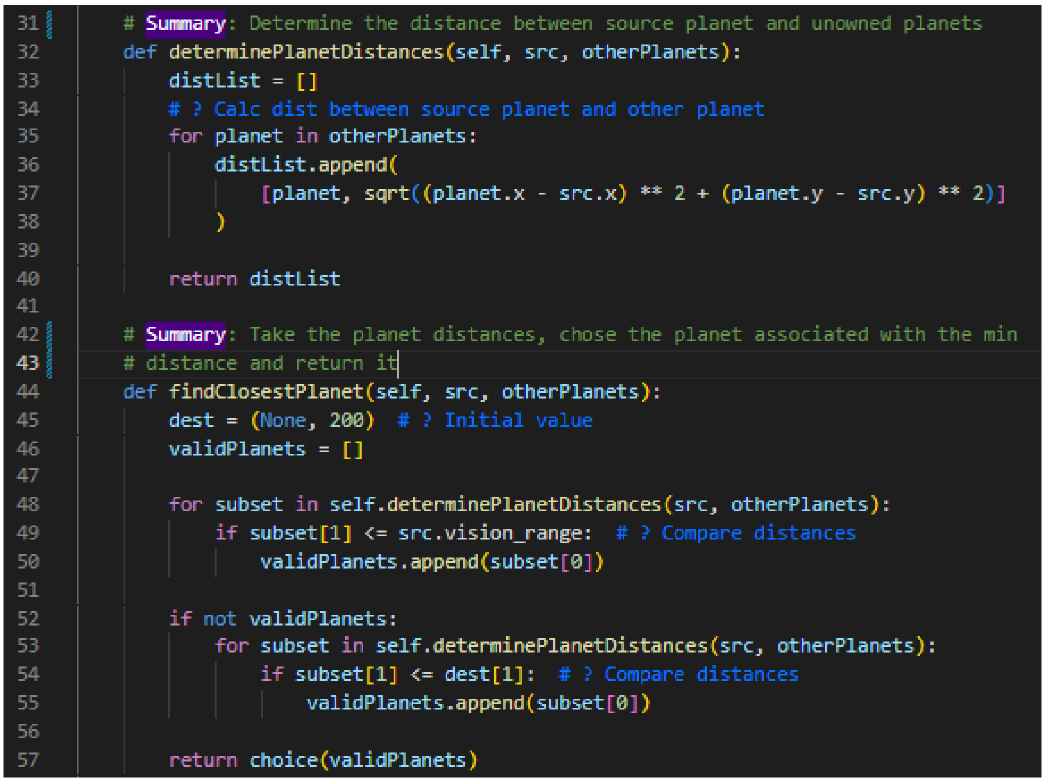
The bot had access to a game facade (player-relevant, publically known frame-data relative to the current gamestate) and from this facade, the bot may scrape data relevant to its’ planets, visible opponents planets, the fleet count of planets and the growth of planets. The bot will use this information to gauge the most strategically “valuable” (valuable relative to the bots strategy) positions before making efforts to capture those positions.



```python
class FIBD:
    # Summary: If this planet is an enemy then check if we can overwhelm with
    # our planets fleet, if so then attack with force.
    def update(self, gameinfo):⋯

    # Summary: Determine the distance between source planet and unowned planets
    def determinePlanetDistances(self, src, otherPlanets):⋯

    # Summary: Take the planet distances, chose the planet associated with the min
    # distance and return it
    def findClosestPlanet(self, src, otherPlanets):⋯

    # Summary: Determine the size of planets fleet and attack with overwhelming
    # force
    def determineFleetSize(self, src, dest):⋯
```

All the functionality and gameinfo the bot (FIBD) needs in order to employ strategy

Each update cycle the bot orders it’s largest planet to send a fleet to the closest neutral or enemy planet. As FIBD employs tactical analysis to make it’s decisions, it can determine if it’s possible to take a destination planet by force or not. If this isn’t possible, the bot will attempt to prevent further growth of that planet by sending smaller fleets until it’s fleets are large enough to take the resource.



```python
    # Summary: Determine the distance between source planet and unowned planets
    def determinePlanetDistances(self, src, otherPlanets):
        distList = []
        # ? Calc dist between source planet and other planet
        for planet in otherPlanets:
            distList.append(
                [planet, sqrt((planet.x - src.x) ** 2 + (planet.y - src.y) ** 2)]
            )

        return distList

    # Summary: Take the planet distances, chose the planet associated with the min
    # distance and return it
    def findClosestPlanet(self, src, otherPlanets):
        dest = (None, 200)  # ? Initial value
        validPlanets = []

        for subset in self.determinePlanetDistances(src, otherPlanets):
            if subset[1] <= src.vision_range:  # ? Compare distances
                validPlanets.append(subset[0])

        if not validPlanets:
            for subset in self.determinePlanetDistances(src, otherPlanets):
                if subset[1] <= dest[1]:  # ? Compare distances
                    validPlanets.append(subset[0])

        return choice(validPlanets)
```

How the bot calculates the set of closest planets.

Each update tick the bot will find the unowned planets before calculating their distances from the fleet source. This data is appended to a 2-dimensional array which is passed to a comparitive function that returns the closest planet from the set of valid planets. In the case that there's multiple, equally attractive destination planets, the bot will randomly chose one.

The looping and iterative nature of these functions caused of UPS bottlenecking on larger map files. To solve this in future iterations, the AI will store already known information and modify this information relative to the current gamestate (more akin to memory) rather than build this information each update cycle.