CS1010E: Programming Methodology

PE2 (2020/2021 Sem 1)

Files

- `PE2.pdf`
- `Question1.py`
- `Question2.py`
- `Question3.py`

Coursemology

- `Past PE2 > CS1010E 2020/21 Sem 1`

Questions

Question 1: Parentheses

We use a lot of arithmetic expressions every day and there are always parentheses (*i.e.*, the symbol `"("` and `")"`) in them. We want to do two things:

1. Extract both open `"("` and close `")"` parentheses from a given string (`str`).
2. Check if the parentheses are balanced according to normal mathematical conventions.

General Restrictions

You are NOT allowed to use the the following Python ***built-in*** string (`str`) or list (`list`) methods/functions in Question 1:

- `encode()`
- `decode()`
- `replace()`
- `partition()`
- `rpartition()`
- `chr()`
- `split()`
- `rsplit()`
- `translate()`
- `map()`
- `count()`
- `ord()`
- `find()`
- `join()`
- `rfind()`
- `rsplit()`
- `rstrip()`
- `index()`
- `strip()`
- `sort()`
- `sorted()`
- `pop()`

continue on the next page...

1.1 Iterative Parentheses [10 marks]

Question

Write the *iterative* function `extract_parentheses_I(stmt)` to extract all the parentheses from statement `stmt` (`str`) and returns a new string (`str`) of parentheses. You do not have to care if the statement `stmt` is a correct arithmetic expressions or not.

Restrictions

- You may not use recursive function(s) to solve this.

Sample Run #1

```
>>> extract_parentheses_I('(1+2)*(3+(4-5))')
()(())
```

Sample Run #2

```
>>> extract_parentheses_I('(1+2)*(3+(4-5)))')
()(()))
```

Sample Run #3

```
>>> extract_parentheses_I('((1+2)*(3+(4-5))')
(()(())
```

1.2 Recursive Parentheses [10 marks]

Question

Write the *recursive* function `extract_parentheses_R(stmt)` to extract all the parentheses from statement `stmt` (`str`) and returns a new string (`str`) of parentheses. You do not have to care if the statement `stmt` is a correct arithmetic expressions or not.

Restrictions

- You may not use iterative constructs (*e.g.*, loop, list comprehensions, *etc.*) to solve this.
- The function `extract_parentheses_R` must be *recursive* (*i.e.*, it calls itself). The use of any recursive helper functions will not be counted as being recursive.

Sample Run #1

```
>>> extract_parentheses_R('(1+2)*(3+(4-5))')
()(())
```

Sample Run #2

```
>>> extract_parentheses_R('(1+2)*(3+(4-5)))')
()(()))
```

Sample Run #3

```
>>> extract_parentheses_R('((1+2)*(3+(4-5))')
(()(())
```

1.3 Check Balanced [15 marks]

Question

Write the function `check_balanced(stmt)` to check if the parentheses are balanced based on normal mathematical conventions.

Sample Run #1

```
>>> check_balanced('(1+2)*(3+(4-5))')
True
```

Sample Run #2

```
>>> check_balanced('(1+2)*(3+(4-5)))')
False
```

Sample Run #3

```
>>> check_balanced('((1+2)*(3+(4-5))')
False
```

Question 2: Premium Burger

In Tutorial 3, Tutorial 4 and Assignment 4, we introduce our burger which we will extend in this question. You are now working for a new fast food chain called the Ten Ten Premium Burger Café! We are selling some *extremely* good selections h ere. In order to provide the best dining experience, here are some *new* rules of our burgers.

- The bun `"B"` is free (*i.e.*, $0).
- All the prices for each individual ingredient are in integer (`int`) dollar (*i.e.*, $).
- We will give you the ingredients in a dictionary excluding the bun.
  - If `"C"` stands for cheese, `"V"` stands for vegetables, `"P"` stands for patty and `"A"` stands for abalone slices (*what!?*) then the prices will be given as dictionary: `{"C": 1, "V": 3, "P": 11, "A": 31}`
  - The ingredients may be arranged in any order. You may not assumed that the dictionary given is sorted.
  - For every price $$p$ –*except for the most expensive price*– the next more expensive price is at least double of $$p$. For instance, in the example price list above, the price for `"V"` is $3 and the next more expensive price is `"P"` at $11. Note that $11 \geq 2 \times 3$.
- Each ingredient can only appear at most in each burger except the bun. In other words, you cannot have a burger `"BPCCB"` because `"C"` appears more than once.
- The burger will only be sandwiched by exactly two pieces of buns: one at the start and one at the end. In other words, you cannot have a burger `"BPBCB"` because there is an additional `"B"` in the middle.

2.1 Most Expensive Burger [30 marks]

One day, your customer comes with exactly $$m$ of money. The customer asks you to find him the most expensive burger you can buy from our Café. As a programmer, you code a quick program to answer this, returning a tuple (`tuple`) `(burger, change)` which is a pair of burger (`str`) satisfying our *representation* of burger and the amount of money left after paying for the burger. Note that if the customer cannot afford any ingredient, then they don't even get the bun.

For maximum taste, the most expensive ingredient should be placed on the topmost part of the burger inside the buns. In our representation, this means that the most expensive ingredient is on the leftmost part of the burger in between the two `"B"`. For instance, given the price list from before, we can have `"BAPVCB"` but not `"BCVPAB"`.

Question

Write the function `most_expensive_burger(money, price_dict)` that takes in the money `money` (`int`) and the price list `price_dict` (`dict`) find the pair of burger (`str`) and the amount of money left (`int`) after paying the burger satisfying the criteria above.

Restrictions

- You are not allowed to modify the input dictionary.

Assumptions

- `money >= 0`

Hint

You can be *greedy* in your construction of the burger. Try to include the most expensive ingredient first into your output burger.

Sample Run #1

```
>>> most_expensive_burger(0, {"C": 1, "V": 3, "P": 11, "A": 31})
('', 0)
```

Sample Run #2

```
>>> most_expensive_burger(25, {"C": 1, "V": 3, "P": 11, "A": 31})
('BPVCB', 10)
```

Sample Run #3

```
>>> most_expensive_burger(55, {'C':1,'W':2,'I':4,'T':9,'O':20,'V':41,'S':85})
('BVTICB', 0)
```

Question 3: Game of Life

In this part, you are given the code that we used before:

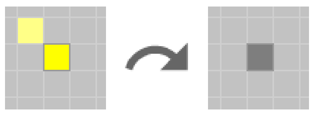
- `create_zero_matrix(n, m)` : Create a 2D array with `n` rows and `m` columns that contains only zeroes.
- `m_tight_print(m)` : Tight printing the 2D array `m`.
- `m_tight_print_gof(m)` : Similar to `m_tight_print(m)` but it prints `" "` and `"#"` instead for better visual effects.

The game of life is not your typical computer game. It is a *cellular automaton* and was invented by Cambridge mathematician John Conway. It consists of a collection of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game.

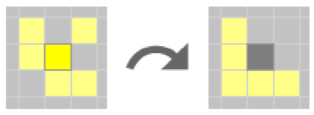
Rules

The following rules apply for a cell that is populated (*i.e.*, live) in the previous step:

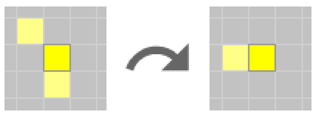
1. Each cell with one or fewer neighbours dies, as if by starvation:



2. Each cell with four or more neighbours dies, as if by overpopulation:

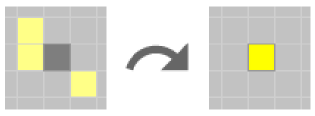


3. Each cell with two or three neighbours survives:



The following rules apply for a cell that is empty or unpopulated (*i.e.*, dead) in the previous step:

1. Each cell with three neighbours becomes populated, as if by migration:



The Game

The game will start with an initial `n` rows and `m` columns board (*i.e.*, 2D array). We denote empty cells with 0 and populated cells with 1. The board is a 2D array created by `create_zero_matrix(n, m)`. You can add initial occupied cells using the function `add_cells(board, cell_list)`. The 2D board is then modified according to the rule above using the function `step(board)`.

For instance, a possible initial state called very long house can be created using the following `cell_list`:
`[(5,6),(5,7),(5,8),(5,9),(5,10),(6,5),(6,8),(6,11),(7,5),(7,6),(7,10),(7,11)]`

continue on the next page...

```
>>> very_long_house = [(5,6),(5,7),(5,8),(5,9),(5,10),(6,5),(6,8),(6,11),(7,5),(7
    ,6),(7,10),(7,11)]
>>> board = create_zero_matrix(15, 15)
>>> add_cells(board, very_long_house)
>>> m_tight_print(board)
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000111110000
000001001001000
000001100011000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
>>> m_tight_print_gof(board)





      #####
     #  #  #
     ##   ##







```

Starting the game and repeatedly calling `step(board)` will give us the sequence shown in Figure 1. In this case, the board is *stabilised* (*i.e.*, unchanged) after some steps. Sometimes, the board will not be stabilised.

Details

We will clarify more about what does it mean by each cell of the next step is fully determined by the neighbours of the previous step. Here is a very simple example of a 2×3 grid. Let the initial board at step 0 be:

```
1100
1111
0100
```

1. The cell at `(0,1)` has 4 neighbours and will die as if by overpopulation.
2. The cell at `(2,0)` has 3 neighbours and will be populated as if by migration.
3. The cell at `(1,3)` has 1 neighbours and will die as if by starvation.
4. The cell at `(2,1)` has 3 neighbours and will survive.

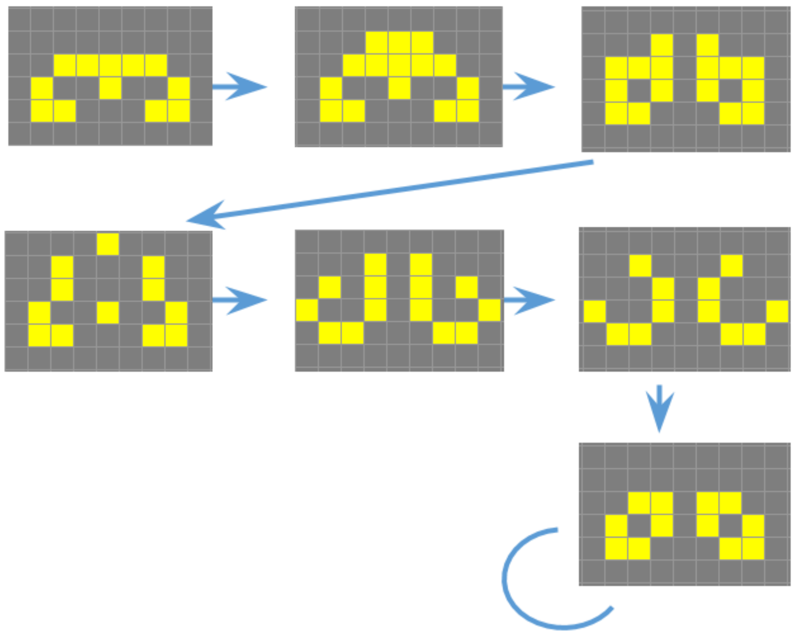


Figure 1: Very long house after 6 steps.

The resulting board will be:

```
1000
0000
1100
```

3.1 Add Cells [5 marks]

Question

Write the function `add_cells(board, cell_list)` that takes in an a `board` and an input sequence `cell_list` of pair of integers (`tuple` of `int`) and *modify* the `board`. The function does not return anything. If there are any locations that are out of the boundary of the board, ignore the location. It should not cause any error or exceptions.

Restrictions

- You are not allowed to return any value and should only modify the board.

Assumptions

- `board` is created from `create_zero_matrix(n, m)` with `n > 0` and `m > 0`

continue on the next page...

Sample Run #1

```
>>> board = create_zero_matrix(2, 3)
>>> add_cells(board, [(1,2), (1,1), (1,0)])
>>> m_tight_print(board)
000
111
>>> add_cells(board, [(0,1), (1,1)])
>>> m_tight_print(board)
010
111
```

3.2 Simulate [30 marks]

Question

Write the function `step(board)` that takes in an input `board` and return a new board representing the next step. You should not modify the input `board`.

Restrictions

- You are not allowed to modify the input `board`.

Assumptions

- `board` is created from `create_zero_matrix(n, m)` with `n > 0` and `m > 0`
- `board` may or may not have been populated using `create_zero_matrix(board, cell_list )`

Sample Run #1

```
>>> board0 = [[1,1,0,0],
              [1,1,1,1],
              [0,1,0,0]]
>>> m_tight_print(board0)
1100
1111
0100
>>> board1 = step(board0)
>>> m_tight_print(board1)
1000
0000
1100
>>> board2 = step(board1)
>>> m_tight_print(board2)
0000
1100
0000
```

– End of Paper –