Question 5: Write a function that takes a string as an argument. Loop through the characters in the string and store the vowels and the nonvowels as separate strings. Then print out those strings. The following is an example execution of such a program. Note that vowels include: a, e, i, o and u. All other characters (numbers, consonants, spaces, punctuation, etc.) are nonvowels.

```python
>>> divide_string_into_vowels_and_nonvowels('This is a sentence')
The vowels are: iiaeee
The nonvowels are: Ths s sntnc
```
Question 7: Write a haggling program. The program is designed to negotiate a price with a user until one of the ending conditions are met. Either the program gives up and there is no sale or the program and user agree on a price. The program should use the global variables below (you do not need to write out these strings):

```python
haggle_statement1 = '''How much money will you pay for this handmade, authentic grunch_o_matic?'''
haggle_statement2 = 'But look at the fine craftsmanship. It is worth at least'
haggle_statement3 = 'I am done negotiating. We are done here. No deal!
'haggle_statement4 = 'It is a deal! You now have your very own grunch_o_matic!
'haggle_statement5 = '''I really cannot afford to sell it for that price. Please try again.''
haggle_statement6 = 'I appreciate the offer, but that is not enough.'
```

Your program should implement the flow chart on the next page. A sample run of the program also appears below.

```python
>>> haggler()
How much money will you pay for this handmade, authentic grunch_o_matic?
How much, in dollars? 50
But look at the fine craftsmanship. It is worth at least 500
I really cannot afford to sell it for only that price. Please try again.
How much, in dollars? 100
I really cannot afford to sell it for only that price. Please try again.
How much, in dollars? 150
I appreciate the offer, but that is not enough.
Is 325.0 OK?
Yes or No? no
How much money will you pay for this handmade, authentic grunch_o_matic?
How much, in dollars? 200
I appreciate the offer, but that is not enough.
Is 262.5 OK?
Yes or No? no
How much money will you pay for this handmade, authentic grunch_o_matic?
How much, in dollars? 225
I appreciate the offer, but that is not enough.
Is 243.75 OK?
Yes or No? yes
It is a deal! You now have your very own grunch_o_matic!
```
**Start**

Print `haggle_statement1` 

Get User Amount from User 

Initialize Old Amount to 0 

Set System Amount to the higher of: 
   a) 500 and 
   b) 2 times User Amount 

Print `haggle_statement2`, followed by System Amount 

Set Haggling to True 

Is Haggling True? 

Yes 

Print `haggle_statement3` 

Set haggling to False 

No 

Old Amount >= User Amount or User Amount equals 0 

Print `haggle_statement3` 

Set haggling to False 

True 

Print `haggle_statement4` 

Set haggling to False 

False 

User amount is equal to System amount 

True 

Print `haggle_statement5` 

Get (new) User Amount from User 

False 

Diff1 = User Amount − Old Amount 

Diff2 = System Amount − User Amount 

Diff2 > (Diff1/4) 

Increase System amount to the average of User Amount and System Ammount 

Print `haggle_statement6` and ask user if (new) System Amount is OK 

True 

False 

Get (new) Old Amount to User Amount 

Get (new) User Amount from user 

False 

User accepts System Amount 

End
Glossary for Python Test

1. Some Basics

- **`return(X)`** causes the current function to exit and cause the expression represented by the function call to evaluate as X. For example given the following steps, the value of `output` would be 5:

```python
def add(num1, num2):
    return(num1+num2)
output = add(2,3)
```

- **`print(X)`** prints X to the screen. This is only for the benefit of the user. It is not useful for having programs interact.

- The parameters of a function are the local variables inside of the parentheses in the function definition. They are useful when you have functions call functions.

- **`input(prompt)`** is used to ask a human being a question so that a program can interact with a human being. This is useful when you want a human being to enter information interactively. `input` statements should be used only when human interaction is appropriate. `input` statements return a string corresponding to what the user typed in. It may be necessary to convert this string to some other data type, e.g., an integer (with `int`) or a float (with `float`).

- The operator `+` will add two numbers or concatenate two strings

- The operator `*` will multiple two numbers or print a string some number of times.

- The operator `**` will represent exponents, e.g., `5**2 == 25`

2. Division and Modulus

- `5 // 2 == 2`
- `5/2 == 2.5`
- `5%2 == 1`

3. More Math

- `round(4.5) == 4, round(4.6) == 5, round(-4.5) == -4, round(-4.6) == -5`
- `round(2/3,2) == .67` ## The second argument of round indicates number of decimal places. The default is to round to the nearest whole number.

- `math.floor(4.9) == 4, math.floor(-4.9) == 5`
- `math.ceil(4.9) == 5, math.ceil(-4.9) == -4`
- `math.trunc(4.9) == 4, math.trunc(-4.9) == -4`
- `math.pi == 3.141592653589793` – a variable for the value of pi
4. **sequences**

- object made up of other objects in an order
- the function `len(sequence)` returns the number of items in the sequence
- the operator `in` tests for membership in sequence, e.g., ('a' in 'abc') would have the value `True`.
- Indices: an index from 0 to (sequence length) represent positions before and after items in a sequence. For example, in the string `duck`, the subscripts indicate indexed positions: $d_0 u_1 c_2 d_3$. In square brackets, index can be used to identify the character that follows that index, e.g., the expression `duck[2]` would evaluate to be `c`, the character after index position 2.
- sequences are used in `for` loops (see below)
- ranges
  - `range(5)` is approximately equivalent to `[0,1,2,3,4]`
  - `range(1,5)` is approximately equivalent to `[1,2,3,4]`
- Strings
  - an empty string has zero characters ""
  - strings are sequences of characters, e.g., 'Hello World!' consists of the items ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']
  - `string.lower()` will return an all-lowercase version of a string, e.g., `'CaT'.lower()` is equivalent to `'cat'`. In the same way, `string.upper()` converts to uppercase, e.g., `'CaT'.upper()` is equivalent to `CAT`.

5. **print**

- `sep` – separator between items
- `end` – printed at the end of print statement

6. **for loops**

- First Line: `for VARIABLE in SEQUENCE:`
  - VARIABLE is set to each item in the sequence one at a time
  - The Indented body repeats once for each item in sequence (for each setting of VARIABLE).
  - It is common to exit a loop of any kind by using a `break`. After a break, the statement after the loop executes.
  - It is common to exit a loop of any kind by using a `return` to exit the function.
  - It is common to initialize a variable outside a loop (called an accumulator) that then gets incremented inside the loop.

7. **while loops**

- First line `While (BOOLEAN-EXPRESSION):`
  - The loop keeps executing the indented body until BOOLEAN-EXPRESSION evaluates as `False`.
  - If BOOLEAN-EXPRESSION is always `True`, the loop is endless.
  - Typically BOOLEAN-EXPRESSION contains one or more variable(s), such that some values of these variables (or this variable) cause BOOLEAN-EXPRESSION to evaluate as `True` and other values cause it to evaluate as `False`.
  - The body of the loop can change these variables. The loop starts when BOOLEAN-EXPRESSION is `True`. Then, under most circumstances, BOOLEAN-EXPRESSION eventually evaluates as `False` and the loop halts. For example, if the BOOLEAN-EXPRESSION is `stop == False`, setting `stop` to `True` inside the loop, will cause the loop to finish.
  - It is common to use accumulator variables in a similar manner as with `for` loops.
  - `break` and `return` behave the same way for `while` loops as they do with `for` loops.
8. if statements

- the first line of an if statement consists of if BOOLEAN-EXPRESSION:
- the body of text indented under the first line is executed if the BOOLEAN-EXPRESSION evaluates to True
- the if statement can be followed by optional elif statements of the same form, except that the first line begins with elif. Each elif statement is only evaluated if the BOOLEAN expressions in the if and elif statements leading up to this one are False.
- The block of if and optional elif statements can end with an optional else statement. The first line is simply else:. The body of text under else executes if the Boolean expressions for all previous if and elif statements in the sequence evaluate to False.

9. logical operators

- X and Y returns True only if both X and Y are True
- X or Y returns True only if X is True, Y is True or both are True
- X in Sequence returns True if X is in a member of a sequence, e.g., 'a' in 'abcdefg' would return True
- X == Y returns True if X and Y are the same
- X != Y returns True if X and Y are the different
- X < Y returns True if X is less than Y
- X > Y returns True if X is greater than Y
- X <= Y returns True if X is less than or equal to Y
- X >= Y returns True if X is greater than or equal to
- Not X returns True if X is False