NATIONAL UNIVERSITY OF SINGAPORE

CS1010S — Programming Methodology

Semester 1, 2017/2018

Time Allowed: 2 hours

INSTRUCTIONS TO STUDENTS

1. Please write your Student Number only. Do not write your name.
2. The assessment paper contains **FIVE (5) questions** and comprises **FOURTEEN (14) pages** including this cover page.
3. Weightage of questions is given in square brackets. The maximum attainable score is 100.
4. This is a **CLOSED** book assessment, but you are allowed to bring **ONE** double-sided A4 sheet of notes for this assessment.
5. Write all your answers in the space provided in the **ANSWER BOOKLET**.
6. You are allowed to write with pencils, as long as it is legible.
7. Common `List` and `Dictionary` methods are listed in the Appendix for your reference.

This page is intentionally left blank.

It may be used as scratch paper.

Question 1: Python Expressions [30 marks]

There are several parts to this problem. Answer each part **independently and separately**. In each part, one or more Python expressions are entered into the interpreter (Python shell). Determine the response printed by the interpreter for the final expression entered and **write the exact output in the answer box**. If the interpreter produces an error message, or enters an infinite loop, explain why.

The code is replicated on the answer booklet. You may show your workings **outside the answer box** in the space beside the code. Partial marks will be awarded for workings if the final answer is wrong.

A.
```
a = [1, [2, 3]]
b = a[::-1]  # slice from back to front
a[1][0], a[1][1] = b[0][1], b[0][0]
print(a, b)
```
[5 marks]

B.
```
def foo(x, y):
    return lambda z: x
def bar(x, y):
    return lambda z: y
print(foo(bar, bar)(0)(1, 2)(3))
```
[5 marks]

C.
```
def sherlock(*args):
    try:
        print("Deduction: " + args[1] \
              + args[-1])
    except ZeroDivisionError:
        print("It is")
    except TypeError:
        print("elementary")
    except Exception:
        print("my dear")
    except IndexError:
        print("Watson")
    finally:
        return (args[0],)
print(sherlock("holmes"))
```
[5 marks]

D.
```
s = 'Lollapalooza'
d = {}
for i in range(len(s)):
    d[s[i%5]] = s[i]
print(d)
```
[5 marks]

E.
```
j = ()
for i in range(1, 10, 3):
    if i % 6 == 1:
        j = (j, i)
    else:
        j = (i, j)
print(j)
```
[5 marks]

F.
```
d = {'c': 's', 10: 10,
     's': 10}
s, k = '', 'c'
while k in d:
    s += str(d[k])
    k = d[k]
print(s)
```
[5 marks]

Question 2: Connect Four [28 marks]

Connect Four is a two-player game played using a vertically suspended grid. Players take turns dropping their tokens into a column of the grid. The tokens will fall straight down, occupying the bottom-most available row of the column. For example, in the figure below, a token dropped into the third column from the left will end up on the third row from the bottom.

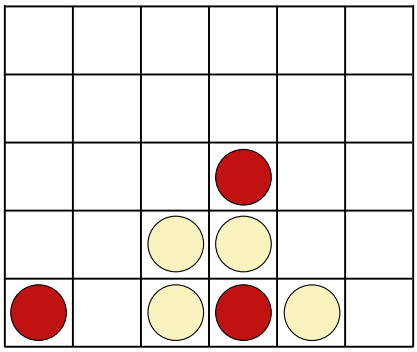


Figure 1: An example midway into a Connect Four game.

A player wins the game if they are the first to form a continuous horizontal, vertical or diagonal line of at least four of their own tokens.

We can represent the game state of a Connect Four game in Python using a list of lists. Each element of the list represents a row of the grid, which is a list of elements. An empty spot on the grid can be represented by `None`, and a player's token can be a string or integer. For example, one player might use the string `"P1"` and the other might use the integer `42` as their tokens. You may assume the two players will not use the same value as their tokens.

A. [Warm up] The function `make_grid` takes as inputs the number of rows and columns of a playing grid, and returns an empty grid of the given size, represented as a list of lists as stated above. Provide an implementation for `make_grid`. [4 marks]

B. In class, we discussed some matrix functions that operate on a matrix represented as a list of lists. Here are some of the functions:

- `rows(m)` which takes in a matrix *m* and returns the number of rows of the matrix.
- `cols(m)` which takes in a matrix *m* and returns the number of columns of the matrix.
- `get(m, row, col)` which returns the value at the given row and column of the matrix *m*. If the given coordinates are beyond the bounds of the grid, `None` is returned. For simplicity, you may assume the first row and column is $0, 0$.
- `set(m, row, col, val)` sets the element at the given row and column of the matrix *m* to *val*.
- `transpose(m)` transposes the matrix *m*, i.e., the rows will become the columns.
- `rotate(m)` rotates the matrix *m* 90 degrees clockwise, i.e., the topmost row will become the rightmost column.

The function `drop` takes as inputs a game grid, a column number, and a player's token. It updates the grid to the new state where the player makes a move by dropping their token in the given column. If the stated column is already filled to top, a `ColumnFullError(col)` is raised. Assume the `ColumnFullError` class has already been defined and that the row and column numbering starts from 0.

Provide an implementation for the function `drop`. You may assume the matrix functions mentioned above have been defined and available to use.

Hint: It does not actually matter which row or column you choose to be $(0, 0)$, as long as you remain consistent. [4 marks]

C. It is quite straightforward to check if a player has formed a **continuous horizontal** row of four of the same token by writing a function `check_rows` as follows:

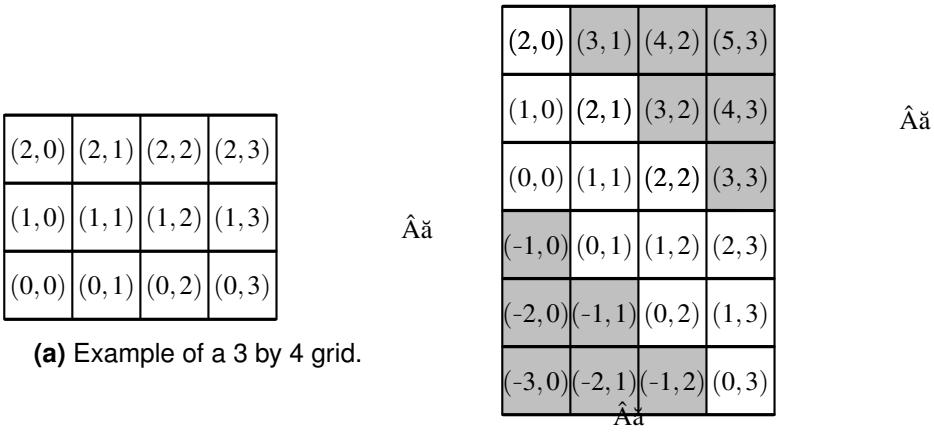
```
def check_rows(grid):
    for row in grid:
        t = check_row(row)  # check individual row
        if t:
            return t
    return False
```

The function `check_row` takes a single row of the grid as input, and returns the token that occupies at least four continuous adjacent positions in the row. If no such token exists, `False` is returned.

Provide an implementation of the function `check_row`. You may assume that at most only one token will satisfy the condition and tokens will not be `False` when evaluated as a boolean. [4 marks]

It is also straightforward to check for any continuous vertical connections. We can simply transpose or rotate the grid like a matrix and since the columns now become the rows, the `check_rows` function can be reused. However, it is not so trivial to check the diagonals.

Effy thinks there is a way to reuse the `check_rows` function if we can somehow skew the grid by 45 degrees, as shown in Figure 2



(a) Example of a 3 by 4 grid.

(b) Skewed by 45 degrees.

Figure 2: A grid skewed vertically 45 degrees will result in the diagonals becoming rows. The shaded cells indicates cells that are not part of the original grid.

Diagonal. Effy has found an ancient matrix function `diagonal(height, width, i, j)` which returns a matrix of the given height and width as a list of lists. The elements of the matrix are diagonally running coordinates with (i, j) at the origin. More specifically, the elements are `(x, y)` tuples where $x = i + row + col$ and $y = j + col$, *row* and *col* is the current row and column of the element, and i, j are inputs to the function.

For example:

```
>>> diagonal(6, 4, -3, 0)
[[( 2, 0), ( 3, 1), ( 4, 2), (5, 3)],   # output formatted
 [( 1, 0), ( 2, 1), ( 3, 2), (4, 3)],   # for readability
 [( 0, 0), ( 1, 1), ( 2, 2), (3, 3)],
 [(-1, 0), ( 0, 1), ( 1, 2), (2, 3)],
 [(-2, 0), (-1, 1), ( 0, 2), (1, 3)],
 [(-3, 0), (-2, 1), (-1, 2), (0, 3)]]
```

Deep Map. In order to complete this task, Effy needs another ancient function `deep_map`. `deep_map(fn, lst)` takes as inputs a function and a list of nested lists. It **modifies all elements** and nested elements of `lst` by applying the function `fn` to each element.

For example:

```
>>> l = [1, 2, [3, 4], [5, 6, [7]]]
>>> deep_map(lambda x: x*2, l)
>>> l
[2, 4, [6, 8], [10, 12, [14]]]
```

D. Effy cannot find or recall the implementation of `deep_map`. Please help her by providing an implementation of `deep_map`. [6 marks]

E. **[Challenging]** The function `skew(grid)` returns a new grid that is a skewed form of the input grid. In other words, one of the two diagonals of grid has become the rows. Provide an implementation for `skew`. You may use the functions `diagonal` and `deep_map`, as well as any of the matrix functions defined in this questions. To keep the game grid valid you may fill the new cells with `None`. [6 marks]

F. Finally, Effy can complete her function that determines if a given grid has a winner.

```
1 def winner(grid):
2     t1 = check_rows(grid)
3     rotate(grid)    # swap cols to rows
4     t2 = check_rows(grid)
5     r = skew(grid)  # get a skewed representation
6     t3 = check_rows(r)
7     rotate(r)       # swap cols to rows
8     t4 = check_rows(r)
9     return t1 or t2 or t3 or t4  # return token if any is not False
```

Her function is supposed to return the token of the winner, if there is a winner, and `False` if there are no winners. However, there are at least two problems with her function.

State the two problems and explain or show how each can be corrected. The line numbers have been shown for you to use as reference. [4 marks]

Question 3: Containers Revisited [24 marks]

The containers that we designed in the midterm test were of poor quality because we were limited to using tuples. Now, we will instead use a Python dictionary to model a container.

A container is simply a dictionary, with the keys representing the items and the values are the total weight of each item stored in the container. For example, a container containing 5 tons of bananas and 3 tons of apples could be `{"banana": 5, "apples": 3}`.

Container is supported by the following functions:

- `make_container()` returns an empty container.
- `add_item(container, item, weight)` takes as input a container, an item and a weight (which is an integer), and updates container with the added weight of the given item.
- `remove_item(container, item, weight)` takes as input a container, an item and a weight, and updates the container with weight amount of item removed from it. If the container does not contain item, then nothing is changed.
- `get_items(container)` takes as input a container, and returns a tuple of items that is in the container.
- `get_weight(container)` takes as input a container, and returns the total weight of all the items in it. Note that we assume the container has no weight.

Sample execution:

```
>>> c = make_container()
>>> add_item(c, 'bananas', 5)
>>> add_item(c, 'apples', 3)
>>> add_item(c, 'bananas', 5)
>>> c
{'bananas': 10, 'apples': 3}

>>> get_items(c)
('bananas', 'apples')
>>> get_weight(c)
13

>>> remove_item(c, 'bananas', 3)  # removes 3 tons of bananas
>>> c
{'bananas': 7, 'apples': 3}  # 7 tons of bananas left

>>> remove_item(c, 'bananas', 10)  # removes more bananas than available
>>> get_items(c)  # only apples left
('apples',)
>>> get_weight(c)  # weight is 3 tons
3
```

A. Assuming the function `make_container` has been defined, provide an implementation for the rest of the above-mentioned functions. [8 marks]

B. It is useful to be able to transfer the contents of several containers into another container. The function `transfer_items(container, *containers)` takes an arbitrary number of containers and transfers the contents of all the containers into the first container. After the transfer, only the first container will contain all the items, and the rest of the containers will be empty.

Example:

```
>>> c1 = {"bananas": 5, "apples": 3}
>>> c2 = {"bananas": 2, "oranges": 7}
>>> c3 = {"apples": 6, "pears": 4}
>>> transfer_items(c1, c2, c3)
>>> c1
{'bananas': 7, 'apples': 9, 'oranges': 7, 'pears': 4}

>>> get_weight(c2)
0
>>> get_weight(c3)
0
```

Provide an implementation of the function `transfer_items`. [4 marks]

C. To stay true to data abstraction, the items that can be loaded in the containers are not restricted to strings. It is possible to load classes and objects as items into a container.

Jia Zhen wants to add mooncakes that can be instanced from a `Food` class and tries the following code:

```
>>> c = make_container()
>>> add_item(c, Food("Mooncake"), 1)  # adds 1 ton of mooncake
>>> add_item(c, Food("Mooncake"), 1)  # adds another 1 ton of mooncake
>>> c
{<Food object at 0x0828>: 1, <Food object at 0x92B0>: 1}
```

She expected to the container to contain two tons of a single mooncake item but she gets two 1-ton items instead. Explain why this happens and suggest a way that she can get what she wants. [4 marks]

D. Gerrie is still determined to lock these containers with her locking mechanism, having acknowledged that it is better to lock them after loading them onto the ship. Since dictionaries are mutable, she can now truly lock the containers by mutating them rather than returning a new locked container and has modified her code as shown:

```
def lock(container, passcode):
    container = lambda x: container if x == passcode else False

def unlock(container, passcode):
    if container(passcode):
        container = container(passcode)
```

However, her code does not work when she tries it out:

```
>>> c = {"bananas": 10, "oranges": 5}
>>> lock(c, 12345)
>>> c
{"bananas": 10, "oranges": 5}
```

Explain what is wrong with Gerrie's code. [2 marks]

E. **[Challenging]** Present a working implementation for the `lock` and `unlock` functions. You may also suggest how the container representation can be modified in order to work with your implementation. [6 marks]

Question 4: Airplanes [14 marks]

Consider the following implementation where we model airplanes using Python classes:

```
1  class Airplane:
2      def __init__(self, weight, mtow):
3          self.weight = weight
4          self.mtow = mtow
5
6      def get_weight(self):
7          return self.weight
8
9      def take_off(self):
10         if self.weight > self.mtow:
11             return 'Too heavy to take off'
12         else:
13             return 'Ok to take off'
14
15 class PassengerPlane(Airplane):
16     def __init__(self, weight, mtow):
17         super().__init__(weight, mtow)
18         self.passengers = 0
19
20     def get_weight(self):
21         return self.passengers * 100 + self.weight
```

An `Airplane` is initialized with two properties: its unladen weight and its maximum take-off weight (MTOW). A plane cannot take off if its weight is above its MTOW.

A `PassengerPlane` is an `Airplane` that can take on passengers. It is assumed that each passenger will add an additional 100 units to the plane's weight.

A. Elsa thinks there must be a mistake in the code as a `PassengePlane` is not able to fly because it does not have the method `take-off` defined. Do you agree with her? If yes, please show how this can be corrected. If no, please explain why. [3 marks]

B. Having her concerns satisfied, Elsa then tries out the following code:

```
>>> plane = PassengerPlane(1000, 2000)
>>> plane.passengers =  11  # 11 passengers add additional 1100 weight
>>> plane.take_off()
'Ok to take off'
```

There must be some error since the plane should be too heavy to take off. **Explain** the error in the code and **propose** a fix. [3 marks]

C. A `CargoPlane` is an `Airplane` that can carry `Cargo`. `Cargo` is a class that has a property `weight`, which represents the weight of the cargo.

Provide an implementation for the class `CargoPlane`. A `CargoPlane` should be able to contain an arbitrary number of `Cargo` objects and its weight computed for take-off should include the weight of all the `Cargo` it contains. Your implementation should not have excessive methods and properties, and you do not need a getter or setter to load the cargo.

[4 marks]

D. Elsa wants to design a plane that is able to carry both passengers and cargo. By taking advantage of Python multiple inheritance, she thinks all she has to do is subclass from both `PassengerPlane` and `CargoPlane`, implements her `PassengerCargoPlane` as follows:

```
class PassengerCargoPlane(PassengerPlane, CargoPlane):
    pass
```

She reasons that since both `PassengerPlane` and `CargoPlane` already has all the methods and properties she needs, there is nothing else she has to do but `pass`.

Do you think her idea is correct? If yes, carefully examine the code for all the classes, and **identify and fix** any bugs that will hinder Elsa's implementation. If you have fixed the bugs earlier in Part A, simply mention it.

If no, provide a working implementation with the minimal methods needed. You can also suggest modifications to the code of the other classes as needed. [4 marks]

Question 5: 42 and the Meaning of Life [4 marks]

Either: (a) explain how you think some of what you have learnt in CS1010S will be helpful for you for the rest of your life and/or studies at NUS; or (b) tell us an interesting story about your experience with CS1010S this semester. [4 marks]

— END OF PAPER —

Appendix

Parts of the Python documentation is given here for your reference.

List Methods

- `list.append(x)` Add an item to the end of the list.
- `list.extend(iterable)` Extend the list by appending all the items from the iterable.
- `list.insert(i, x)` Insert an item at a given position.
- `list.remove(x)` Remove the first item from the list whose value is *x*. It is an error if there is no such item.
- `list.pop([i])` Remove the item at the given position in the list, and return it. If no index is specified, removes and returns the last item in the list.
- `list.clear()` Remove all items from the list
- `list.index(x)` Return zero-based index in the list of the first item whose value is *x*. Raises a `ValueError` if there is no such item.
- `list.count(x)` Return the number of times x appears in the list.
- `list.sort(key=None, reverse=False)` Sort the items of the list in place.
- `list.reverse()` Reverse the elements of the list in place.
- `list.copy()` Return a shallow copy of the list.

Dictionary Methods

- `dict.clear()` Remove all items from the dictionary.
- `dict.copy()` Return a shallow copy of the dictionary.
- `dict.items()` Return a new view of the dictionary's items (`(key, value)` pairs).
- `dict.keys()` Return a new view of the dictionary's keys.
- `dict.pop(key[, default])` If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.
- `dict.update([other])` Update the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.
- `dict.values()` Return a new view of the dictionary's values.