Question 1(a)

Complete the following function. The function takes in a string `thing`, and returns the string `"NOO!"` if thing is `"ghost"`, `"monster"`, or `"midterm"`, and the string `"YAY!"` otherwise. For example, the call `halloween_reaction("ghost")` should return the string `"NOO!"`, and the call `halloween_reaction("candy")` should return the string `"YAY!"`.

In [1]:
```
def halloween_reaction(thing):
    if thing in ["ghost", "monster", "midterm"]:
        return "NOO!"
    else:
        return "YAY!"
```

Other, less elegant, variants are also possible. For example:

In [2]:
```
def halloween_reaction2(thing):
    if thing == "ghost" or thing == "monster" or thing == "midterm":
        return "NOO!"
    else:
        return "YAY!"

def halloween_reaction3(thing):
    if thing == "ghost":
        return "NOO!"
    elif thing == "monster":
        return "NOO!"
    elif thing == "midterm":
        return "NOO!"
    else:
        return "YAY!"
```

Question 1(b)

Complete the following function. The function takes in a list `L`, and prints all the elements of the list `L`, in order, except for the first element and the last element. One element should be printed per line. For example, `print_mid_part({"pumpkins", "candy", "costumes", "autumn", "zombies"})`

should print

```
candy
costumes
autumn
```

In [3]:
```
def print_mid_part(L):
    for i in range(1, len(L)-1):
        print(L[i])
```

In [4]:
```
def print_mid_part(L):
    for elem in L[1:-1]:
        print(elem)
```

Question 2(a)

Complete the following function, which computes the sum of all the elements in the list `L` that are odd (i.e.,not divisible by 2.) For example, if `L` is `[1, 3, 4, 5]`, the function should return 9 since 9 = 1 + 3 + 5. Assume `L` is a list of integers.

In [5]:
```
def odds_sum(L):
    """Return the sum of the odd elements of L"""
    s = 0
    for elem in L:
        if elem % 2 == 0:
            s += elem
    return s
```

Question 2(b)

The following code is written using a for-loop. Write it using a `while`-loop instead:

```
for i in range(5, 500, 3):
    print(i)
```

Solution:

```
i = 5
while i < 500:
    print(i)
    i += 3
```

Question 3(a)

Complete the following function. The function takes in a list of strings `faves`, and a list of strings, of the same length, `kids`. The favourite thing about Halloween of the kid whose name is `kids[i]` is `faves[i]`. The function returns the list of the names of the kids whose favourite thing about Halloween is `"candy"` (`"candy"` is in lowercase). The names in the list that the function returns should appear in the same order that the names appear in the list kids. For example, if faves == ["candy", "costumes", "weather", "candy"] and kids == ["Bob", "Dorothy", "Mike", "Alice"],

then `kids_who_like_candy(faves, kids)` should return the list `["Bob", "Alice"]`.

In [6]:
```
def kids_who_like_candy(faves, kids):
    res = []
    for i in range(len(faves)):
        if faves[i] == "candy":
            res.append(kids[i])
    return res
```

Question 3(b)

Complete the following function. The function returns the cube root of n (i.e., $\sqrt[3]{n}$.) **Assume that $\sqrt[3]{n}$ is an integer**. You may **not** `import` any modules or use the `**` operator.

In [7]:
```
def cube_root(n):
    if n >= 0:
        for r in range(n+1):
            if r*r*r == n:
                return r
    else:
        for k in range(n-1, 0):
            if r*r*k == n:
                return k
```

Question 4

Complete the following code in such a way that the output is as stated in the comments.

```
def halloween_surprise():



if __name__ == '__main__':
    print(halloween_surprise())  #Output: 3
    print(halloween_surprise())  #Output: 2
    print(halloween_surprise())  #Output: 1
    print(halloween_surprise())  #Output: SURPRISE
```

In [8]:
```
def halloween_surprise():
    global count
    count -= 1
    if count > 0:
        return count
    else:
        return "SURPRISE!"


count = 4
if __name__ == '__main__':
    print(halloween_surprise())  #Output: 3
    print(halloween_surprise())  #Output: 2
    print(halloween_surprise())  #Output: 1
    print(halloween_surprise())  #Output: SURPRISE!
```

```
3
2
1
SURPRISE!
```

Question 5

Each of these subquestions contains a piece of code. Treat each piece of code independently (i.e., code in one question is not related to code in another), and write the expected output for each piece of code. If the code produces an error, write down the output that the code prints before the error is encountered, and then write "ERROR." You do not have to specify what kind of error it is.

In [9]:
```
L1 = [1, 2]
L2 = L1[:]
L2 = [3, 4]
print(L1)
```

```
[1, 2]
```

Reasoning: `L2` is assigned a copy of `L1`, and then it's assigned the new list `[3, 4]`. In both cases, only the variable table is affected. The list that `L1` refers to, which lives in the memory table, is unaffected.

In [10]:
```
def f():
    n = 5

n = 4
n = f()
print(n)
```

```
None
```

Reasoning: functions that return nothing actually return `None`. The last thing that's assign to `n` before it is printed is `None`.

```
def f(L):
    L2 = L
    L = [1, 2]
    L[0] = 5
    print(L)
L = [2, 3]
f(L)
print(L[0])
print(L2)
```

Output: [5, 2]

```
2

ERROR
```

Reasoning: Inside of `f()`, the list `L` is a local variable (since it's assigned the new list `[1, 2]`. The changes are reflected when `L` is printed inside of `f()`, but not outside of it.

`L2` is a local variable, so trying to pring it outside of `f()` produces an error.

In [11]:
```
L1 = [[[1, 2], 3], 4]
L2 = L1[:]
L1[1] = 5
L1[0][1] = 5
L1[0][0][1] = 5
print(L2)
```

```
[[[1, 5], 5], 4]
```

Reasoning: `L2` is a shallow copy of `L1` -- only one new list is created. `L1` and `L2` are not aliases, but `L1[0]` and `L2[0]` are, as are `L1[0][0]` and `L1[0][0]`.

Question 6

Marking scheme: 2 for the idea, 2 for the implementation (implementations of incorrect ideas don't get the full 2 marks)

Write a function that determines if a list has only a single "peak." A list has a single peak if the list is non-decreasing up to a certain point, and is non-increasing after that point. For example, the list `[1, 2, 2, 3, 4, 5, 0, -1]` has a single "peak," since it is non-decreasing up to 5, and non-increasing after 5. On the other hand, the list [1, 2, 1, 2] has more than one "peak," so we say that it is not true that it has only a single "peak." Non-decreasing lists like `[1, 2, 3]` and non-increasing lists like `[3, 2, 1]` are considered to have a single "peak."

In [12]:
```
def has_single_peak(L):
    """Return True iff the list of integers L has only a
    single peak"""

    #first, keep increasing i while the list up to L[i] is
    #non-decreasing
    i = 1
    while i < len(L) and L[i] >= L[i-1]:
        i += 1
    #Now, keep increasing i while the list up to L[i] is
    #non-increasing, starting from the point where it
    #stopped being non-increasing
    while i < len(L) and L[i] <= L[i-1]:
        i += 1

    #If we reached the end of the list that way, the list
    #only has one peak. Otherwise, it doesn't

    return i == len(L)
```

An approach with a state variable: the idea is to keep track of how many times we switched directions (in terms of going up ot down). If we have to switch directions more than once, we have more than one peak

In [13]:
```
def has_single_peak2(L):
    switched = False
    for i in range(1, len(L)):
        if not switched:
            if L[i] < L[i-1]:
                switched = True
        else:
            #Switched once, can't switch again
            if L[i] > L[i-1]:
                return False

    #Never returned False: so we can return True
    return True
```

Question 7

Marking scheme: 2.5 for the idea, 1.5 for the implementation (implementations of incorrect ideas don't get the full 1.5 marks)

Write a function with the signature `def max_arrivals_2hrs(arrivals)` that returns the maximum number of arrivals of trick-or-treating kids that happened within the span of two hours, as recorded in the list `arrivals`. The list `arrivals` is a list of times (in minutes) after 5PM on Oct. 31 that kids arrived trick-or-treating. For example, if arrivals is `[0, 30, 40, 150, 160, 170, 370]`, then kids arrived at 5:00PM, 5:30PM, 5:40PM, 7:30PM, 7:40PM, etc.; and the maximum number of arrivals within the span of two hours was 3 (the arrivals at 5:00PM, 5:30PM, and 5:40PM.) You can assume that the list arrivals contains only integers and is non-decreasing.

In [14]:
```
#The most obvious approach: try all possible start times and end times
#for the span of two hours
def max_arrivals_2hrs(arrivals):
    max_arrivals = 0
    for i in range(len(arrivals)):
        for j in range(len(arrivals)):
            if arrivals[j]-arrivals[i] < 120:
                max_arrivals = max(max_arrivals, j-i+1)

    return max_arrivals
```

In [15]:
```
#A more efficient approach
def max_arrivals_2hrs(arrivals):
    max_arrivals = 0

    start_span = 0
    end_span = 0
    while end_span < len(arrivals):
        if arrivals[end_span] - arrivals[start_span] < 120:
            max_arrivals = max(max_arrivals, end_span-start_span+1)
            end_span += 1
        else:
            start_span += 1
    return max_arrivals
```