PSEUDOCODE STANDARD adapted from http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html for python

Pseudocode is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudocode needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code.

In general the vocabulary used in the pseudocode should be the vocabulary of the problem domain, not of the implementation domain. The pseudocode is a narrative for someone who knows the requirements (problem domain) and is trying to learn how the solution is organized.

Examples:

> Extract the next word from the line (good)
> set word to get next token (poor)
>
> Append the file extension to the name (good)
> name = name + extension (poor)
>
> FOR all the characters in the name (good)
> FOR character = first to last (ok)

- Note that the logic must be decomposed to the level of a single loop or decision. Thus "Search the list and find the customer with highest balance" is too vague because it takes a loop AND a nested decision to implement it. It's okay to use "Find" or "Lookup" if there's a predefined function for it such as String.indexOf().

*Each textbook and each individual designer may have their own personal style of pseudocode. Pseudocode is not a rigorous notation, since it is read by other people, not by the computer. There is no universal "**standard**" for the industry, but for instructional purposes it is helpful if we all follow a similar style. The format below is recommended for expressing your solutions.*

- The "structured" part of pseudocode is a notation for representing six specific structured programming constructs: SEQUENCE, WHILE, IF, IF-ELSE OR ELIF, FOR. Each of these constructs can be embedded inside any other construct. These constructs represent the logic, or flow of control in an algorithm.
- It has been proven that three basic constructs for flow of control are sufficient to implement any "proper" algorithm.
    - *SEQUENCE* is a linear progression where one task is performed sequentially after another.
    - *WHILE* is a loop (repetition) with a simple conditional test at its beginning.
    - *IF-THEN-ELSE* is a decision (selection) in which a choice is made between two alternative courses of action.

Although these constructs are sufficient, it is often useful to include three more constructs:

- CASE is a multiway branch (decision) based on the value of an expression. CASE is a generalization of IF-THEN-ELSE. **(python does not have a case statement)**
- FOR is a "counting" loop.

SEQUENCE

- Sequential control is indicated by writing one action after another, each action on a line by itself, and all actions aligned with the same indent. The actions are performed in the sequence (top to bottom) that they are written.

Example (non-computer)

- Brush teeth
- Wash face
- Comb hair
- Smile in mirror

Example

- READ height of rectangle
- READ width of rectangle
- COMPUTE area as height times width
- Common Action Keywords

Several keywords are often used to indicate common input, output, and processing operations.

- Input: READ, OBTAIN, GET
- Output: PRINT, DISPLAY, SHOW
- Compute: COMPUTE, CALCULATE, DETERMINE
- Initialize: SET, INIT
- Add one: INCREMENT, BUMP

IF-THEN-ELSE

- Binary choice on a given Boolean condition is indicated by the use of four keywords: IF, THEN, ELSE, and ENDIF. The general form is:

```
IF condition THEN
        sequence 1
ELSE
        sequence 2
```

- **The ELSE keyword and "sequence 2" are optional. If the condition is true, sequence 1 is performed, otherwise sequence 2 is performed.**

Example

```
IF HoursWorked > NormalMax THEN
        Display overtime message
ELSE
        Display regular time message
```

WHILE

- The WHILE construct is used to specify a loop with a test at the top. The general form is:

```
WHILE condition
    Sequence
```

- The loop is entered only if the condition is true. The "sequence" is performed for each iteration. At the conclusion of each iteration, the condition is evaluated and the loop continues as long as the condition is true.

Example

```
WHILE Population < Limit
        Compute Population as Population + Births - Deaths
```

Example

```
WHILE employee.type NOT EQUAL manager AND personCount < numEmployees
    INCREMENT personCount
CALL employeeList.getPerson with personCount RETURNING employee
```

FOR

* This loop is a specialized construct for iterating a specific number of times, often called a "counting" loop. Two keywords, FOR and ENDFOR are used. The general form is:

```
FOR iteration bounds
    sequence
```

* In cases where the loop constraints can be obviously inferred it is best to describe the loop using problem domain vocabulary.

Example

```
FOR each month of the year (good)
FOR month = 1 to 12 (ok)
FOR each employee in the list (good)
FOR empno = 1 to listsize (ok)
```

NESTED CONSTRUCTS
The constructs can be embedded within each other, and this is made clear by use of indenting. Nested constructs should be clearly indented from their surrounding constructs.

Example

```
SET total to zero
WHILE Temperature < zero
    READ Temperature
    IF Temperature > Freezing THEN
        INCREMENT total
Print total
```

* In the above example, the IF construct is nested within the While construct, and therefore is indented.

INVOKING SUBPROCEDURES

Use the CALL keyword. For example:

```
CALL AvgAge with StudentAges
CALL Swap with CurrentItem and TargetItem
CALL Account.debit with CheckAmount
CALL getBalance RETURNING aBalance
```

```
        CALL SquareRoot with orbitHeight RETURNING nominalOrbit
---------------------------------------------------------------------------------
```

Sample Pseudocode

"Adequate"

```
FOR X = 1 to 10
   FOR Y = 1 to 10
      IF gameBoard[X][Y] = 0
         Do nothing
      ELSE
         CALL theCall(X, Y) (recursive method)
         increment counter
```

"Better"

```
Set moveCount to 1
FOR each row on the board
   FOR each column on the board
      IF gameBoard position (row, column) is occupied THEN
         CALL findAdjacentTiles with row, column
         INCREMENT moveCount
```

- (Note: the logic is restructured to omit the "do nothing" clause)

"Not So Good"

```
FOR all the number at the back of the array
   SET Temp equal the addition of each number
   IF > 9 THEN
      get the remainder of the number divided by 10 to that index
      and carry the "1"
   Decrement one
Do it again for numbers before the decimal
```

"Good Enough (not perfect)"

```
SET Carry to 0
FOR each DigitPosition in Number from least significant to most significant
   COMPUTE Total as sum of FirstNum[DigitPosition] and SecondNum[DigitPosition] and Carry
   IF Total > 10 THEN
      SET Carry to 1
      SUBTRACT 10 from Total
   ELSE
      SET Carry to 0

   STORE Total in Result[DigitPosition]

IF Carry = 1 THEN
   RAISE Overflow exception
```
