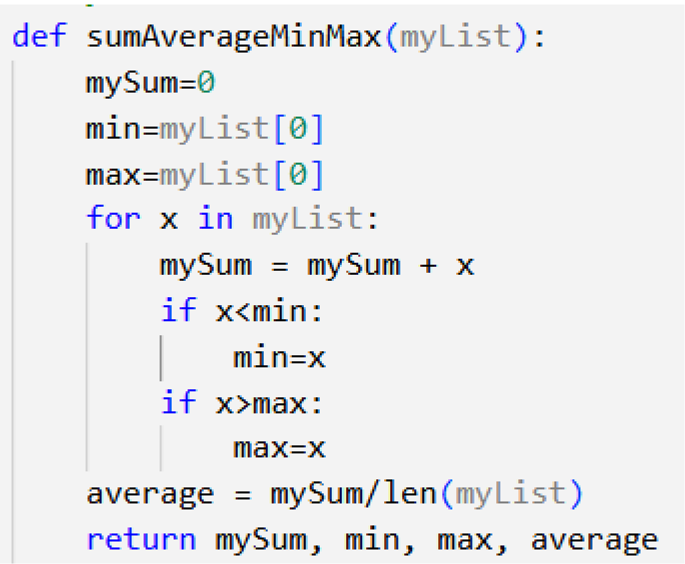
Exercises: Basic Python, Binary search and Recursion

Easy Exercise 1:

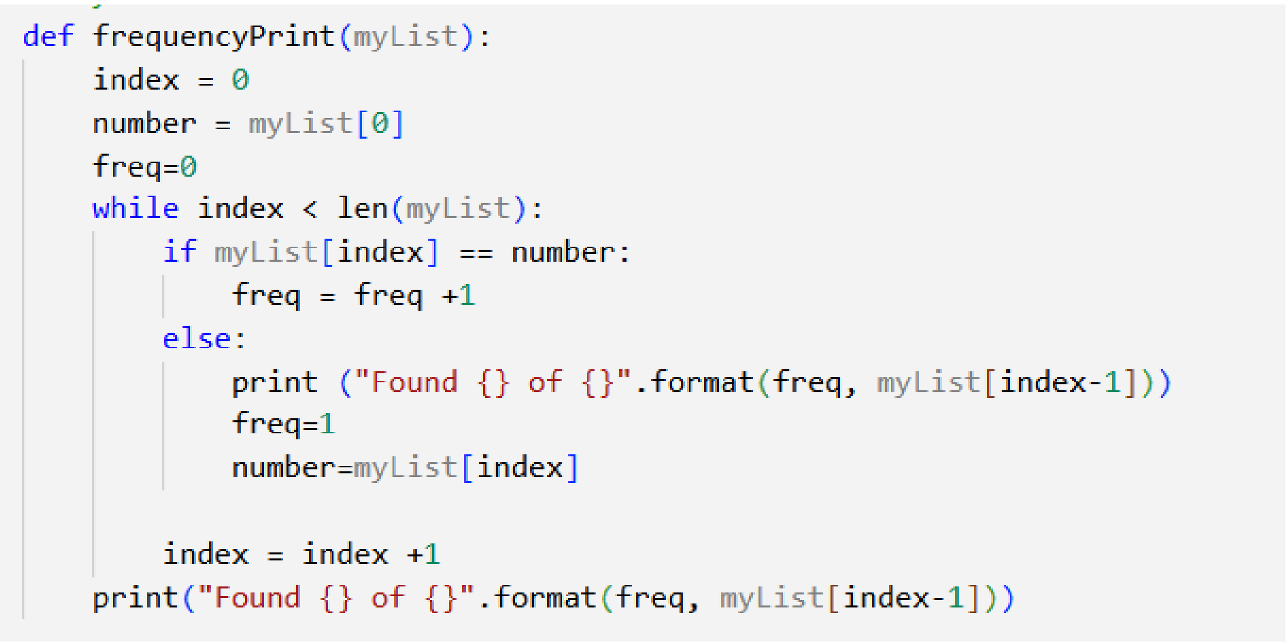
Implement a function that finds the sum, average, min and max for the numbers in an unsorted list



```
def sumAverageMinMax(myList):
    mySum=0
    min=myList[0]
    max=myList[0]
    for x in myList:
        mySum = mySum + x
        if x<min:
            min=x
        if x>max:
            max=x
    average = mySum/len(myList)
    return mySum, min, max, average
```

Easy Exercise 2:

Implement a function that finds the frequency of all numbers in a sorted list.

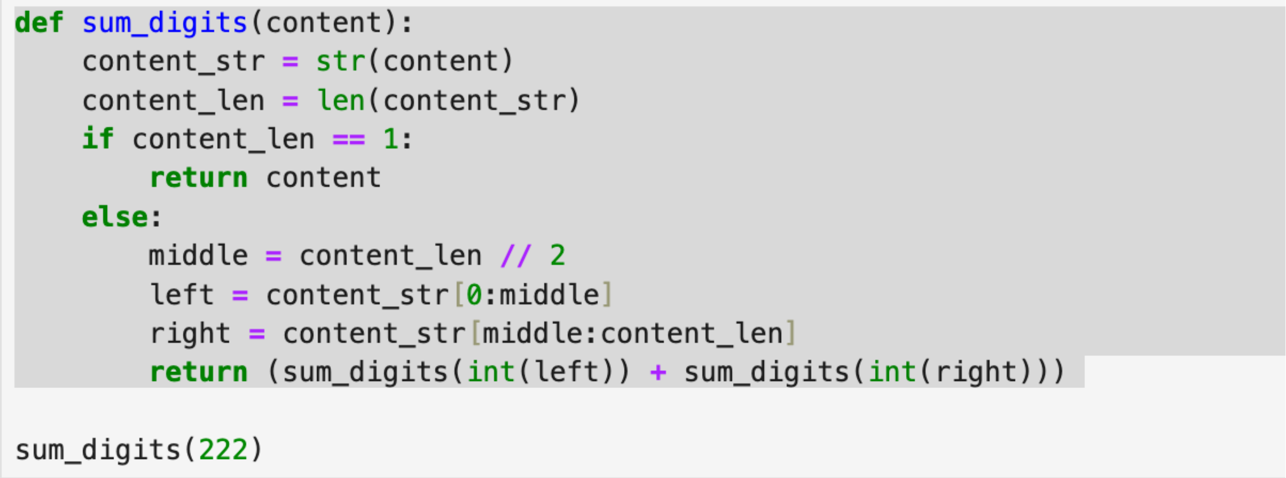


```
def frequencyPrint(myList):
    index = 0
    number = myList[0]
    freq=0
    while index < len(myList):
        if myList[index] == number:
            freq = freq +1
        else:
            print ("Found {} of {}".format(freq, myList[index-1]))
            freq=1
            number=myList[index]

        index = index +1
    print("Found {} of {}".format(freq, myList[index-1]))
```

Exercise 1: What is the sum?

Implement a recursive function that returns the sum of all digits in a positive integer

(e.g. sum_digits(222) → 2+2+2 = 6)



```
def sum_digits(content):
    content_str = str(content)
    content_len = len(content_str)
    if content_len == 1:
        return content
    else:
        middle = content_len // 2
        left = content_str[0:middle]
        right = content_str[middle:content_len]
        return (sum_digits(int(left)) + sum_digits(int(right)))

sum_digits(222)
```

1. What is the base case
2. What is the time complexity?

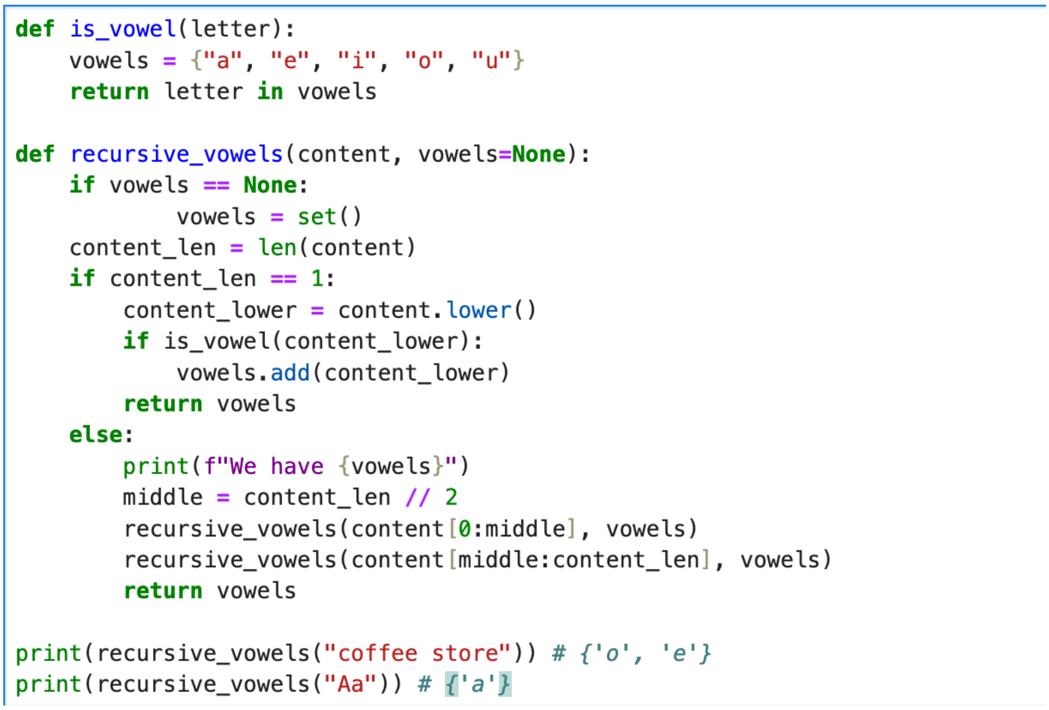
d = number of digits in the original number

We can conclude that the total number of recursive calls is bounded by a constant * d
Since asymptotic analysis requires us to remove constants, the time complexity is O(d)

Exercise 2: Unique vowels

Implement a recursive function that returns all unique vowels

(e.g. unique_vowels(”Recursion is great!”) → [’e’, ’u’, ’i’, ’o’, ’a’])



```python
def is_vowel(letter):
    vowels = {"a", "e", "i", "o", "u"}
    return letter in vowels

def recursive_vowels(content, vowels=None):
    if vowels == None:
            vowels = set()
    content_len = len(content)
    if content_len == 1:
        content_lower = content.lower()
        if is_vowel(content_lower):
            vowels.add(content_lower)
        return vowels
    else:
        print(f"We have {vowels}")
        middle = content_len // 2
        recursive_vowels(content[0:middle], vowels)
        recursive_vowels(content[middle:content_len], vowels)
        return vowels

print(recursive_vowels("coffee store")) # {'o', 'e'}
print(recursive_vowels("Aa")) # {'a'}
```

1. What is the base case

   When content_len == 1 (the string is one character)

2. What is the time complexity?

   O(n log (n)): There are log n levels of calls and we do n work at each level due to slicing of the arrays (all elements are moved from the original array to the new smaller array)

   Can be improved to O(n) if indexes are used instead of slicing (no copying of elements)

Exercise 3: Coffee Inheritance

Create a Coffee class with the attributes:

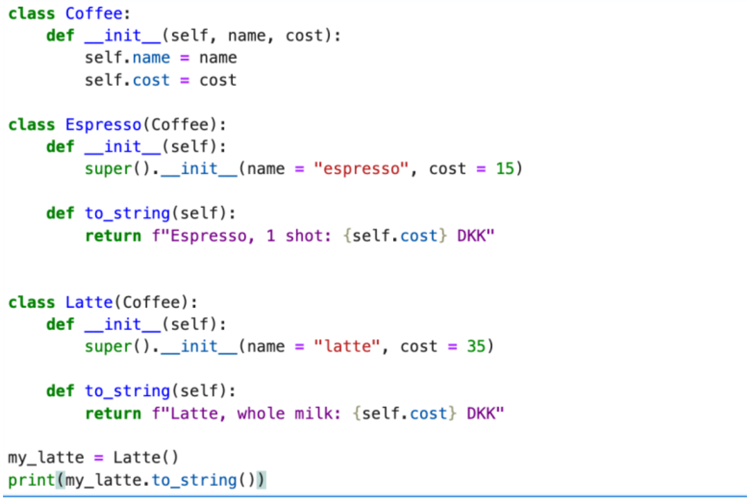
- name
- price

Create two subclasses:

- Espresso
- Latte

Each subclass should:

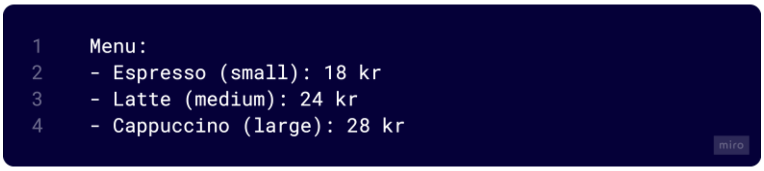
- Have its own default price
- Have a to_string function for printing its content



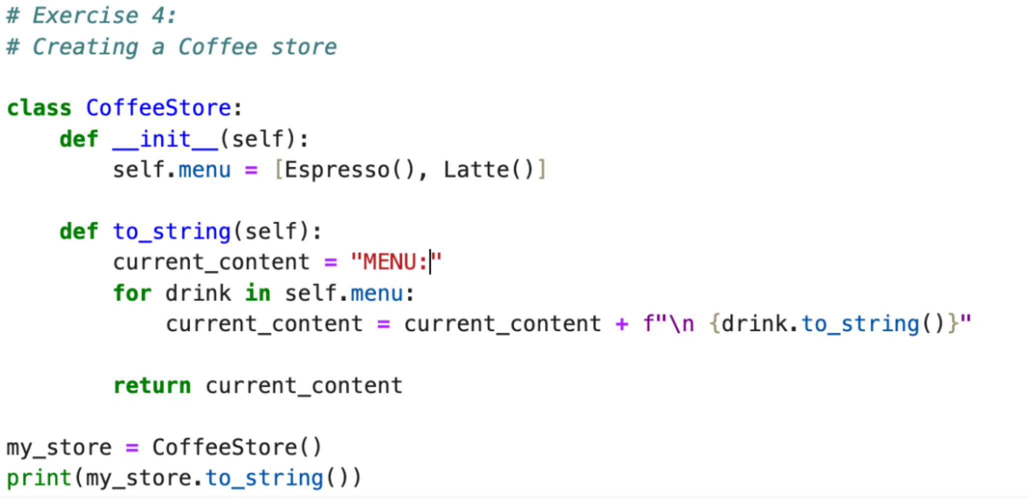
```python
class Coffee:
    def __init__(self, name, cost):
        self.name = name
        self.cost = cost

class Espresso(Coffee):
    def __init__(self):
        super().__init__(name = "espresso", cost = 15)

    def to_string(self):
        return f"Espresso, 1 shot: {self.cost} DKK"


class Latte(Coffee):
    def __init__(self):
        super().__init__(name = "latte", cost = 35)

    def to_string(self):
        return f"Latte, whole milk: {self.cost} DKK"

my_latte = Latte()
print(my_latte.to_string())
```

Exercise 4: Creating a CoffeeStore

Implement a CoffeeStore class that holds a coffee_menu with a to_string method that could return this:



```
Menu:
- Espresso (small): 18 kr
- Latte (medium): 24 kr
- Cappuccino (large): 28 kr
```

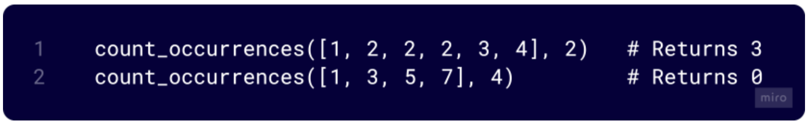


```python
# Exercise 4:
# Creating a Coffee store

class CoffeeStore:
    def __init__(self):
        self.menu = [Espresso(), Latte()]

    def to_string(self):
        current_content = "MENU:"
        for drink in self.menu:
            current_content = current_content + f"\n {drink.to_string()}"

        return current_content

my_store = CoffeeStore()
print(my_store.to_string())
```

Exercise 5: Binary search

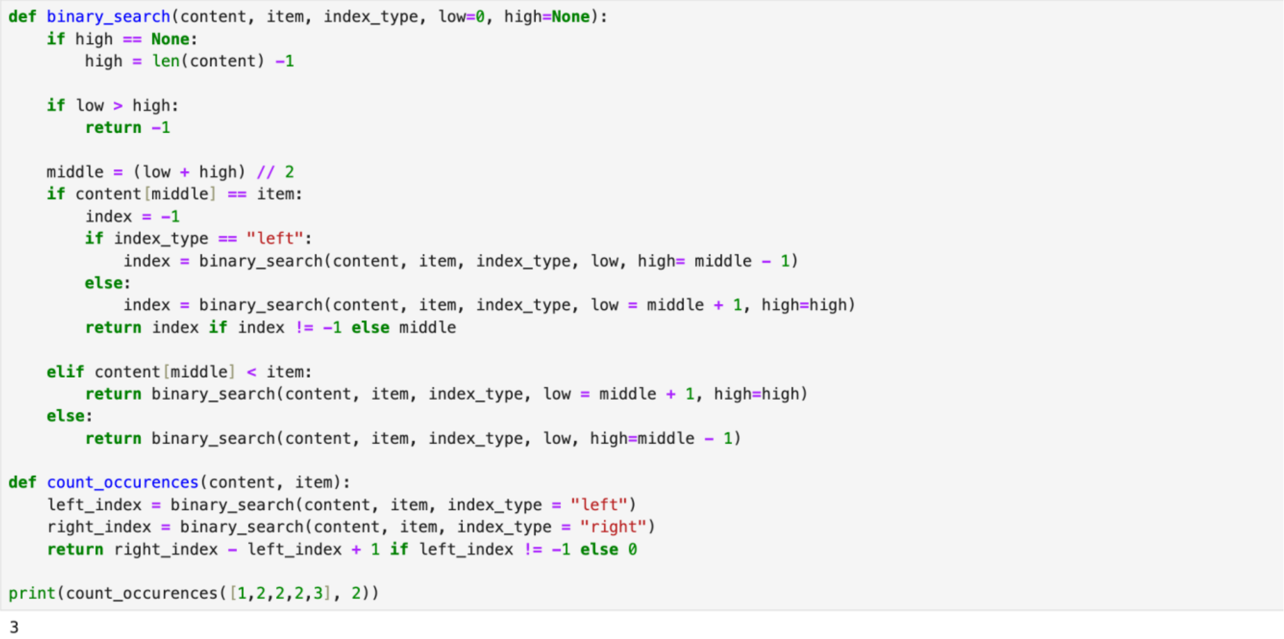
Implement a function that uses binary search on a sorted list of ints, to find out how many times an int occurs in the list:

Hint: find the index of left-most occurence and right-most and calculate the distance between these



```
count_occurrences([1, 2, 2, 2, 3, 4], 2)    # Returns 3
count_occurrences([1, 3, 5, 7], 4)          # Returns 0
```

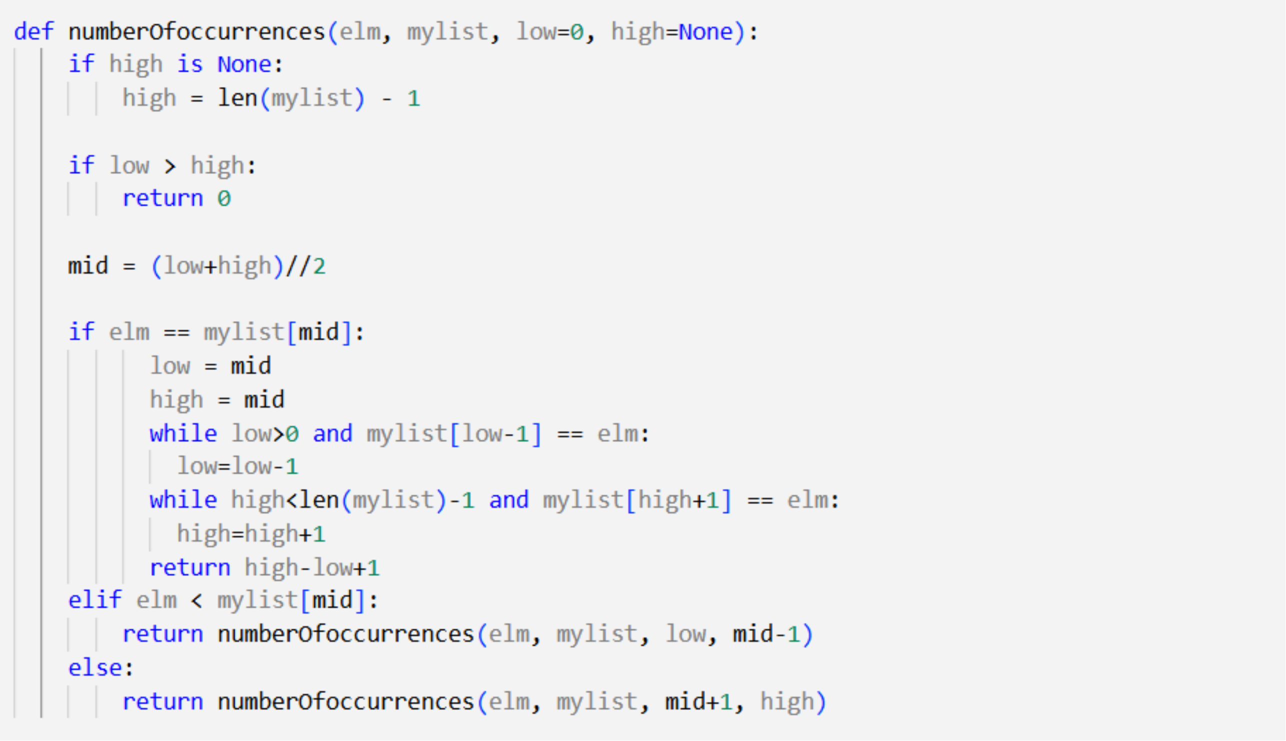
A solution that use recursion to find the element and then the first and last index – very complex, an example of recursion that is difficult to understand.



```
def binary_search(content, item, index_type, low=0, high=None):
    if high == None:
        high = len(content) -1

    if low > high:
        return -1

    middle = (low + high) // 2
    if content[middle] == item:
        index = -1
        if index_type == "left":
            index = binary_search(content, item, index_type, low, high= middle - 1)
        else:
            index = binary_search(content, item, index_type, low = middle + 1, high=high)
        return index if index != -1 else middle

    elif content[middle] < item:
        return binary_search(content, item, index_type, low = middle + 1, high=high)
    else:
        return binary_search(content, item, index_type, low, high=middle - 1)

def count_occurences(content, item):
    left_index = binary_search(content, item, index_type = "left")
    right_index = binary_search(content, item, index_type = "right")
    return right_index - left_index + 1 if left_index != -1 else 0

print(count_occurences([1,2,2,2,3], 2))
```

```
3
```

A solution uses recursion to find the element and then iteration to find the first and last element (much simpler).



```
def numberOfoccurrences(elm, mylist, low=0, high=None):
    if high is None:
        high = len(mylist) - 1

    if low > high:
        return 0

    mid = (low+high)//2

    if elm == mylist[mid]:
          low = mid
          high = mid
          while low>0 and mylist[low-1] == elm:
            low=low-1
          while high<len(mylist)-1 and mylist[high+1] == elm:
            high=high+1
          return high-low+1
    elif elm < mylist[mid]:
        return numberOfoccurrences(elm, mylist, low, mid-1)
    else:
        return numberOfoccurrences(elm, mylist, mid+1, high)
```