National University of Singapore
School of Computing
CS1010S: Programming Methodology

Mock Finals

You have only **2 hours** to solve all **FIVE (5) questions**.
The maximum attainable score is **100**. Good luck!

Prepared by Russell Saerang.

Question 1: Python Expressions [30 marks]

There are several parts to this problem. Answer each part **independently and separately**. In each part, the Python snippet is entered into a Python script and then run. Determine the response printed by the interpreter (Python shell) and **write the exact output**. If the interpreter produces an error message, or enters an infinite loop, explain why and **clearly state the responsible evaluation step**.

A.
```
a = [0, 1]
b = [[1], [[2]]]
b.extend(a)
print(b)
a.append(a)
print(b)
```

[5 marks]

B.
```
d, msp = {}, "mississippi"
for x in msp:
    if x not in d:
        d[x] = 1
    else:
        d[x] += 1
        d[d[x]] = 1
print(d)
```

[5 marks]

C.
```
kh, z = "kenghwee", "zoink"
sus = 0
for s in kh:
    if s in z:
        sus += 1
    if s == "e":
        print(sus)
    elif s > "k":
        sus *= 2
print(sus)
```

[5 marks]

D.
```
def debug(s):
    try:
        return s + 123
    except ValueError:
        return str(s) + "123"
    except Exception:
        return "F"
    except:
        return "FF"
for q in ["Gday", 42, [0]]:
    print(debug(q))
```

[5 marks]

E.
```
def bar(z):
    return lambda r: baz(r - 1) \
            if r > 5 else r + z
def baz(r):
    return lambda z: bar(z // 2) \
            if z > 0 else 5
print(baz(4)(100)(4))
```

[5 marks]

F.
```
def p(p, q):
    print(p)
    return q(p + q(p))
def q(q):
    print(q)
    return q + q
print(p(2, q))
```

[5 marks]

Question 2: Plagiarism [28 marks]

Ben Bitdiddle is back as a CS1010Z lecturer and he decided to investigate how the plagiarism checker works in the module!

Suppose we represent an assessment's list of submissions with a list of answer data. Each answer data is represented as a `list` containing the **name** of the answer submitter, the **time** of submission (`str`, in HHMM format), and the **answer** (`str`) itself.
For example, suppose a question is answered by six people:

```
answers = [
    ["Russell", "0930", "recursion"], ["Keng Hwee", "1020", "recursive"],
    ["Jonathan", "1021", "recurse"], ["Wei Han", "1100", "recursion"],
    ["Gerald", "1104", "recursion tree"], ["Hoi Yin", "1216", "recurse"]
]
```

A. **[Warm-up]** One of the requirements for the plagiarism checker is the time difference between two submissions. Implement the function `minutes(time)` that takes in a **string** (`str`) and returns the number of minutes that has passed since the first minute of the day. For example, `minutes("0912")` returns `552`.

[2 marks]

B. Implement the function `get_answer(answers, name)` that takes as input an answers list (`list` of `list`), and a **name** (`str`). Return **name**'s answer on the answers list. If **name** is not found, return `None`.
Sample execution:

```
>>> get_answer(answers, "Keng Hwee")
'recursive'
>>> get_answer(answers, "Gerald")
'recursion tree'
>>> get_answer(answers, "Pak")
None
```

[2 marks]

C. Of course, it is possible for another students to submit their answer for the first time. Implement the function `submit_answer(answers, name, time, answer)` that takes as input an answers list (`list` of `list`), a **name** (`str`), the **time** (`str`), and the **answer** (`str`). Modify the original answers list by adding the answer data to it.
You may assume the student has never submitted any answer before.
Sample execution:

```
>>> submit_answer(answers, "Pak", "1247", "recursing")
>>> answers
[['Russell', '0930', 'recursion'], ['Keng Hwee', '1020', 'recursive'],
 ['Jonathan', '1021', 'recurse'], ['Wei Han', '1100', 'recursion'],
 ['Gerald', '1104', 'recursion tree'], ['Hoi Yin', '1216', 'recurse'],
 ['Pak', '1247', 'recursing']]
```

[4 marks]

D. Since we can implement a function to submit an answer, why not resubmitting an answer?

Implement the function `resubmit_answer(answers, name, time, answer)` that takes as input an answers list (`list` of `list`), a **name** (`str`), the new **time** (`str`), and the new **answer** (`str`). Modify the original answers list by modifying the correct answer data. You may assume the student has already submitted an answer before.
Sample execution:

```
>>> resubmit_answer(answers, "Russell", "1316", "iterative")
>>> answers
[['Russell', '1316', 'iterative'], ['Keng Hwee', '1020', 'recursive'],
 ['Jonathan', '1021', 'recurse'], ['Wei Han', '1100', 'recursion'],
 ['Gerald', '1104', 'recursion tree'], ['Hoi Yin', '1216', 'recurse']]
```

[4 marks]

E. Now that we have handled incoming submissions, it's time for us to handle the current submissions. For now, CS1010Z's plagiarism technology is quite simple and therefore we are going to build it from scratch.
First, implement the function `similarity_score(str1, str2)` that calculates the similarity score between two strings as follows:

- Ignore spacings on both strings, meaning `"bob cat"` is treated the same way as `"bobcat"`. This step will produce two new strings.
- If the new `str1` is longer than the new `str2`, then return the proportion of letters of the new `str2` that are in the same position as the new `str1`. Otherwise return the proportion of letters of the new `str1` that are in the same order as the new `str2`.

You may assume both strings are in low-capitalized.
Sample execution:

```
>>> similarity_score("iteration", "iterative")
0.7777777777777778
>>> similarity_score("iteration is cool", "iterator is fun")
0.46153846153846156 # 6 out of 13 matches ("iterat")
```

[6 marks]

F. After implementing the similarity scoring, we are just a method away from determining the suspects! **Using the previously defined functions**, implement a function `blame(answers, name1, name2)` that takes in two **distinct names inside the answers list** and verifies whether both names are involved in plagiarism. Return `True` if the time difference between both submissions is less than an hour and the similarity score on the answers is at least 0.8, otherwise return `False`.
You will be heavily penalized if you do not use the function defined in the previous parts but you could have.
Sample execution:

```
>>> blame(answers, "Russell", "Wei Han")
False
>>> blame(answers, "Keng Hwee", "Jonathan")
True
```

[6 marks]

G. Ben Bitdiddle finally figured out what's wrong with the current implementation of the plagiarism checker. Suppose Jonathan decides to resubmit the exact same answer again:

```
>>> blame(answers, "Jonathan", "Keng Hwee")
True
>>> resubmit_answer(answers, "Jonathan", "1700", \
        get_answer(answers, "Jonathan"))
>>> blame(answers, "Jonathan", "Keng Hwee")
False
```

This will implicitly free Keng Hwee from the plagiarism blaming, which of course does not make any sense.
Suggest an improvement on the representation such that regardless of resubmission, a student is still flagged for plagiarism due to his/her past answers.

[4 marks]

Question 3: Terminal Trauma [21 marks]

After looking at the CS1010S Trauma Support Group chat, you became bored and decided to open your terminal. However, you began to realize that dictionaries are haunting you inside it.

Suppose we model a terminal's home directory with a dictionary of dictionaries where the keys are either file names or directory names and the values are **`None`** if the key is a file name and a dictionary otherwise, which may contain more files and directories.
For example, the home directory below contains one file and two directories containing more files.

```
>>> home = {
        'a.py': None,
        'b': {
            'd.py': None,
            'e.py': None,
        },
        'c': {
            'f.py': None,
            'g': {
                'i.py': None,
                'j.java': None
            },
            'h': {} # empty directory
        }
    }
```

A. One of the most useful command inside the terminal is **`cd`**, where you can **c**hange your current working **d**irectory from one to another.
Implement the function `cd(directory, path)` that takes in a dictionary **`directory`** representing a directory and a string **`path`** representing the path where you want to go.
Note that **`path`** is a series of directories that you want to go to, separated by a **`"/"`** and always ends with a **`"."`** to indicate the end of traversal.
You may assume that the given input is a valid path.

Sample execution:

```
>>> cd(home, "c/.")
{'f.py': None, 'g': {'i.py': None, 'j.java': None}, 'h': {}}
>>> cd(home['b'], ".")
{'d.py': None, 'e.py': None}
>>> cd(home, "c/g/.")
{'i.py': None, 'j.java': None}
```

[4 marks]

B. Explain the order of growth in terms of **time** of the function `cd(directory, path)` that you have implemented previously.

[2 marks]

C. Besides changing your current working directory, you can rename a file as well. Implement the function `rename(directory, path, old_name, new_name)` that takes in a dictionary **`directory`** representing a directory, a string **`path`** representing the path, and two strings **`old_name`** and **`new_name`** for the files that you want its name to change. Simply rename the file name into the new name by modifying the directory itself. You may assume that the given input is a valid path.

Sample execution:

```
>>> rename(home, "c/g/.", "i.py", "j.java")
>>> cd(home, "c/g/.")
{'i.py': None, 'j.java': None} # nothing happens
>>> rename(home, "c/g/.", "i.py", "k.py")
>>> cd(home, "c/g/.")
{'j.java': None, 'k.py': None}
>>> rename(home, "c/g/.", "i.py", "l.py")
>>> cd(home, "c/g/.")
{'j.java': None, 'k.py': None} # i.py doesn't exist anymore
```

[5 marks]

D. Implement the function `copy(directory, path, new_path)` that takes in a dictionary **`directory`** representing a directory, a string **`path`** representing the path, and a string **`new_path`** for the new path. Copy all the contents of the old directory to the new directory.
You may assume the new directory is never contained inside the old directory. There's a reason for this assumption and you may ponder about it for the bonus part.

Sample execution:

```
>>> copy(home, "c/g/.", ".")
>>> home
{
    'a.py': None,
    'b': {
        'd.py': None,
        'e.py': None
    },
    'c': {
        'f.py': None,
        'g': {
            'i.py': None,
            'j.java': None
        },
        'h': {}
    },
    # It's copied!
    'i.py': None, 'j.java': None
}
# BONUS PART
>>> copy(home, ".", "c/.") # what will happen?
```

[5 + 2 marks]

E. Implement the function `move(directory, path, new_path)` that takes in a dictionary **`directory`** representing a directory, a string **`path`** representing the path, and a string **`new_path`** for the new path. Move all the contents of the old directory to the new directory.
Similarly, you may assume the new directory is never a proper subset of the old directory and that the given input is a valid path.

Sample execution:

```
>>> cd(home, "c/.")
{'f.py': None, 'g': {'i.py': None, 'j.java': None}, 'h': {}}
>>> move(home, "c/g/.", ".")
>>> home
{
    'a.py': None,
    'b': {
        'd.py': None,
        'e.py': None
    },
    'c': {
        'f.py': None,
        'g': {}, # g is now an empty directory
        'h': {}
    },
    # It's moved!
    'i.py': None,
    'j.java': None
}
>>> move(home, ".", ".")
>>> home # nothing changes
{
    'a.py': None,
    'b': {
        'd.py': None,
        'e.py': None
    },
    'c': {
        'f.py': None,
        'g': {},
        'h': {}
    },
    'i.py': None,
    'j.java': None
}
>>> move(home, "c/.", "c/.") # similarly, nothing should happen
>>> cd(home, "c/.")
{'f.py': None, 'g': {}, 'h': {}}
```

[5 marks]

Question 4: Cells at Work! [17 marks]

Consider the following classes.

```
class Cell:
    def __init__(self, health, damage):
        self.health = min(100, health)
        self.damage = damage
        self.dead = False

    def regenerate(self):
        self.health = min(100, self.health + 20)

class RedBloodCell(Cell):
    def __init__(self, health, damage, utility):
        super().__init__(health, damage)
        self.utility = utility

    def regenerate(self):
        self.health = min(100, self.health + 30 * self.utility)
```

A. What will be output of executing the following lines:

```
>>> blood = RedBloodCell(40, 0, 1.3)
>>> blood.health

>>> blood.regenerate()
>>> blood.health

>>> blood.attack(Cell(100, 1))
>>> blood.regenerate()
>>> blood.health
```

[2 marks]

B. Wilson decides to create a dead red blood cell as follows. However, he finds something weird because dead cells cannot regenerate anymore.

```
>>> dead_blood = RedBloodCell(0, 0, 1.5)
>>> dead_blood.health
0
>>> dead_blood.regenerate()
>>> dead_blood.health
45.0
```

Explain why does the dead cell still regenerate and modify the code above to fix this issue according to OOP principles, i.e. redundant code will be penalised.

[4 marks]

C. A `WhiteBloodCell` is also a special type of cell. Constructed by only its damage and utility, it is always created with 30 health and it regenerates just like a red blood cell but everytime it does, its utility increases by 10% of its original utility. Moreover, it can attack cells other than itself as long as its own health is at least 50.

- If the attack is successful, the target cell's health will be decreased by **the product between the white blood cell's damage and utility**. Additionally, if the target cell is a red blood cell, its utility will be decreased by 30% of the original utility.
- If the target cell's health is less than the damage taken, it simply dies.
- Finally, if the attack is a success, print **`"Cell attacked!"`**. Otherwise, print **`"Cell needs to regenerate..."`**.

Sample execution:

```
>>> U1146 = WhiteBloodCell(40, 1.5)
>>> ordinary = Cell(30, 1)
>>> U1146.attack(U1146)      # nothing happens
>>> U1146.attack(ordinary)
'Cell needs to regenerate...'
>>> ordinary.health
30                           # ordinary is still in full health
>>> U1146.regenerate()
>>> U1146.attack(ordinary)
'Cell attacked!'
>>> ordinary.health
0                            # ordinary is dead

>>> U1147 = WhiteBloodCell(40, 1.5)
>>> AE3803 = RedBloodCell(80, 0, 1.2)
>>> U1147.regenerate()       # utility is now 1.65
>>> U1147.attack(AE3803)
'Cell attacked!'
>>> AE3803.health
14.0                         # 80 - 40 * 1.65
>>> AE3803.utility
0.84                         # 1.2 * 0.7
```

Provide an implementation of the class **`WhiteBloodCell`** using OOP principles. Redundant code or methods will be penalised.

[6 marks]

D. Right after implementing a white blood cell, Wilson decides to implement the class **`CancerCell`** as follows:

```
class CancerCell(RedBloodCell, WhiteBloodCell):
    pass
```

Now he decides to try the implementation out with a few tests.

```
>>> cancer = CancerCell(100, 1000, 1.7)
>>> skin = Cell(50, 1)
>>> cancer.attack(skin)
'Cell needs to regenerate...'
```

Something's wrong! The cancer cell is supposed to even kill the skin cell. Explain why the mistake happens and suggest a fix by modifying the code based on OOP principles.

[5 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;

(b) tell us an interesting story about your experience with CS1010S this semester;

(c) share how the COVID 19 situation has positively influenced/impacted your learning in CS1010S;

(d) **how the CS1010S Trauma Support Group has been full of memes**.

Remark

The marking scheme is always like this.

The student will be awarded points as long as he/she is coherent and doesn't say something obviously wrong.

This question is designed for students who get stuck with the other questions. They can at least spend some time writing an essay for marks. Points will be awarded for effort.

Make the prof laugh out loud and you get full marks.

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.