CLASS Partition // aka Memory

```
class Partition:
    def __init__(self, maxSize):
        super().__init__()
        self.type = "Partition"
        self.maxSize = maxSize
        self.memory = [False for i in range(maxSize)]
        self.gaps = [[0,self.maxSize]]
        self.size = 0
        self.objects = []
```

> This basically represents a partition
> How i conceptualized it, is that one partition would represent the entire memory, and other instances of the partition class would serve as well... the partitions...
> As said, a partition may take another partition and put it in “self.objects”
> “self.type” identifies the type of partition
    > it’s “partition” when it’s just a regular partition, or when its the one representing the entire memory
    > Otherwise, it contains the names of the respective types of memory the instance should be representing
> “self.maxSize” is the amount of storage the partition should have / the amount of bytes it can store; it also specifies the amount of items “self.memory” should have
> “self.memory” represents the actual contents of the partition
    > how i envisioned it, each item in the array represents a byte
    > while the things in “self.objects” is represented in “self.memory”, manually manipulating things will result in a desync between the two. **DO** use the functions.
    > on initialization, “self.memory” is filled with the value “False” equal to “self.maxSize”
        > a “False” value represents an empty space that can be filled by a byte in memory / partition.
> “self.gaps” is the list of gaps in a partition
    > gaps are represented by a 2-item list, first item is the index where the gap starts and second item is the last index of the gap
> “self.size” is the amount of bytes currently in the partition, calculated by the sum of the “size” property of the items in “self.objects”.
> “self.objects” is the list where the actual objects are stored
    > it's not named “self.partitions” because it may also take in jobs/tasks.
    > For the sake of consistency and accuracy to actual partitions, “self.objects” should **NEVER** have both partitions and jobs at the same time
> **DO NOT** instantiate a Partition object for the main memory, use the child classes instead; **ONLY** used for making subpartitions.

Partition.gapCheck()

```
# The original gapcheck algorithm, found a more simpler approach
# Welp, cant make that work, imma stick with this
def gapCheck(self):
    self.gaps.clear()
    counter = 0
    gapStart = None
    gapEnd = None
    while counter < len(self.memory):
        if gapStart is None and self.memory[counter] is False:
            gapStart = counter
        elif gapEnd is None and gapStart is not None:
            if self.memory[counter] is True:
                gapEnd = counter - 1
                self.gaps.append([gapStart,gapEnd])
                gapStart = None
                gapEnd = None
            elif counter + 1 >= len(self.memory):
                gapEnd = counter
                self.gaps.append([gapStart,gapEnd])
                gapStart = None
                gapEnd = None
        counter += 1
```

> clears the current contents of “self.gaps” and does a complete recheck of existing gaps in the instance’s memory, which will be stored in “self.gaps”.
> I’ve tested this and it seems to work properly, with edge cases tested

Partition.newJob()

```
def newJob(self, name, jobSize):
    for gap in self.gaps:
        if gap[1] - gap[0] >= jobSize:
            self.objects.append(Job(name, jobSize, gap[0]))
            for i in range(gap[0],gap[0]+jobSize):
                self.memory[i] = True
            print(self.memory)
            self.gapCheck()
            return
```

> Takes name(string), jobSize(int) as an argument
> Instantiates a new job and puts it into “self.objects” **IF:**
- There’s a big enough gap that can accommodate the size of the job

> It also sets the indexes in “self.memory” where the job should be to the value “True” to represent that the space is taken

Partition.delJob()

```
def delObject(self, index):
    for i in range(self.objects[index].startLoc, self.objects[index].endLoc):
        self.memory[i] = False
    del self.objects[index]
    self.gapCheck()
```

> takes index(int) as an argument
> Deletes the job/task in the index provided. (may be changed to name selection depending on the front-end implementation)
> Also sets the indexes where the job/task is represented in “self.memory” to the value “False” to represent the space that was freed.

Partition -> Single User Contiguous

```
class SUC(Partition): # Single User Contiguous
    def __init__(self, maxSize):
        super().__init__(maxSize)
        self.type = "Single User Contiguous"
```

> Child class of “Partition”
> Takes maxSize(int) as an argument
> Works as a single partition

Partition -> Fixed Partition

```
class FP(Partition): # Fixed Partition
    def __init__(self, maxSize):
        super().__init__(maxSize)
        self.type = "Fixed Partition"
        # I don't exactly know how a computer assigns partition size
        # That's why im just gonna make 5 partitions and assign them arbitrary percentages of memory, namely [40%, 15%, 20%, 20%, 5%]
        self.objects.append(Partition(int(self.maxSize*0.40)))
        self.objects.append(Partition(int(self.maxSize*0.15)))
        self.objects.append(Partition(int(self.maxSize*0.20)))
        self.objects.append(Partition(int(self.maxSize*0.20)))
        # Had to workaround for the decimal values disposed during integer conversion
        self.objects.append(Partition(self.maxSize - (int(self.maxSize*0.40) + int(self.maxSize*0.15) + int(self.maxSize*0.20) + int(self.maxSize*0.20))))
        self.sum = 0
        for obj in self.objects:
            self.sum += obj.maxSize
        print(f"Max Memory: {self.sum}")
```

> Child class of “Partition”
> Takes maxSize(int) as an argument
> Makes 5 subpartitions taking about 40, 15, 20, 20, and 5 percent of the Partition’s max size
> “self.sum” is the amount of space all the partitions have together

Partition -> Dynamic Partitions

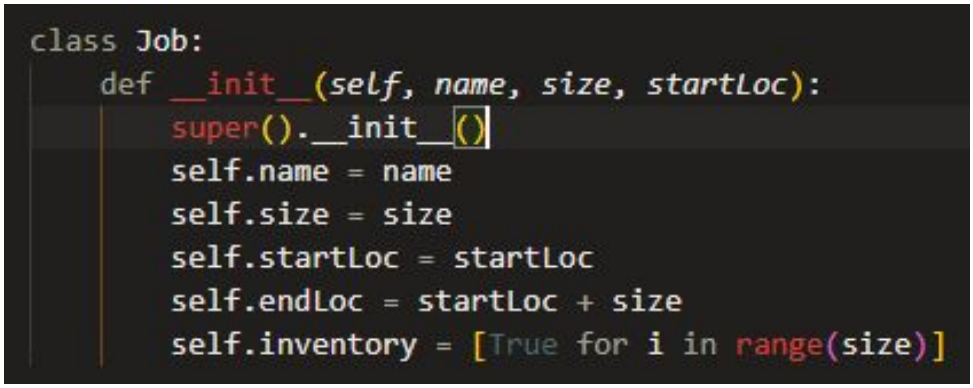
```
class DP(Partition): # Dynamic Partition
    def __init__(self, maxSize):
        super().__init__(maxSize)
        self.type = "Dynamic Partition"

class RDP(Partition): # Relocatable Dynamic Partition
    def __init__(self, maxSize):
        super().__init__(maxSize)
        self.type = "Relocatable Dynamic Partition"

    def deallocate(self):
        pass
```

> **NOT COMPLETE** as of yet**;**

CLASS Job // aka Tasks



```
class Job:
    def __init__(self, name, size, startLoc):
        super().__init__()
        self.name = name
        self.size = size
        self.startLoc = startLoc
        self.endLoc = startLoc + size
        self.inventory = [True for i in range(size)]
```

> Takes name(String), size(int), startLoc(int) as an argument
> “self.name” is just for identification purposes
> startLoc means starting location, as in what index in the partition (which is represented by an array) in which it is contained does it start at.
> endLoc, aka ending location, is automatically computed
> “self.inventory” is the representation of the size of the job...

NOW that i think about it, it's completely unnecessary and will probably be removed at the time of you reading this.