Assignment Guidelines:

- This assignment covers material in Module 3.
- Submission details:
  - Solutions to these questions must be placed in files a03q1.py, a03q2.py, a03q3.py, and a03q4.py, respectively, and must be completed using Python 3.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the Python section of the CS116 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page
  - Helper functions need design recipe elements but not examples and tests.
- Download the testing module from the course web page. Include import check in each solution file.
  - When a function returns a floating point value, you must use check.within for your testing. Unless told otherwise, you may use a tolerance of 0.00001 in your tests.
  - Test data for all questions will always meet the stated assumptions for consumed values.
- Restrictions:
  - Do not import any modules other than math and check.
  - Do not use Python constructs from later modules (e.g. loops and lists). You may use any string methods but do not use any other Python functions not discussed in class or explicitly allowed elsewhere. See the allowable functions post on Piazza. You are always allowed to define your own helper functions, as long as they meet the assignment restrictions.
  - While you may use global constants in your solutions, do not use global variables for anything other than testing.
  - Read each question carefully for additional restrictions.
  - The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
1 Phone Number

Write a function `num_only` that consumes nothing and prompts the user for a valid phone number. If a valid number is entered, a string consisting of the ten digits in the valid number is returned. Otherwise, `None` is returned.

Valid phone numbers consist of exactly ten digits. They may have round parentheses around the first three digits and if so, optionally a dash just before the seventh digit. If there are no round parentheses, there can be a dash just before both the fourth and seventh digits. The inclusion of anything else is invalid. For example, '5198884567', '(519)8884567', '(519)888-4567' and '519-888-4567' are the only valid phone numbers resulting in the return string '5198884567'.

The user must be prompted using the string 'Enter a phone number:' as an argument to the `input` function. If the number is invalid, the string 'Invalid number.' must be printed. Otherwise, the string 'Thanks!' must be printed. Make sure you match these strings exactly. Do not print the quotation marks.

Here are four sample interactions:

```
>>> [evaluate a03q1.py]
>>> num_only()  
'Enter a phone number:(519)888-4567
Thanks!
'5198884567'
'5198884567'
'5198884567'
'5198884567'
```

2 Cyclic Shift

Consider the string `abcde`. The cyclic shifts of this string are `bcdea`, `cdeab`, `deabc`, `eabcd`, and `abcde`. In general, the cyclic shifts of a string `s` are all the strings `t` for which there are strings `a` and `b` such that `s==a+b` and `t==b+a`. (Note that `a` and/or `b` could be empty.)

Write a function `contains_cyclic_shift` that consumes strings `s1` and `s2` and returns `True` if `s1` has a cyclic shift of `s2` as a substring. Otherwise, it returns `False`.

For example,

```
contains_cyclic_shift('abccdeababc','abcde') => True and
contains_cyclic_shift('abcdeababc','abcde') => False.
```

3 Bridge

Write a function `bridge` that consumes a positive integer `k`, returns nothing and prints a picture of a bridge of size `k`. 

```
Examples for bridges of size 1, 3, and 8 are shown below.

```
>>> bridge(1)

>>> bridge(3)

>>> bridge(8)
```

You can probably determine the pattern. However, it is difficult to count invisible spaces! So, more precisely, the first line in the picture of a bridge of size $k$ consists of $k$ spaces followed by $k$ `=` characters followed by $k$ spaces. Exactly $k$ more lines follow also with $3k$ characters on each line but two `=` characters and $3k - 2$ spaces. On the last line, the `=` characters appear as the first and last characters. In every line before this, the first `=` character appears one position to the right of the first `=` character in the following line, and the last `=` character appears one position to the left of the last `=` character in the following line.

4 Run Length Encoding

Run length encoding is a simple way to compress strings that contain the same character repeated many times in a row. It uses two key ideas:

- a string can be partitioned into blocks (e.g. `'aaabbabbbb'` can be partitioned into the blocks `'aaa', 'bb', 'a' and 'bbbbb'`), and
- each block can be represented by a number and one character (e.g. `'aaaa'` can be represented by `'5a'`).

For example, the encoding of `'aaabbabbbb'` is `'3a2b1a11b'`.

More formally, a block is a substring of identical characters that is as long as possible. A block is represented by the length of the block followed by the character. The encoding of a string is the representation of each block in the string in the order in which they appear in the string.

The compression ratio of a string $s$ is the length of the encoding of $s$ divided by the length of $s$. For example, the compression ratio of `'aaabbabbbb'` is $\frac{9}{17}$.

Write a function `compression_ratio` that consumes the encoding of a non-empty string where each character is `a` or `b`, and returns the compression ratio of the string as a float. For example, `compression_ratio` (`'3a2b1a11b'`) should return approximately 0.53. Use a tolerance of 0.01 for your tests.

At least 80% of the correctness marks for this problem will include tests where the length of every block in the consumed string is less than ten. For the remaining 20%, you may want to learn about and use the built-in function `lstrip`. 