[10 marks]

Question 1

Evaluate the following expressions, and provide the output in each case.

(a) `'happiness'[1:4] + 'spelling'[4::-2]`

A: *'apples'*

(b) `2 ** 3 - 1`

A: 7

(c) `'grin' in 'chagrin' and 'you' > 'me'`

A: *True*

(d) `sorted(list({'fruit': ('pear',), 'rock': ('opal', 'ruby')}.values()))[-1][-1]`

A: *'pear'*

(e) `[k/2 for k in range(1, 7, 2)]`

A: *[0.5, 1.5, 2.5]*

[8 marks]

Question 2

Rewrite the following function, replacing the `for` loop with a `while` loop, but preserving the remainder of the original code structure:

```
def grabble_all(wookles):
    plods = []
    for i in range(len(wookles)):
        if i % 3 != 0:
            plods.append(wookles[i] * i)
    return plods
```

A:

```
def grabble_all(wookles):
    plods = []
    i = 0
    while i < len(wookles):
        if i % 3 != 0:
            plods.append(wookles[i] * i)
        i += 1
    return plods
```

[10 marks]

Question 3

The following code is intended to generate the first `num` numbers in the sequence defined as follows: beginning with `start`, each successive number $n + 1$ in the sequence is formed from the preceding number n by adding the product of the nonzero digits of n.
For example, 38 is followed by 62 (ie, $38 + (3 \times 8)$), while 104 is followed by 108 (ie, $104 + (1 \times 4)$). Thus:

```
>>> digit_prod(15, 1)
[1, 2, 4, 8, 16, 22, 26, 38, 62, 74, 102, 104, 108, 116, 122, 126]
```

As presented, the lines of the function are out of order. Put the line numbers in the correct order and introduce appropriate indentation (indent the line numbers to show how the corresponding lines would be indented in your code).

```
1  if digit > 0:
2  for i in range(num):
3  prod *= digit
4  return seq
5  cur += prod
6  seq = [start]
7  prod = 1
8  cur_str = str(cur)
9  for digit in cur_str:
10 def digit_prod(num, start):
11 digit = int(digit)
12 cur = seq[-1]
13 seq.append(cur)
```

A:
```
10
    6
    2
        12
        8
        7
        9
            11
            1
                3
        5
        13
    4
```

OR

```
10
    6
    2
        7
        12
        8
        9
            11
            1
                3
        5
        13
    4
```

[9 marks]

Question 4

The following function is meant to take a list `a` as input, and return the "flattened" version of `a`, based on removing any sub-lists but preserving the sequence of items. For example:

```
>>> flatten_list([1, 2, 3,[4, 5],[6, [7, 8, 9]]])
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

As presented, there are bugs in the code. Identify exactly three (3) errors in the code (using the provided line numbers), identify for each whether it is a "syntax", "run-time" or "logic" error, and provide a replacement line which corrects the error.

```
1  def flatten_list(a, result=None):
2      if result is None:
3          result == []
4
5      for x in a:
6          if x.type() == list:
7              flatten_list(x:result)
8          else:
9              result += x
10
11     return result
```

A: *3 out of 4 of:*

- *line 3; logic; should be* `result = []`
- *line 6; run-time; should be* `if type(x) == list`
- *line 7; syntax; should be* `flatten_list(x, result)`
- *line 9; run-time; should be* `result.append(x)`

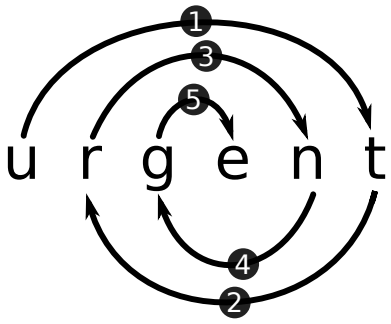
Question 5

[10 marks]

The following code is intended to determine whether a word is a "spiral word" word or not (ascending or descending), but contains logic errors:

```
def spiral_word(word):
    sorted_word = ''.join(sorted(word))
    despiral_word = ''
    while word:
        despiral_word += word[0]
        word = word[:0:-1]
    if (despiral_word == sorted_word
        or despiral_word == sorted(sorted_word, reverse=True)):
        return True
    return False
```

Recall that a "spiral word" is a string made up of 2 or more letters, where the sequence of characters defined by spiralling over the letters from outside in, is in *strictly* ascending or descending order. When we "spiral" over the letters, we start with the first letter, then examine the last letter, then the second letter, followed by the second-last letter, etc, spiraling towards the middle letter. For example, in the case of `'urgent'`, the letter sequence defined by the spiral is `'u'` (the first letter), `'t'` (the last letter), `'r'` (the second letter), `'n'`, `'g'`, and finally `'e'`, as illustrated below:



Examples of spiral words are `'flush'` (in ascending order: `'f'` < `'h'` < `'l'` < `'s'` < `'u'`) and `'urgent'` (in descending order: `'u'` > `'t'` > `'r'` > `'n'` > `'g'` > `'e'`). An example of a non-spiral word, on the other hand, is `'aunt'` (`'a'` > `'t'` > `'u'` < `'n'`).

The provided code is imperfect, in that it sometimes correctly detects spiral and non-spiral words, but equally, sometimes misclassifies a spiral word as being a non-spiral word, and sometimes misclassifies a non-spiral word as being a spiral word.

(a) Provide an example of a spiral word that is correctly classified as such by the provided code (i.e. a spiral word input where the return value is `True`):

A: *Any increasing spiral word without repeating letters (e.g. `'flush'`)*

(b) Provide an example of a non-spiral word that is correctly classified as such by the provided code (i.e. a non-spiral word where the return value is `False`):

A: *Any word which is not strictly increasing or decreasing (e.g. `'cat'`)*

(c) Provide an example of a non-spiral word that is *in*correctly analysed as being a spiral word by the provided code (i.e. a non-spiral word where the return value is `True`):

A: *An increasing word with a repeating letter (e.g. `'fluush'`)*

(d) Provide an example of a spiral word that is *in*correctly analysed as being a non-spiral word by the provided code (i.e. a spiral word where the return value is `False`):

A: *A decreasing spiral word (e.g. `'urgent'`)*

Part 2: Generating Code

Question 6

[10 marks]

Write a single Python statement that generates each of the following exceptions + error messages, assuming that it is executed in isolation of any other code.

(a) `AttributeError: 'str' object has no attribute 'a'`

A: *any string with attribute* `a`*, e.g.* `'hi'.a`

(b) `IndexError: string index out of range`

A: *a string with out of range index, e.g.* `'hi'[2]`

(c) `KeyError: 'a'`

A: *attempt to index a dictionary not containing key* `'a'`*, e.g.* `{}['a']`

(d) `TypeError: 'set' object does not support indexing`

A: *an index operation over any set, e.g.* `set()[0]`

(e) `StopIteration`

A: *any attempt to iterate past the end of an interable, e.g.* `next(iter([]))`

Question 7

[15 marks]

The following code centres around a fictional TV series, and determines potential "match-ups" between male and female characters from different "houses". Here, the characters are represented in a list of 3-tuples, each of which is made up of the house, given name, and (binary) gender (`'f'` = female, and `'m'` = male). Complete the code by providing a single statement to insert into each of the numbered boxes. Note that your code should run at the indentation level indicated for each box.

Recall that `groupby` combines items which share some common criterion, and `product` generates all combinations of the elements in the provided arguments.

```
from itertools import groupby, product

characters = [('banister', 'je t\'aime', 'm'), ('killjoy', 'freon', 'm'),
              ('snark', 'bob', 'm'), ('snark', 'salsa', 'f'),
              ('banister', 'tyrant', 'm'), ('tag-along', 'dani', 'f'),
              ('snark', 'heya', 'f'), ('tag-along', 'con', 'm')]

def get_house(x):
    [      1      ]

def matched_pairs(characters):
    [      2      ]
    houses = {}
    for k, g in groupby([      3      ])
        houses[k] = list(g)
    possible = set()
    for house_a, house_b in product(houses, houses):
        [      4      ]
            for char_a, char_b in product(houses[house_a], houses[house_b]):
                if char_a[2] != char_b[2]:
                    cur_pair = sorted([char_a, char_b])
                    [      5      ]
    return possible

if __name__ == '__main__':
    print(matched_pairs(characters))
```

When run, your code should produce the following output:

```
{(('snark', 'heya', 'f'), ('tag-along', 'con', 'm')),
  (('killjoy', 'freon', 'm'), ('snark', 'heya', 'f')),
  (('banister', 'tyrant', 'm'), ('tag-along', 'dani', 'f')),
  (('banister', "je t'aime", 'm'), ('snark', 'heya', 'f')),
  (('snark', 'salsa', 'f'), ('tag-along', 'con', 'm')),
  (('banister', "je t'aime", 'm'), ('snark', 'salsa', 'f')),
  (('snark', 'bob', 'm'), ('tag-along', 'dani', 'f')),
  (('banister', 'tyrant', 'm'), ('snark', 'heya', 'f')),
  (('banister', "je t'aime", 'm'), ('tag-along', 'dani', 'f')),
  (('killjoy', 'freon', 'm'), ('tag-along', 'dani', 'f')),
  (('killjoy', 'freon', 'm'), ('snark', 'salsa', 'f')),
  (('banister', 'tyrant', 'm'), ('snark', 'salsa', 'f'))}
```

A: (1) `return x[0]`

(2) `characters = sorted(characters)`

(3) `characters, get_house`

(4) `if house_a != house_b:`

(5) `possible.add(tuple(cur_pair))`

[25 marks]

Question 8

A "tatami" puzzle takes the form of an $N \times N$ square grid, made up of a series of horizontal tatami tiles of identical size $M \times 1$ (where N is a whole-number multiple of M), in which each cell must be filled with a single number of value $\{1, 2, ..., M\}$, subject to the following constraints:

- the tatami tiles must not overlap, and must form a square of prescribed size
- within each row, each number $\{1, 2, ..., M\}$ must be represented an equal number of times
- within each column, each number $\{1, 2, ..., M\}$ must be represented an equal number of times
- within each tile, each number $\{1, 2, ..., M\}$ must be represented exactly once
- a given number must not be adjacent to itself in any row or column

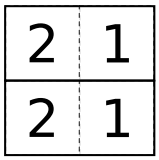
The following are examples of valid tatami puzzles, made up of two 2×1 tiles in the first instance, and twelve 3×1 tiles in the second instance:

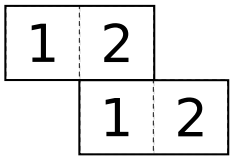
	0	*1*
0	2	1
1	1	2

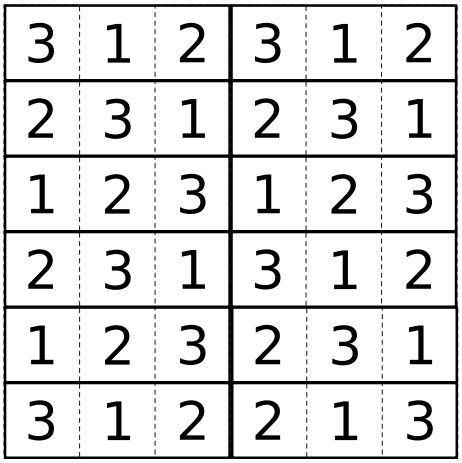
3	1	2	3	1	2
2	3	1	2	3	1
1	2	3	1	2	3
2	3	1	3	1	2
1	2	3	2	3	1
3	1	2	1	2	3

Note that we will refer to individual elements within tiles via the (x, y) coordinate system indicated for the first example, where the number 1 in the first row has coordinates $(1, 0)$, for example.

On the next page are examples of *invalid* tatami puzzles: for the first, the columns do not include a single 1 and a single 2, and the 2s are adjacent in the first column and the 1s are adjacent in the second column; for the second, the tatami tiles don't form a square; and for the third, the 2 in the bottom-right tile is adjacent to two other 2s, and the numbers 1 and 2 are represented unequally in the fourth and fifth columns.







Write a function `tatami` that takes three arguments:

- `tlist` — a list of tiles, each in the form of a 2-tuple, made up of:
  (1) the (x, y) coordinates of the left element of the tile, in the form of a 2-tuple of (positive) integers
  (2) a list of numbers contained in the tile, in sequence from the left element
- `tsize` — a positive integer indicating the size of each tile (corresponding to M in the preceding description)
- `grid_size` — a positive integer indicating the size of the overall square (corresponding to N in the preceding description)

The function should return `True` if the input describes a valid tatami puzzle, and `False` otherwise. Example calls to the function are (corresponding to the first through fifth examples, respectively):

```
>>> tatami([((0, 0), [2, 1]), ((0, 1), [1, 2])], 2, 2)
True
>>> tatami([((0, 0), [3, 1, 2]), ((3, 0), [3, 1, 2]),
... ((0, 1), [2, 3, 1]), ((3, 1), [2, 3, 1]),
... ((0, 2), [1, 2, 3]), ((3, 2), [1, 2, 3]),
... ((0, 3), [2, 3, 1]), ((3, 3), [3, 1, 2]),
... ((0, 4), [1, 2, 3]), ((3, 4), [2, 3, 1]),
... ((0, 5), [3, 1, 2]), ((3, 5), [1, 2, 3])], 3, 6)
True
>>> tatami([((0, 0), [2, 1]), ((0, 1), [2, 1])], 2, 2)
False
>>> tatami([((0, 0), [1, 2]), ((1, 1), [1, 2])], 2, 2)
False
>>> tatami([((0, 0), [3, 1, 2]), ((3, 0), [3, 1, 2]),
... ((0, 1), [2, 3, 1]), ((3, 1), [2, 3, 1]),
... ((0, 2), [1, 2, 3]), ((3, 2), [1, 2, 3]),
... ((0, 3), [2, 3, 1]), ((3, 3), [3, 1, 2]),
... ((0, 4), [1, 2, 3]), ((3, 4), [2, 3, 1]),
... ((0, 5), [3, 1, 2]), ((3, 5), [2, 1, 3])], 3, 6)
False
```

Note that marks will be awarded for partial solutions to this problem, based on what proportion of the different constraints your code successfully checks for.

A:

```
from itertools import product
from collections import defaultdict

def tatami(tlist, tsize, board_size):
    all_vals = list(range(1, tsize + 1))
    board = {}
    for i, j in product(range(board_size), range(board_size)):
        board[i,j] = 0
    for start, tatami in tlist:
        if len(tatami) != tsize or sorted(tatami) != all_vals:
            return False
        curr_pos = start
        for curr_val in tatami:
            if curr_pos not in board or board[curr_pos]:
                return False
            board[curr_pos] = curr_val
            curr_pos = (curr_pos[0] + 1, curr_pos[1])
    for i in range(board_size):
        row = []
        col = []
        row_vals = defaultdict(int)
        col_vals = defaultdict(int)
        for j in range(board_size):
            row_vals[board[i, j]] += 1
            if j and board[i,j-1] == board[i,j]:
                return False
            col_vals[board[j, i]] += 1
            if j and board[j-1, i] == board[j,i]:
                return False
        for val in all_vals:
            if (row_vals[val] != board_size/tsize
                or col_vals[val] != board_size/tsize):
                return False
    return True
```

Part 3: Conceptual Questions and Applications of Computing

Question 9: Computing Fundamentals

(a) What is the difference between "integration" and "unit" testing, in the context of execution-based verification?

A: *integration testing = make sure all components work in unison; unit testing = testing of each component in isolation*

(b) Identify two properties that every object in Python has, and illustrate with the use of examples.

A: *any two out of: value, type, identity*

Question 10: Applications of Computing

(a) With the aid of an example, describe what "regular expressions" are used for, including mention of what Python type they are applicable to.

A: *Regular expressions are used to describe string patterns/extract components out of strings*

applicable to strings

(b) With reference to the transmission of an encrypted message, describe the role of the "private" and "public" keys in "public key cryptography".

A: *public key used to encrypt the message (accessible to anyone); private key used by the recipient to decrypt the message (accessible only to the recipient)*

[6 marks]

Question 11: URLs and the Web

With reference to the following URL:

```
https://www.comp10001.hq:10001/exam/nirvana
```

provide the portion of the URL which stipulates each of the following:

(i) the port

A: *10001*

(ii) the path

A: *exam/nirvana*

(iii) the protocol

A: *https*

(iv) the host name

A: *www.comp10001.hq*