CS1010S Programming Methodology
National University of Singapore

Practical Examination

16 Apr 2022

Time allowed: 1 hour 30 minutes

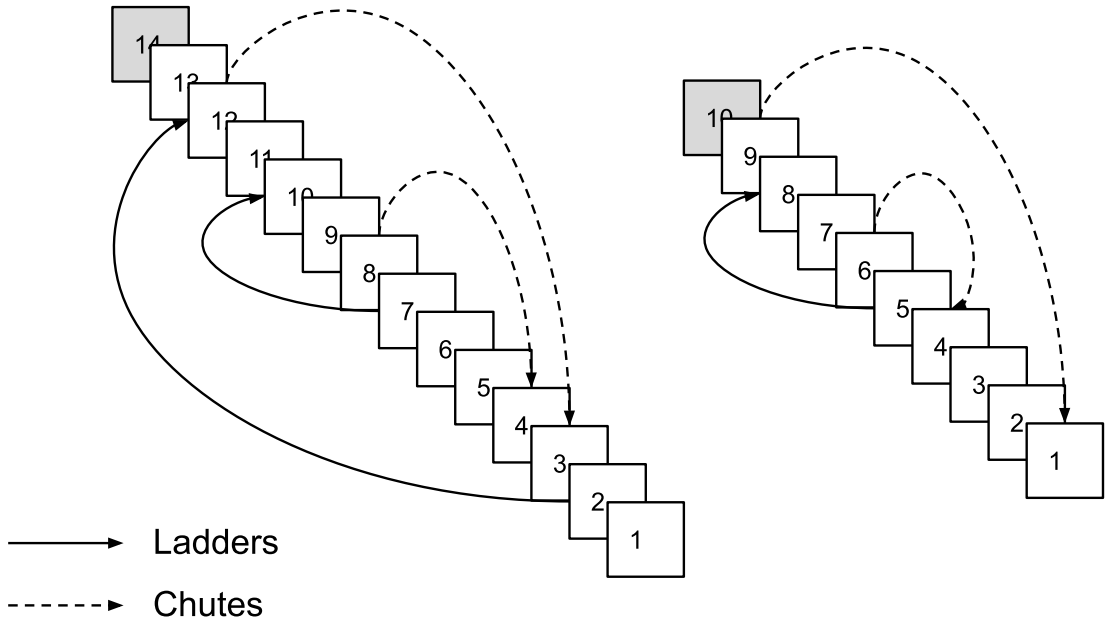
Instructions (please read carefully):

1. This is **an open-book exam**. You are allowed to bring in any course or reference materials in printed form. No electronic media or storage devices are allowed.
2. This practical exam consists of **three** questions (one question with two sub-questions). The time allowed for solving this test is **1 hour 30 minutes**.
3. The maximum score of this test is **20 marks**. Note that the number of marks awarded for each question **IS NOT** correlated with the difficulty of the question.
4. You are advised to attempt all questions. Even if you cannot solve a question correctly, you are likely to get some partial credit for a credible attempt.
5. While you are also provided with the template `practical-template.py` to work with, your answers should be submitted on Coursemology. Note that you can **only run the test cases on Coursemology for a limited number of tries** because they are only for checking that your code is submitted correctly. You are expected to test your own code for correctness using IDLE and not depend only on the provided test cases. Do ensure that you submit your answers correctly by running the test cases at least once.
6. In case there are problems with Coursemology.org and we are not able to upload the answers to Coursemology.org, you will be required to name your file `<mat no>.py` where `<mat no>` is your matriculation number and leave the file on the Desktop. If your file is not named correctly, we will choose any Python file found at random.
7. Please note that it shall be your responsibility to ensure that your solution is submitted correctly to Coursemology and correctly left in the desktop folder at the end of the examination. Failure to do so will render you liable to receiving a grade of **ZERO** for the Practical Exam, or the parts that are not uploaded correctly.
8. Please note that while sample executions are given, it is **not sufficient to write programs that simply satisfy the given examples**. Your programs will be tested on other inputs and they should exhibit the required behaviours as specified by the problems to get full credit. There is no need for you to submit test cases.
9. While you may use any built-in function provided by Python, you may not import functions from any packages, unless otherwise stated and allowed by the question.

GOOD LUCK!

Question 1 : 1-D Chutes-and-Ladders! [5 marks]

Most of you would have played the game of chutes and ladders (also known as *slides and ladders*, *snakes and ladders*). In this game, a player rolls a 6-sided die. The die outcome determines how many squares the player moves. If a player lands at the bottom of a ladder, the player moves up the ladder. If a player lands on top of a chute, the player goes down the chute. The goal is to reach the finish square. In this question, we will deal with one-dimensional version of the same game that contains a sequence of N squares.



Your task is to implement `count_plays` that counts the number of possible ways starting from a given square to the finish square *without ever landing on top of any chute*. The function takes as input a starting position (**a number from** 1 **to** $N-1$) and a representation of a chutes-and-ladders game board. A player need not exactly land on the final square to finish a play. For example, if player is 2 spots away for the the final square and rolls a 6, it is counted as a play.

The board is represented by a tuple `(num_squares, ladders, chutes)` where:

- `num_squares` is the number of squares in the board (N),
- `ladders` is a *dictionary* whose keys are the bottoms of the ladders and whose values are the tops of the respective ladders,
- `chutes` is a *dictionary* whose keys are the tops of the chutes and whose values are the bottoms of the respective chutes.

Sample execution for the boards in the above examples is given as follows:

```
board1 = (14, {2: 12, 7: 12}, {13: 3, 9: 4})
board2 = (10, {5: 8}, {9: 1, 7: 4})

num_plays(13, board1)  # returns 0 (Starts on the chute)
num_plays(12, board1)  # returns 5 (Can be won by 2, 3, 4, 5, 6)
num_plays(5, board2)   # returns 5 (Take the ladder)
num_plays(3, board2)   # returns 37
```

Question 2 : Housing in Wonderland [10 marks]

You are planning to buy your first house in Wonderland. Since you are new there, you first need to understand the housing market to make an informed decision. The mayor of Wonderland has provided you with the previous housing sales data in a csv file.

Important note: You are provided with a data file `house_data.csv` for this question for testing, but your code should work correctly for *any* data file with the same column format and *will* be tested on other data files. There is no inherent ordering among different rows in the data files.

Each row in the data file represents a housing sale transaction, with the first row being the header. The data file contains the following columns in the following fixed format:

price	bedrooms	bathrooms	sqft_living	floor	year_built	zipcode

A. After a some exploration, you observe that this wonderland is pretty expensive. There are 100+ years old houses, probably antiques, that are sold at millions of dollars. You want to get an idea about such antiques in various neighbourhood (identified by `zipcode`). Write a function `get_antiques` that takes the name of the data file and neighbourhood zipcode as inputs. It returns a tuple that contains *built_year and price* of the oldest and the most expensive house sold in the specified neighbourhood. It returns an empty tuple if the neighbourhood does not exists in the data file. [5 marks]

Sample Execution:

```
>>> get_antiques("house_data.csv", 98001)
(1903, 230000.0)
>>> get_antiques("house_data.csv", 98010)
(1900, 214000.0)
```

B. Woah... The wonderland seems very expensive indeed! But is the price a good indicator? Should we not compute price per square foot of the living area?

Write a function `topk_expensive_neighbourhoods` that takes the name of data file and an integer k as inputs. The function returns a list of top k neighbourhoods with highest average price per square foot rounded to two decimal places. The result is represented as a list of tuples wherein every tuple contains a zipcode and the average price per square foot in the respective neighbourhood. You can use in-built `round(num, places)` to perform rounding. [5 marks]

Sample execution:

```
>>> topk_expensive_neighbourhoods("house_data.csv", 1)
[(98039, 568.24)]
>>> topk_expensive_neighbourhoods("house_data.csv", 2)
[(98039, 568.24), (98004, 475.61)]
```

Question 3 : Enn Eff Tees [5 marks]

A non-fungible token (NFT) is a non-interchangeable unit of data stored on the blockchain, which could exist in the form of an image, audio, or videos and can be sold or traded. In recent years, there is considerable increase in interest in NFTs, after a number of high-profile sales and art auctions in the range of millions, and celebrities making these purchases.

Typically, a new collection of NFTs is managed by an NFT contract set up by the collection's creator. The contract specifies the cost price of the NFT, and the maximum number of NFTs that can be minted from the collection. Other users (identified by their blockchain wallets) can then *mint* the NFT by paying the cost price, which will then make them the first owner of the NFT. They are then able to freely sell the NFT to other users.

This question deals with three classes: `Token`,which represents a single NFT; `Contract`, which represents the NFT contract that manages a collection of NFTs; and `Wallet`, which represents a user's blockchain wallet.

`Token` is initialized by its ID number (integer) and its metadata, which is a string containing crucial information about the NFT, such as its image data, etc. It does not contain any methods.

`Wallet` is initialized by the wallet address, amount of Cryptocurrency and the tokens held by the user. It supports following method.

- `sell` takes in the buyer's wallet, the sale price in Cryptocurrency and a token ID. It returns a string based on the following conditions:
  - `'NFT #<token id> does not exist in wallet <wallet address>.'` if the Token with the specified ID does not exist in the wallet.
  - `'Buyer does not have enough funds to make the purchase.'` if the buyer does not have enough Cryptocurrency.
  - `'NFT #<token.id> transferred from wallet <seller address> to wallet <buyer address>.'` if the token with the specified ID exists in the wallet, and the buyer have enough Cryptocurrency. It also transfers the token from seller's wallet to the buyer's wallet.

`Contract` is initialized with a name, NFT value, and a list of token IDs and metadata, stored in the form of tuples `(id, metadata)`. Each time the user mints a new NFT, a new `Token` object is created with the next unsold token ID and metadata. The Contract also stores the contract owner's wallet address, and the cost price of the NFT. It supports following method.

- `mint` takes a `Wallet` and the number of NFTs to mint. It returns a string based on the following conditions:
  - `'Buyer does not have enough funds to mint.'` if the buyer does not have enough Cryptocurrency.
  - `'You need to mint at least one NFT.'` if the mint amount is less than one.
  - `'NFT #<token id> minted to <wallet address>, '` for each NFT minted if there are enough NFTs and the buyer has required Cryptocurrency. It also transfers minted tokens to the buyer's wallet.

Partial implementation of the `Token` and `Wallet` has been given to you in the template

file. Please complete those implementations and provide an implementation for the class `Contract`. You may add other additional methods and properties to the classes as needed. Remember not to break abstractions while implementing your functions.

Sample Execution:

```
>>> kelvin_wallet = Wallet('0x1234abcde', 0.03)
>>> ashish_wallet = Wallet('0x36912CDEF', 0.09)
>>> nft_data = [(1, '001.jpg'), (2, '002.jpg'), (3, '003.jpg')]
>>> nft_contract = Contract('0x2468defAB', 0.03, nft_data)

>>> nft_contract.mint(kelvin_wallet, 0)
'You need to mint at least one NFT.'
>>> nft_contract.mint(kelvin_wallet, 2)
'Buyer does not have enough funds to mint.'
>>> nft_contract.mint(kelvin_wallet, 1)
'NFT #1 minted to wallet 0x1234abcde, '

>>> kelvin_wallet.sell(ashish_wallet, 0.1, 1)
'Buyer does not have enough funds to make the purchase.'
>>> kelvin_wallet.sell(ashish_wallet, 0.03, 2)
'NFT #2 does not exist in wallet 0x1234abcde.'
>>> kelvin_wallet.sell(ashish_wallet, 0.09, 1)
'NFT #1 transferred from wallet 0x1234abcde to wallet 0x36912CDEF.'

>>> nft_contract.mint(kelvin_wallet, 3)
'Max supply of NFTs exceeded.'
>>> nft_contract.mint(kelvin_wallet, 2)
'NFT #2 minted to wallet 0x1234abcde, NFT #3 minted to wallet
   0x1234abcde, '
```

NOTE. Pay close attention to the output strings. Even a single space or punctuation mark may lead to failure for a test case.

You are advised to solve this problem incrementally. Even if you cannot fulfil all the desired behaviours, you can still get partial credit for fulfilling the basic behaviours.

— E N D O F P A P E R —