Pseudocode 101

What is pseudocode? Pseudocode is a simplified, half-English, half-code outline of a computer program.

Why use it?

- Because it can help you to clarify your thoughts, and design a routine properly, before you start to write any code.

  One of the hardest things to resist is the temptation to start writing code! Compared to typing source code, designing the functions which will make up a program seems dull, and perhaps even like a time wasting activity. But spending ten minutes to think out carefully the pros and cons of different approaches to the goal can save you hours of time debugging and refactoring your code later on.

- Pseudocode makes reviews easier. You can review detailed designs without examining source code. Pseudocode makes low-level design reviews easier and reduces the need to review the code itself.

- Pseudocode supports the idea of iterative refinement. You start with a high-level design, refine the design to pseudocode, and then refine the pseudocode to source code. This successive refinement in small steps allows you to check your design as you drive it to lower levels of detail. The result is that you catch highlevel errors at the highest level, mid-level errors at the middle level, and low-level errors at the lowest level – before any of them becomes a problem or contaminates work at more detailed levels.

- Pseudocode makes changes easier. A few lines of pseudocode are easier to change than a page of code. Would you rather change a line on a blueprint or rip out a wall and nail in the two-by-fours somewhere else? The effects aren't as physically dramatic in software, but the principle of changing the product when it's most malleable is the same. One of the keys to the success of a project is to catch errors at the "least-value stage," the stage at which the least effort has been invested. Much less has been invested at the pseudocode stage than after full coding, testing, and debugging, so it makes economic sense to catch the errors early.

- Pseudocode minimizes commenting effort. Often, students write the code and add comments afterward. When we use pseudocode, the pseudocode statements directly become the comments, so it actually takes more work to remove the comments than to leave them in.

- Pseudocode is easier to maintain than other forms of design documentation. With other approaches, design is separated from the code, and when one changes, the two fall out of agreement. With the PPP, the pseudocode statements become comments in the code. As long as the inline comments are maintained, the pseudocode's documentation of the design will be accurate.

Now let's look at some examples of pseudocode in action!

- Example #1 - Computing Sales Tax : Pseudo-code the task of computing the final price of an item after figuring in sales tax.

```
1.          get price of item
2.          get sales tax rate
3.          sales tax = price of item times sales tax rate
4           final price = price of item plus sales tax
5.          display final price
6.          halt
```

Variables: price of item, sales tax rate, sales tax, final price

Note that the operations are numbered and each operation is unambiguous and effectively computable. We also extract and list all variables used in our pseudo-code. This will be useful when translating pseudo-code into a programming language.

Now let's turn this pseudocode into real code. We start by making our pseudocode into Python comments; then, we "fill in the blanks" under each line of pseudocode, filling it with real code (sometimes one line of pseudocode may represent more than one line of real code).

```
def compute_salestax():
      # get price of item
      price = float(raw_input("What is the item's price? ")

      # get sales tax rate
      tax_rate = float(raw_input("Enter the sales tax rate, in decimal: ")

      # sales tax = price of item times sales tax rate
      tax = price * tax_rate

      # final price = price of item plus sales tax
      final_price = price + tax

      # display final price
      print "The final price is:", final_price

      # halt
      return
```

- Example #2 - Computing a Quiz Average: Pseudo-code a routine to calculate your quiz average.

```
Get number of quizzes as a parameter

1.          Initialize "sum" and "count" variables to 0
2.          while count < number of quizzes
             2.1        get quiz grade
             2.2        add quiz grade to "sum"
             2.3        increment count
3.          compute average of sum over number of quizzes
4.          return average
```

Hmm... that looks *pretty* good, but is it the best way? Let's try pseudocoding a different approach, where we instead accept a list of quiz grades as input:

```
Get a list of quiz grades as a parameter


1.          Initialize "sum" variable to 0
2.          Go through each quiz grade in the list
             2.1        add quiz grade to "sum"
3.          compute average of sum over number of quizzes
4.          return average
```

Now we can compare our two pseudocode versions of this function and decide which one to implement. The second version is shorter and doesn't rely on getting user input, which is a better way of coding things, so we choose to implement that version.

```
#  Get a list of quiz grades as a parameter

def compute_quiz_average(quiz_grade_list):
    # Initialize "sum" variable to 0
    sum = 0
    # Go through each quiz grade in the list
    for qgrade in quiz_grade_list:
        # add quiz grade to "sum"
        sum += qgrade
    # compute average of sum over number of quizzes
    num_quizzes = len(quiz_grade_list)
    average = float(sum)/num_quizzes
    # return average
    return average
```