KIT103/KMA155 Programming Assignment 4: Number Theory 2

Enter your answers to these questions in the accompanying Python script file `programming4.py`. Include your name and student ID in the places indicated in the header comment.

Submit your completed script file to the *Programming Assignment 4: Number Theory 2* assignment folder on MyLO by **1500 (3pm) Wednesday 2 October 2019**.

> Test your solutions thoroughly. Your submission is expected to run without failure (even if it doesn't produce the correct answer for each question). If we have to correct your submission in order for it to run then the maximum total mark you can receive will be 3/5.

Question 1: Decode Linux Permissions (2 marks)

Linux file permissions can be modified by a command (`chmod`) that accepts a three-digit octal number as one of its inputs. *Refer to the Apps 8 lecture slides for details of the structure of this number and how to interpret it.*

Linus Torvalds, the creator of the Linux family of operating systems, is (perhaps undeservedly) renowned for naming his creations after himself. Although the Linux file permissions utility `chmod` is actually from the earlier Unix family of operating systems, it is still fitting that this question be personalised, so you will answer questions about the permissions indicated by octal triplets (000–777) derived from your student ID.

Task 1: Enter your student ID and learn which octal numbers you will decode (0–2 marks)

In the assignment script file is a dictionary of answers for this question and a helper function, `sid2permissions(sid)`, which will generate a pair of octal triplets based on the student ID `sid` it is given.

Your first task is to replace the 0 that is currently assigned to `q1['sid']` with your student ID as an integer with no leading zeroes. Then run the script and inspect the values of `q1['file A']` and `q1['file B']` to learn what octal numbers you will be decoding in the next task.

For example, if your student ID were 134567 then upon running the script you would find that:

- `sid2permissions(q1['sid'])` returns the value `('406', '647')`, so
- `q1['both permissions']` ↦ `('406', '647')`
- `q1['file A']` ↦ `'406'`
- `q1['file B']` ↦ `'647'`

Completing this task contributes no marks in itself, but you will receive zero for all of Question 1 in any of the following situations: you do not modify the `'sid'` entry; you use a student ID that is not your own; or you modify the values that the script generates and assigns to 'both permissions', 'file A' or 'file B'.

Task 2: Answer questions about those permission sets (2 marks)

Now it's time to answer questions about the permissions for two fictitious files, 'file A' and 'file B'. You may use any approach to arrive at answers to the questions below that doesn't involve asking someone else to derive the answer on your behalf. All of them can be answered without the assistance

of a computer (once you know the numbers you are to decode), but if you find it useful to use a function to generate the answer then feel free to do so.

a. The following questions relate to the permissions on **file A**
    1. Rewrite the octal permissions value as a binary literal (a 9-bit integer beginning with 0b)
       In parts 2–4 write `True`, `False` or any valid Boolean expression that will calculate the answer.
       Given these permissions:
    2. Can the user's *group* *r*ead file A?
    3. Can all *other* users *w*rite to file A?
    4. Can all *other* users e*x*ecute file A?
b. The following questions related to the permissions on **file B**
    1. Rewrite the octal permissions value as a binary literal (a 9-bit integer beginning with 0b)
       In parts 2–4 write `True`, `False` or any valid Boolean expression that will calculate the answer.
       Given these permissions:
    2. Can the *user* *w*rite to file B?
    3. Can the user's *group* e*x*ecute B?
    4. Can all *other* users *r*ead file B?

Question 2: base2base (1.5 marks)

In general, to convert a number in base $b_1 \neq 10$ to another base $b_2 \neq 10$ involves two steps: base b_1 to base 10, and then base 10 to base b_2. (Although there exist shortcuts when converting between binary and octal, and binary and hexadecimal, we won't use them here.) **Your task** has two parts:

a. The **first task** is to complete the implementation of the stub function `base2base(n, b1, b2)` according to the following specifications:
    - The parameter `n` is a *string* representing a number in base `b1`. You may assume that `n` will always represent a valid number in base `b1`. The bases `b1` and `b2` are integers in the range 2 through 36 (inclusive)
    - The output of the function is a string representing the equivalent value in base `b2`
    - You may define an additional function to perform one of the steps. That function may be your own creation, or come from the lecture slides or lab solutions
    - For a solution to receive full marks the body of `base2base` must be one line of code (and not merely an alias for another, longer function). Anything longer will receive a maximum of 0.5 marks
b. The **second part of the task** is to record *four* of the test cases you used to check that your function works. Each test case will be recorded as a 4-tuple composed of the arguments (parameter values) for `base2base` (`n`, `b1` and `b2`) and a string representing the expected result. There is a list named `q2_test_cases` declared in the assignment script that you need to modify to contain your four test cases. It already contains one test case, that you must replace, and the others all contain `None`s, which you need to replace with valid values:

    ```
    ('42', 10, 10, '42')
    ```

    which represents calling `base2base('42', 10, 10)` and expecting the result to be '42' (yes, in this 'test' we are expecting it to convert from base 10 to base 10, so the result should be the same as the input).

    You must define four *different* test cases. You may specify any (valid) inputs you wish, as long as `b1` ≠ `b2` in at least three of the cases, you do not repeat any particular combination of `b1` (original base) and `b2` (destination base) values, and none of your test cases uses `b1` = `b2` = 10 (as in the

example). The last entry in each tuple must be the correct result for that particular conversion, even if your implementation of `base2base` doesn't work (i.e., you can get full marks for this part of the question even if you don't attempt part a).

> **Note:** While you *could* use these tests to automatically check your `base2base` implementation, that isn't part of your task. We, however, will check that your test cases are valid.

Question 3: Really Stupid Encryption (RSE) (1.5 marks)

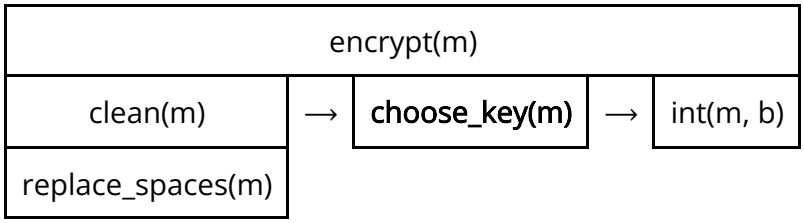
Bonus content in the lecture slides introduced the nearly-unbreakable form of encryption RSA, but in this question you will consider an alternative form of encryption suggested by the Week 10 practical: Really Stupid Encryption (RS*E*).

In this encryption system a plain text message is written using the digits 0–9 and upper case letters A–Z, which is then 'encrypted' by parsing (converting) that string to a decimal integer using a suitable original base (for example, base 36 allows all digits and letters to be used). Decryption is done by converting the integer value back into a string representing a number in the original base.

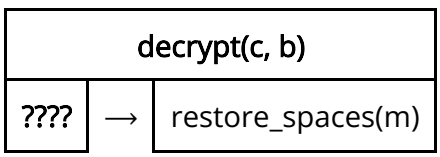
If the key (base) 36 was used in all instances then this system would be too easy to break, so there is a function `choose_key(m)` whose role is to select the *smallest valid key* that can be used to encrypt the message `m`. In other words, it should select a base to use so that every letter in the original message is a valid digit in that base. (The current implementation just selects 36 all the time though.)

The assignment script file already contains a number of functions that can be used to implement this system, and your task will be to provide implementations of the remaining components and, in the last task, write a utility to 'break' the encryption by trying every possible key (original base). These diagrams summarise the various functions and how they work together, and indicates which are already done and which **you need to modify**:

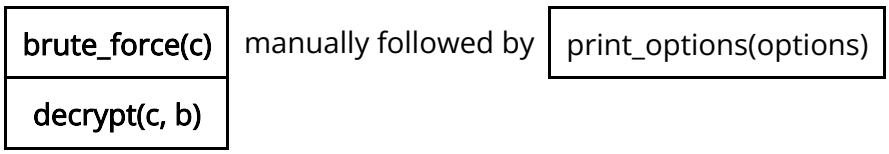
Encryption:



Decryption:



Code breaking:



Your tasks:

a. **Implement `choose_key(m)`** such that it returns the *smallest valid key* that can be used to encrypt the message `m`. **Hint:** Look at the documentation for Python lists to identify a list method that will help in this task.
b. **Implement `decrypt(c, b)`** such that it converts the integer `m` back into a string of base `b` digits and then calls `restore_spaces(m)` on the result. **Hint:** Consider if anything you did in Question 2 will be useful here.
c. **Implement `brute_force(c)`** such that it enumerates the possible bases (2–36) that may have been used to generate the encrypted message `c` and what decrypted message is obtained in each case. The function should return a *list of pairs* in the form (*b*, *m*), where *b* is the candidate key and *m* the result of `decrypt(c, b)`. Its current, default implementation returns a list of just one pair, which is the original encrypted message (which is in base 10).

> You may find it helpful to use `print_options(brute_force(c))` to display the results generated by `brute_force(c)`, but remember that you will *not* use `print_options` within `brute_force`.

Your answers will be assessed such that you will gain marks for any completed component, even if other parts have not been done.

Sample test cases

- For `choose_key()`:
  - `choose_key('123456789')` ⟼ 10
  - `choose_key('HELP ME')` ⟼ 26
  - `choose_key('ATOZ')` ⟼ 36
  - `choose_key('ZTOA')` ⟼ 36
- For `decrypt()`:
  - `decrypt(123456789, 10)` ⟼ `'123456789'`
  - `decrypt(5427943938, 26)` ⟼ `'HELP ME'`
  - `decrypt(505043, 36)` ⟼ `'ATOZ'`
  - `decrypt(606045923, 36)` ⟼ `'A TO Z'`
- Then try your `brute_force()` on the following and look through the decrypted options to see if one looks like a valid English message:
  - 692088749
  - 13020462164572244307999285192829454
  - 24667162469726385787053359406515136 28
  - 15264040648171321964089519467 2
  - 13715763295155569244089153221999309114220164165585836444666440 47
  - 21562081396187580502439703653 4

How your assignment is assessed

Your submitted Python script will be assessed initially by a Python program that will execute your code and check the values generated by your answers against the expected results. This produces a tab-delimited text file named AssessmentReport.txt with columns containing the *expected* result, the *actual* result produced by your code, the *score* achieved for that question (and *maximum* score possible), and any *feedback* explaining the score.

Next, a human assessor will inspect your submission and adjust the computer-based assessment accordingly: code that produced the wrong value but is close to correct will have some marks restored;

code that produced the correct answer but did so incorrectly may have marks reduced. The assessor will add further feedback to the assessment report text file as needed.

You can download the Assessment Report from MyLO: go to Assignments then click feedback link in the Evaluation Status column. View it by opening in Excel or Numbers.

Assessment criteria

Submissions that require modification in order to run without error will receive a maximum mark of 3/5.

- Question 1:
  - Parts a1 and b1: Full marks (0.25 each) to binary literal values that correctly translate the relative 3-digit octal number into binary. Partial marks (0.125 each) may be awarded to values that are not binary literals but which resemble the correct values.
  - Parts a2–4, b2–4: Full marks (0.25) to each correct boolean answer, based on the answers to a1 and b1. Partial marks may be awarded to answers that cannot be based on the information encoded in answers to a1 and b1.
- Question 2a:
  - Full marks (1) will be awarded to solutions operate as specified in the question and in which `base2base` is one line of code that performs some work (and does not *only* call another, longer function that does all the work).
  - Partial marks (up to 0.5) will be awarded to correctly functioning but longer versions of `base2base` or solutions which do not work correctly but which exhibit evidence of progress toward a working implementation.
  - Solutions that do not function correctly and which exhibit little or no evidence of progress toward a valid answer may receive zero.
- Question 2b: Only the first four test cases will be assessed. Do not provide more than that.
  - Full marks (0.5 ≈ 4 x 0.125 each case) will be awarded to valid test cases that exhibit variation in original and destination bases (at most one case can use the same value for `b1` and `b2`) and which don't repeat the same combination of `b1` and `b2`.
  - Individual test cases that duplicate an earlier test case's `b1` and `b2` values will be awarded 0.
  - Individual test cases where the expected result (fourth entry of the tuple) is incorrect will be awarded 0.
- Question 3:
  - Full marks (0.5) to each component (a–c) that has been correctly implemented.
  - Partial marks (0.25) will be awarded to components that, while not working correctly, exhibit evidence of progress toward a valid solution.

Working independently: This is an individual assignment so your submission should be substantially your own work.