NATIONAL UNIVERSITY OF SINGAPORE

CS1010S—Programming Methodology

2017/2018 Semester 2

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 **TWELVE (12) pages** including this cover page.
3. Weightage of each question 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.

```
n = 4
while n < 10:
    n += 1
    if n % 2 == 0:
        continue
    if n % 3 == 0:
        break
print(n)
```

[5 marks]

B.

```
phrase = "Zzzz foolish deeds"
for i in range(len(phrase)):
    if phrase[i] == phrase[i+1]:
        print(i)
```

[5 marks]

C.

```
a = [0, 1, 2]
a.append(a)
b = [a[0]+a[1], a[1:2], a[3][3][2]]
print(b)
```

[5 marks]

D.

```
def foo(x):
    return lambda y: bar(x) if y % 2 else x
def bar(y):
    return lambda x: foo(x) if y % 2 else y
print(foo(2)(3)(4))
```

[5 marks]

E.

```
def force(x):
    try:
        return int(x)
    except ValueError:
        return float(x)
    except Exception:
        return "NaN"
print(force("100"))
print(force("1.0"))
print(force("abc"))
```

[5 marks]

F.

```
def boo(s):
    if s:
        return boo(s[1::2]) + s[0]
    return s
print(boo("banana"))
```

[5 marks]

Question 2: Jenga [30 marks]

Jenga is a game of physical skill created by Leslie Scott, and currently marketed by Hasbro. Players take turns removing one block at a time from a tower constructed of 54 blocks. Each block removed is then placed on top of the tower, creating a progressively taller and more unstable structure. *— Source: Wikipedia*

For this question, we will use Python list to model the state of a Jenga tower.

A Jenga tower is made up of several levels of blocks, stacked orthogonal (90 degrees) to each other. Each layer is made up of n blocks placed side by side. The original game of Jenga uses $n = 3$, but we keep it general in our model.

A layer is represented by a `list` of n elements, and each element is a `bool` where `True` represents a block is present at the given position and `False` otherwise. For example, a layer of $n = 4$ with only blocks at the ends is represented as `[True, False, False, True]`.

A. Implement the function `new_layer` which takes as input n, the maximum number of blocks for the *layer*, and returns our list representation of a *layer* with no blocks. [2 marks]

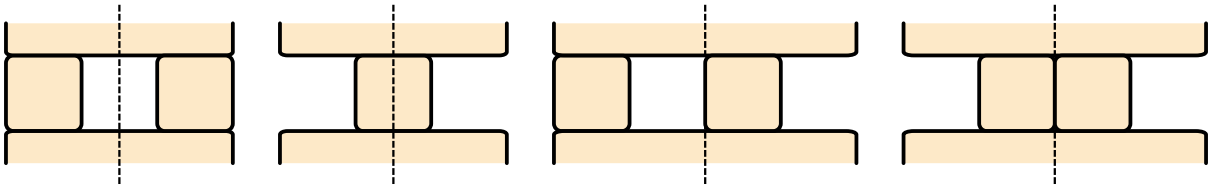
B. The function `add_block` takes as input a *layer* and adds a new block to an unoccupied position in the *layer*. If there are no available positions, i.e., all positions already have an existing block, then `LayerFilledException` is raised.

Implement the function `add_block` and define the exception `LayerFilledException`. The new block may be inserted into any available position you choose. [6 marks]

C. Implement the function `remove_block`, which takes as input a *layer* and a position, and removes the block in the given position from the *layer*. You may assume that position starts from 0, and a valid position with a block will always be given. [2 marks]

D. A layer is said to be stable if it has sufficient blocks arranged in a way to prevent it from collapsing. More specifically, there needs to be at least one block on each side from the centre of the layer.

The following illustrations show stable layers of $n = 3$ and $n = 4$.



Note that when n is odd, a single block in the centre position is sufficient for it to be stable.

Implement the function `is_stable` which takes as input a *layer*, and returns `True` if the *layer* is stable, and `False` otherwise. [6 marks]

E. Now we are ready to construct our tower. A tower is simply a list of levels, with the bottommost level at index 0.

Without breaking the abstraction of a *layer*, implement the function `new_tower` that takes as input a height and n, and returns a *tower* of the given height with each layer completely filled with n blocks. [5 marks]

F. With our newly constructed tower, we can now play Jenga!

In each turn, a block is removed from one of the layers in the tower and placed on the top layer. If the top layer is filled, then a new empty layer is created on top of the tower and the block is placed in the new layer. However, if removing the block causes the layer to be unstable, the entire tower will collapse and the game will end.

Implement the function `play` which takes as inputs the tower, the level of the layer to remove and the position of the block within the layer. It returns the string `'Game Over!'` if the game ends, otherwise the tower will be update to reflect the new state as described above.

You may assume that the function `width(layer)` has been defined that will return n, the maximum number of blocks of the layer, and that the bottommost layer of the tower is level 0. There is no need to model the "collapsed" state of the tower. [5 marks]

G. Jerryl thinks it is a waste of space to use a list to represent each layer. He proposes to label each position in a layer with consecutive numbers from 1 to n, and represent the state of the layer by adding the numbers of positions with blocks.

For example, a layer represented as `[True, False, True]` has blocks numbered 1 and 3. So in Jerryl's representation, it will be an integer 4. Thus, a tower of $n = 3$ with 5 levels of filled blocks will be represented as `[6, 6, 6, 6, 6]`.

What is the flaw in Jerryl's reasoning? Propose a correct way to use integers to represent a layer. [4 marks]

Question 3: T9 [20 marks]

T9, which stands for Text on 9 keys, is a U.S.-patented predictive text technology for mobile phones (specifically those that contain a 3x4 numeric keypad), originally developed by Tegic Communications, now part of Nuance Communications. — *Source: Wikipedia.*

Old telephone keypads have a numerical keypad layout as shown below, where every letter is associated with a number (0 represents space, and the * and # keys are ignored):

Thus, a phrase like "I LUV U" can be entered by pressing the associated number key for each character, which is "4058808".

We can represent the keypad in two ways:

1. a list where each element is a string of characters that are associated with a number which is the element's index. The keypad above is represented as:

```
keys = [' ', '', 'abc', 'def', 'ghi', 'jkl', 'mno', 'pqrs', 'tuv', 'wxyz']
```

2. a dictionary where the keys are the character and the values begin their associated number. The keypad above is represented as:

```
keyd = {'a': 2, 'b': 2, 'c': 2, 'd': 3, 'e': 3, 'f': 3,
        'g': 4, 'h': 4, 'i': 4, 'j': 5, 'k': 5, 'l': 5,
        'm': 6, 'n': 6, 'o': 6, 'p': 7, 'q': 7, 'r': 7, 's': 7,
        't': 8, 'u': 8, 'v': 8, 'w': 9, 'x': 9, 'y': 9, 'z': 9, ' ': 0}
```

Note: Your implementations should not assume a fixed number of keys or letters.

A. Suppose you are only given a list following representation 1 above. Implement the function `to_dict` which takes in such list, and returns the dictionary representation (representation 2 above) of the input keys.

In other words, `to_dict(keys) == keyd` is `True`.

Hint: Recall that strings, tuples and lists are all sequences. [4 marks]

B. Now suppose you are only given a dictionary following representation 2 above. Implement the function `to_keys` which takes in such a dictionary, and returns the list representation (representation 1 above) of the input keys.

In other words, `to_keys(keyd) == keys` is `True`.

Hint: Recall the built-in function `max` *returns the largest value in a sequence.* [4 marks]

C. The function `to_nums` takes as input a string, and returns an integer that represents the numbers to be pressed to input the string. You may assume that `keys` and `keyd` have already been defined as above, and that the input string only consist of lowercase letters. [4 marks]

Sample:

```
>>> to_nums("i luv u")
4058808
```

D. Finding the text from an input integer is not easy because there are many combinations of letters for a given integer.

The function `to_letters` takes as input an integer, and returns a list of all possible combinations of letters that can be represented by the numbers on a keypad. Again, you may assume that `keys` and `keyd` have already been defined and only need to be concerned about lowercase letters. [6 marks]

Sample:

```
>>> to_letters(293)
['awd', 'awe', 'awf', 'axd', 'axe', 'axf', 'ayd', 'aye', 'ayf', 'azd',
 'aze', 'azf', 'bwd', 'bwe', 'bwf', 'bxd', 'bxe', 'bxf', 'byd', 'bye',
 'byf', 'bzd', 'bze', 'bzf', 'cwd', 'cwe', 'cwf', 'cxd', 'cxe', 'cxf',
 'cyd', 'cye', 'cyf', 'czd', 'cze', 'czf']
```

E. Which of the two representations (list and dictionary) is the best? Explain in terms of time and space efficiency. [2 marks]

Question 4: ÜberJiak [16 marks]

ÜberJiak runs a delivery service with a fleet of different vehicles. Its business is in the delivery of food, so it is important to deliver in a timely manner. Thus, each of its mode of delivery vehicles has a maximum distance of which an order can be delivered.

Consider the following implementation where we model the delivery vehicles of ÜberJiak.

```
1  class Vehicle:
2      def __init__(self, reach):
3          self.reach = reach
4          self.orders = []
5
6      def deliver(self, order):
7          if self.reach < order.distance:
8              print("Too far")
9              return False
10         else:
11             self.orders.append(order)
12             print("Order delivered")
13             return True
14
15
16 class Motorcycle(Vehicle):
17     def __init__(self, fuel):
18         self.fuel = fuel
19         self.orders = []
20
21     def top_up(self, amount):  # top up fuel
22         self.fuel += amount
23
24     def deliver(self, order):
25         if self.fuel < order.distance:  # fuel units is in distance
26             print("Not enough fuel")
27             return False
28         else:
29             self.orders.append(order)
30             print("Order delivered")
31             return True
```

A `Vehicle` is initialized with a property reach, and can only deliver orders which distance is within its reach. It also records a history of all orders delivered.

One of their mode of delivery is using `Motorcycle` which is also a `Vehicle`. Motorcycle consumes fuel, which adds another constrain to the distance the motorcycle can cover. For simplicity, fuel is represented as the distance that can be travelled with the available fuel.

A. Rachel realizes something is not right when using her `Motorcycle` for delivery. Somehow the deliveries are limited only by the amount of fuel in the motorcycle, and her motorcycle never runs out of fuel.

The right behaviour is that the orders that can be delivered are limited by **both** the fuel and the reach of the motorcycle, and the fuel should decrease by the order's distance for every delivery.

i) **Explain** why the code fails and ii) **propose** replacement code to fix the mistakes. You should use OOP concepts as much as possible.

You may use the line numbers given as references for suggest edits to the code. [6 marks]

B. Another mode of delivery ÜberJiak has is `Bicycle`, which is also a `Vehicle` and initialized with the same inputs.

`Bicycle` behaves like a `Vehicle` except that after each delivery, the rider will be thirsty and has to call a method `.hydrate()` before it can do another delivery. Otherwise, if not hydrated, calling the method `.deliver(order)` will print `"Hydration needed"` and return `False`.

Example, assume class `Order` is initialised with distance.

```
>>> bicycle = Bicycle(5)       # only 5km reach
>>> bicycle.deliver(Order(3))  # deliver order 3km away
'Order delivered'
True

>>> bicycle.deliver(Order(5))  # deliver 5km away
'Hydration needed'
False

>>> bicycle.deliver(Order(8))  # deliver 8km away
'Hydration needed'
False

>>> bicycle.hydrate()
>>> bicycle.deliver(Order(5))  # deliver 5km away
'Order delivered'
True
```

Provide an implementation for the class `Bicycle`. [6 marks]

C. Yet another mode of delivery is the `EScooter`, which behaves both like a bicycle and a motorcycle, in that it requires fuel and for the rider to hydrate between deliveries.

Using the fixed version of `Motorcycle`, provide a minimal implementation for `EScooter`.

Note that you will be penalized for any unnecessary and redundant code. [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]

— E N D O F P A P E R —

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.