Question 1. [9 marks]

Treat each subquestion independently (*i.e.*, code in one question is not related to code in another), and answer each question in the space provided.

Part (a) Functions [2 marks]

Write a function that prints "Happy Halloween" (without the quotes). Also write code that calls the function that you wrote.

Sample Solution:

```
def make_a_wish():
    print("Happy Halloween")

make_a_wish()
```

Marking Scheme:

- **Correct function definition:** [1 mark]
- **Calling declared function:** [1 mark]
- **Syntax errors:** [−1 mark each]

Part (b) Loops and Lists [3 marks]

Complete the following function. The function should print all the elements of the list `L` that is passed to it as an argument, except the last element. One element should be printed per line. For example, `print_almost_all([41, 42, 43])` should print:

```
41
42
```

```
def print_almost_all(L):
```

Sample Solution:

```
def print_almost_all(L):
    for i in range(len(L) - 1):
        print(L[i])
```

Marking Scheme:

- **Traversing correct range in list L:** [1 mark]
- **Correct loop:** [1 mark]
- **Correct print call:** [1 mark]
- **Syntax errors:** [−1 mark each]

Part (c) **Global Variables** [2 MARKS]

Complete the function `h()` so that the effect of running the code below is to print "candy" (without the quotes) and not "pumpkin."

```
def h():



treat = "pumpkin"
h()
print(treat)
```

SAMPLE SOLUTION:

```
    def h():
        global treat
        treat = "candy"
```

MARKING SCHEME:

- **Use of global statement:** [1 mark]
- **Successfully changing the value of variable *treat*:** [1 mark]
- **Syntax errors:** [−1 mark]
- **Code that leads to incorrect output:** [−1 mark]

Part (d) **Order of Execution** [2 MARKS]

What is the output of the following piece of code?

```
x = 4
y = x + 5
x = 7
print(x, y)
```

SAMPLE SOLUTION:

```
7 9
```

MARKING SCHEME:

- **No part marks.**

Question 2. [16 MARKS]

Part (a) [4 MARKS]

Complete the following function. The function returns the number of kids whose favourite part of Halloween is candy, according to the data in the dictionary `faves`. For a string `name`, `faves[name]` contains the favourite part of Halloween of the kid named `name`. Both the names and the favourites are lowercase strings. For example,

`count_candy({"ben": "costumes", "adam":"candy", "matt":"candy", "anna":"costumes"})`
returns 2, since two kids (adam and matt) like candy. Assume `faves` is given in the correct format.

```
def count_candy(faves):
```

SAMPLE SOLUTION:

```
def count_candy(faves):
    candy_lovers = 0
    for k in faves:
        if faves[k] == "candy":
            candy_lovers += 1
    return candy_lovers
```

MARKING SCHEME:

- **Successfully keeping track of number of people who favour "candy":** [1 mark]
- **Correctly accessing elements of the dictionary:** [1 mark]
- **Correctly traversing through all the keys in *faves*:** [1 mark]
- **Returning correct value:** [1 mark]
- **Syntax errors:** [-1 mark (upto -2)]

Part (b) [4 MARKS]

Write code that prints all the integers from -500 to 500, from smallest to largest. That is, the code should print all of
$-500, -499, -498, ..., -2, -1, 0, 1, 2, 3, ..., 499, 500.$
Each integer should be printed on a separate line.

SAMPLE SOLUTION:

```
i = -500
while i <= 500:
    print(i)
    i += 1
```

MARKING SCHEME:

- **Correctly traversing from -500 to 500:** [1 mark]
- **Good loop:** [2 marks]
- **Correct Output:** [1 mark]
- **Syntax errors:** [-1 mark (upto -2)]

Part (c) [4 marks]

Complete the following function according to its docstring.

```
def get_key_by_val(d, val):
  """Return a key in the dictionary d such that d[key] == val. If there are several
  such keys, return one of them (it doesn't matter which). If there are no such
  keys, return None."""
```

Sample Solution:

```
    def get_key_by_val(d, val):
        for k in d:
            if d[k] == val:
                return k
        return None
```

Marking Scheme:

- **Correctly traversing *d*:** [1 mark]
- **Correctly accessing elements of the dictionary:** [1 mark]
- **Successfully checking for *val*:** [2 marks]
- **Correctly returning:** [1 mark]
- **Syntax errors:** [-1 mark (upto -2)]

Part (d) [4 marks]

Complete the following function according to its docstring.

```
def all_not_same(a, b, c):
  """Return True if a, b, and c are **not** all equal to each other. Return
  False otherwise."""
```

Sample Solution:

```
    def all_not_same(a, b, c):
        if a == b:
            if b == c:
                return True
        return False
```

Marking Scheme:

- **Successfully comparing *a*, *b*, and *c*:** [2 marks]
- **Correct returning *True*:** [1 mark]
- **Correct returning *False*:** [1 mark]
- **Syntax errors:** [-1 mark (upto -2)]

Question 3. [8 marks]

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.

Part (a) [2 MARKS]

```
def f(n):
  print(n)

m = 2
f(m)
```

OUTPUT:

```
2
```

MARKING SCHEME:

- **No part marks.**

Part (b) [2 MARKS]

```
def f(n):
  n = 5

n = 2
f(n)
print(n)
```

OUTPUT:

```
2
```

MARKING SCHEME:

- **No part marks.**

Part (c) [2 MARKS]

```
def f(L):
  L[0] = 42
  print(L[0])

L = [1, 2, 3]
f(L)
print(L[0])
```

OUTPUT:

```
42
42
```

MARKING SCHEME:

- **First line:** [0.5 mark]
- **Second line:** [1.5 mark]
- **Incorrect format:** [-0.5 mark]
- **Correct output + Error:** [-0.5 mark]

Part (d) [2 MARKS]

```
d1 = {1:[2], 3:[4]}
d2 = d1
d1 = {1: d2[1], 3: d2[3]}
d1[1][0] = 5
print(d1[1][0], d2[1][0])
```

OUTPUT:

```
5 5
```

MARKING SCHEME:

- **First *5*:** [0.5 mark]
- **Second *5*:** [1.5 mark]
- **Incorrect format:** [-0.5 mark]
- **Correct output + Error:** [-0.5 mark]

Question 4. [6 MARKS]

Complete the function below according to its docstring. For full marks, do **not** use `list`'s method `.extend()` anywhere in your solution. Solutions that use `.extend()` will get at most 3 marks.

```
def flatten(list_of_lists):
    """Return a new list that contains every value in each of the sub-lists of
    list_of_lists.  For example,
    >>> flatten([[1, 3], ['a', 'b', 'c']])
    [1, 3, 'a', 'b', 'c']
    Assume that all the elements of list_of_lists are lists and that the elements
    of the elements of list_of_lists are integers or strings."""
```

SAMPLE SOLUTION:

```
def flatten(list_of_lists):
    output = []
    for l in list_of_lists:
        for element in l:
            output.append(element)
    return output
```

MARKING SCHEME:

- **Successfully traversing *list_of_lists*:** [1 mark]
- **Successfully traversing elements of elements of *list_of_lists*:** [1 mark]
- **Successfully keeping tracking of all the elements in a new list:** [1 mark]
- **Correctly appending to the new list:** [2 mark]
- **Returning correct value:** [1 mark]
- **Syntax errors:** [-1 mark (upto -2)]

Question 5. [6 MARKS]

Write code that prints out all the possible 4-letter strings that can be written using the letters “a”, “b”, “c”, “d”, “e”, “f” and “g” (and no other character) such that no letter appears in any string more than twice. For example, `"ccbb"` should be printed, but `"aaab"` should not. Hint: think about how you would print *all* the 4-letter strings.

SAMPLE SOLUTION:

```
letters = ['a','b','c','d','e','f','g']
for a in letters:
    for b in letters:
        for c in letters:
            for d in letters:
                candidate = a+b+c+d
                if no_3_4_repeats(candidate):
                    print(candidate)

def no_3_4_repeats(candidate):
    for c in candidate:
        if candidate.count(c) > 2:
            return False
    return True
```

MARKING SCHEME:

- **Good structure and correct output:** [1 mark]
- **Enumerating through all relevant strings:** [2 marks]
- **Filtering: correctly counting the characters in candidate strings:** [2 marks]
- **Filtering: correctly using the counts to filter out illegal strings:** [1 mark]
- **Duplicates in output:** [-1 mark]
- **Syntax errors:** [-1 mark (upto -2)]

Bonus. [3 MARKS]

Write a function with the signature `near_anagram(w1, w2)` that takes in two strings, `w1` and `w2`, and returns `True` iff `w1` can be obtained from `w2` by rearranging the characters in `w2`, and then changing exactly one of the characters for another character. For example `near_anagram("cat, "tap")` is `True` since `"cat"` can be rearranged into `"tac"`, and then changed into `"tap"` by exchanging `c` for `p`. Assume `w1` and `w2` only contain lowercase letters.

SAMPLE SOLUTION:

```
def near_anagram(w1, w2):
    diff1_2 = 0
    diff2_1 = 0
    for c in set(s1+s2):
        if s1.count(c)-s2.count(c) == 1:
            diff1_2 += 1
        elif s1.count(c)-s2.count(c) == -1:
            diff2_1 += 1
        elif abs(s1.count(c)-s2.count(c)) > 1:
            return False
    return diff1_2 == diff2_1 == 1
```

MARKING SCHEME:

- **Correct solution:** [3 mark]
- **Correct solution except for an easily fixable problem:** [2 mark]
- **Incorrect solution but code exhibits a good attempt at all the required steps:** [1 mark]
- **Code works if the strings don't have any repeated characters:** [0.5 mark]