Faculty of Science
COMP-202 - Foundations of Programming (Winter 2020)
Final Examination

Post date: April 23rd, 2020 at 18:30 — Examiner: Giulia Alberini
Submission period ends: April 26th, 2020 at 18:30 — Associate Examiner: Elizabeth Patitsas

Instructions:

- This exam contains 9 long-answer questions. Under normal circumstances (writing on paper), it is intended to take **3 hours to complete**. You will have 72 hours to submit the exam (see date above). **Late submissions will not be accepted**.
- This is an open book examination. You are allowed to consult your course notes and slides. You can use Thonny (or any other Python IDE) to write and test your code.
- Create a file named `COMP202_final.py`, answer **all** questions within this file, and upload it to codePost (https://codepost.io/). **Please write your code following the order of the questions as presented on this pdf**, i.e. the code for Q1 should be at the top and the code for Q9 should be at the bottom.
- In your codePost submission, you **must** include the file `COMP202W2020_FinalExaminationForm.pdf` signed by you. **If the form is not signed or missing, your exam will not be graded.**
- You can re-upload your submission to codePost as many times as you like. Note that only the latest submission will be graded and points will be given for partially-working programs.
- **Solutions that use functions, methods, or syntactic constructs (such as list comprehensions) that we have not seen in class will be significantly penalized.** Third-party libraries are also prohibited unless explicitly specified in the question.
- Unless stated otherwise:
  - You are allowed to use any build-in function or method we have learned in class during the semester.
  - A docstring for a function should include: the type contract, a description of the function, at least 3 examples demonstrating how the function works (make sure to add at least 1 edge case per function). **You cannot use the examples provided on this pdf.**
  - There's no need for you to implement any input validation. If the instructions state that a function receives a certain kind of input, you can simply make that assumption in your code.
- Marks will be given for the style of your code. You are welcome to add any additional helper function you might need to reduce repetitions and better organize your code. Comments are not required to receive full marks.
- If you think there is an error or ambiguity in an exam question, please post your question **privately** (to all instructors) on Piazza. Do **NOT** email us questions and do **NOT** post your questions publicly on Piazza. We will try to review questions periodically during the exam period, but a response cannot be guaranteed. It is important, therefore, that you express any concerns over errors or ambiguities on your exam script and still answer questions to the best of your ability.
- McGill University values academic integrity. Therefore all students must understand the meaning and consequences of cheating, plagiarism, and other academic offenses under the Code of Student Conduct and Disciplinary Procedures (see www.mcgill.ca/integrity/ for more information). We will be using automated software similarity detection tools to compare your submissions to that of all other students registered in the course. Data generated by this program can be used as admissible evidence, either to initiate or corroborate an investigation or a charge of cheating under Section 16 of the Code of Student Conduct and Disciplinary Procedures.
- Good luck!

Questions

The exam contains 9 questions. Question 1 to 8 are worth 10 points, question 9 is worth 20 points.

1. (10 points) Write a function (including docstring) called `brick_tower` which takes as input three non-negative integers. The first input denotes the number of large bricks you have available. The second input denotes the number of small bricks you have available. The third one indicates the height (in inches) of the tower you have to build. Large bricks have a height of 7 inches, while small bricks have a height of 2 inches. Your function returns true if you can build a brick tower of the specified height given the available bricks, false otherwise. **To get full marks on this function you should not use any loops.** Correct solutions using loops will get partial marks. Note that if the third input is 0, the function should return true.

   For example:

   ```
   >>> brick_tower(1, 3, 13)
   True
   >>> brick_tower(1, 3, 14)
   False
   >>> brick_tower(2, 3, 14)
   True
   >>> brick_tower(2, 3, 15)
   False
   ```

2. (10 points) Write a function (including docstring) called `get_triangle` that takes a non-negative integer `n` and a string `s` as input. The function returns a string representing a triangle where `n` is one of the triangle's heights and `s` is the symbol used to draw it. If the provided integer is a negative number or the provided input string does not contain at least one character then a `ValueError` should be raised. Note that:

   - It does not matter whether or not a space character appears before each new line.
   - If the input provided is 0, the function should return an empty string.

   For example:

   ```
   >>> a = get_triangle(3, '@')
   >>> print(a)
   @
   @ @
   @ @ @
   @ @
   @
   >>> b = get_triangle(2, 'art')
   >>> print(b)
   art
   art art
   art
   ```

3. (10 points) Write a function (including docstring) called `sort_numbers` which takes a one dimensional list of integers as input. The function rearranges the elements so that all the even numbers appear at the beginning of the list, and all the odd numbers appear at the end. The number 0 should be considered an even number. The function modifies directly the input list and it is void. Note that the order of the even elements or the odd elements does not matter. What matters is that even numbers are on the left, and odd numbers are on the right. **To get full marks on this function you should not use any list other than the one the function receives as input (i.e. you should modify the input list in place)**. Correct solutions using additional temporary lists will receive partial marks.

For example, this would be a correct output:

```
>>> a = [1, 2, 4, 5, 7, -8]
>>> sort_numbers(a)
>>> a
[2, 4, -8, 5, 7, 1]
```

Also the following one would be a correct output:

```
>>> a = [1, 2, 4, 5, 7, -8]
>>> sort_numbers(a)
>>> a
[-8, 4, 2, 5, 7, 1]
```

4. (10 points) Write a function (including docstring) called `select_vector` that takes as input a 2-D list of integers, `x`, and a 1-D list of integer, `y`, representing a list of indices. The function returns a 1D list of integers containing the elements of the sublists of `x` which appear in the position specified by `y`. Raise a `ValueError` if either `y` contains too many or too few elements, or if the indices in `y` are out of bounds.

   For example:

   - `select_vector([[8, 4, 5], [2], [9, 5, 3, 1]], [-1, 0, 2])` returns `[5, 2, 3]`
   - `select_vector([[8, 4, 5, 1], [9, 5, 3, 1]], [1, 1, 1])` raises a `ValueError`
   - `select_vector([[8, 4], [5, 1], [9, 5]], [1, 2, 0])` raises a `ValueError`

5. (10 points) Write a function (including docstring) called `get_coordinates` that takes as input a 2-D list of strings `w` and a string `target`. The function returns a list containing tuples representing the indices of where the `target` appears in `w`. This function must be case sensitive. You cannot make any assumption on the inputs beside the fact that the first input is a 2-dimensional list and remember that an empty list is also considered to be a 2-dimensional list.

   For example:

```
>>> w = [['a', 'g'], ['g', 'a', 'k', 'a'], ['a']]
>>> t = get_coordinates(w, 'g')
>>> t
[(0, 1), (1, 0)]

>>> t = get_coordinates(w, 'a')
>>> t
[(0, 0), (1, 1), (1, 3), (2, 0)]

>>> t = get_coordinates(w, 'b')
>>> t
[]

>>> w = [['cats', 'dogs'], ['cat'], ['', 'cats', 'CATS', 'cats'], []]
>>> t = get_coordinates(w, 'cats')
>>> t
[(0, 0), (2, 1), (2, 3)]
```

6. (10 points) Write a function (including docstring) called `evaluate_polynomial` that evaluates polynomials. We can represent polynomials as a collection or power-coefficient pairs. Consider the following polynomial: $p(x) = 2 \cdot x^4 - 3 \cdot x + 5$. We can represent it as the following collection of pairs: (4, 2), (1, -3), (0, 5).

   The function should that take as input a dictionary representing a polynomial, and a value for x. It then returns the value of the polynomial given the value of x.

   For example:

   - `evaluate_polynomial({3: -1, 1: 2}, 3)` returns -21
     (since the value of $-x^3 + 2 \cdot x$ when x is equal to 3 is -21)
   - `evaluate_polynomial({4: 2.5, 1: -3, 0: 5}, 2)` returns 39.0
     (since the value of $2.5 \cdot x^4 - 3 \cdot x + 5$ when x is equal to 2 is 39.0)
   - `evaluate_polynomial({2: -1, 3: 3, 0: 6}, -1.5)` returns -6.375
     (since the value of $3 \cdot x^3 - 1 \cdot x^2 + 6$ when x is equal to -1.5 is -6.375)

   You can assume that the function will receive as input a dictionary mapping non-negative integers to numbers, and as second input a number. No need for you to implement any type of input validation.

7. (10 points) Write a function (including docstring) called `anagrams` which takes as input a list of strings, and a string. *An anagram is a word formed by rearranging the letters of another word.* For example, *cinema* is an anagram of *iceman*. The function returns true if the list contains only anagrams of the specified string. For this exam, we consider a string to be an anagram of another string if they contain exactly the same characters. For instance, `'spear'` is an anagram of `'pears'`, `'trap'` is an anagram of `'trap'`, `'apple'` is **not** an anagram of `'pale'`. Your function should ignore cases. **For this function you are not allowed to use** the built in function `sorted` or the list method `sort`.

   For example:

```
>>> anagrams(['claimed', 'decimal', 'medical'], 'medical')
True
>>> anagrams(['east', 'eatS', 'SEAT'], 'teas')
True
>>> anagrams(['east', 'eats', 'seal'], 'seat')
False
>>> anagrams(['setter', 'street', 'tester'], 'retest')
True
>>> anagrams(['tree', 'test'], 'street')
False
```

8. (10 points) Write a function (including docstring) called `refill_rack`. This function takes as input two dictionaries (one representing the rack of a player, the other representing the pool of letters, respectively) and a positive integer `n`. The function draws letters at random from the pool and adds them to the rack until there are either `n` letters on the rack or no more letters in the pool. The function does not return anything and it might modify both input dictionaries.

   For example,

```
>>> random.seed(5)
>>> r1 = {'a': 2, 'k': 1}
>>> b = {'a': 1, 'e': 2, 'h': 1, 'l': 2, 'n': 1, 'p': 2, 's': 3, 't': 2, 'z': 1}
>>> refill_rack(r1, b, 7)
>>> r1
{'a': 2, 'k': 1, 's': 1, 'l': 1, 't': 1, 'n': 1}
>>> b
{'a': 1, 'e': 2, 'h': 1, 'l': 1, 'p': 2, 's': 2, 't': 1, 'z': 1}
```

```
>>> r2 = {'e': 3, 'q' : 1, 'r' : 1}
>>> refill_rack(r2, b, 8)
>>> r2
{'e': 3, 'q': 1, 'r': 1, 'z': 1, 's': 1, 'a': 1}
>>> b
{'e': 2, 'h': 1, 'l': 1, 'p': 2, 's': 1, 't': 1}

>>> refill_rack(r2, b, 5)
>>> r2
{'e': 3, 'q': 1, 'r': 1, 'z': 1, 's': 1, 'a': 1}
>>> b
{'e': 2, 'h': 1, 'l': 1, 'p': 2, 's': 1, 't': 1}
```

Note that you can use the function `random.choice` to selected a random element out of a list. Note also that if in the pool there are five a's and one b, then the probability of drawing an a should be 5 times larger than the probability of drawing the b.

9. (20 points) Create a class `Cake`, with the following instance attributes:

   - `name` (a string)
   - `ingreds` (a list of strings)
   - `price` (a float)

   The `Cake` class has also the following methods:

   - Write a constructor that takes as input a string indicating the cake name, and a list of strings indicating the cake ingredients. (No docstring needed.) The constructor uses these values to initialize the attributes accordingly. Note that each cake should be assigned a random price between $10.00 (inclusive) and $15.00 (exclusive). You may import `random`.

     Moreover, a cake must contain at least three or more ingredients. If this is not the case, the constructor should raise a `ValueError`. For this constructor, do not worry about making a copy of the input array. You can simply copy the reference received as input into the corresponding attribute.

   - Write a method (including docstring) called `__str__` which returns a string containing the name of the cake and its price (rounded to 2 decimal places using `round`). For instance, the method might return the following string: `'Carrot Cake $13.75'`. Note that this method should **not** modify the values stored in the attributes of the given object.

   - Write a method (including docstring) called `is_better` that takes another cake as input and returns true if the current one has a better (lower) ratio between price and number of ingredients, false otherwise. For example, a $12.00 cake with 6 ingredients (i.e. with ratio 2) is considered to be better than a $10.00 cake with 4 ingredients (i.e. with ratio 2.5). While a $11.00 cake with 4 ingredients (i.e. with ratio 2.75) is considered to be better than a $12.75885 cake with 3 ingredients (i.e. with ratio 4.25295). If the ratio is the same, the method should return false.

   Outside the class `Cake`, write the following two functions (including docstring):

   - A function called `create_menu` that takes as input a dictionary mapping cake names (strings) to their list of ingredients (lists of strings). The function should use the dictionary to create and return a list of `Cake` objects. The function should also display the menu, one cake per line.

   - A function called `find_best` that takes a non-empty list of cakes as input and returns the best cake in the list. If there are ties, the function should return the best cake which appears first in the list.