CS 1: Introduction To Computer Programming, Fall 2013

Assignment 7: The Argument Clinic

Due: *Saturday, December 7, 02:00:00*

Coverage

This assignment covers the material up to lecture 21.

Part What to hand in

You will be handing in two files for this assignment.

You should collect the answers to sections B and C into a file called `lab7_bc.py`. Make sure that it passes the test script for section B: `lab7_b_tests.py`. The answers for section C should be written as comments only.

For section D, you should hand in a file called `Sudoku.py` containing your program. As will be mentioned below, there is a template file that you should use as a starting point when writing your code. Some of the code has already been written for you, notably the `show` method.

We are supplying you with four Sudoku puzzles to test your program on: an easy puzzle, a medium puzzle, a hard puzzle, and a very hard puzzle. Don't copy-and-paste these puzzles into an editor; instead, use the "save page as" feature of your browser (usually located under the File menu) to download the files.

Both files (`lab7_bc.py` and `Sudoku.py`) should be handed in to csman as usual.

Part A: *[OPTIONAL]* Installing new Python packages

There are no new Python packages that you need to install for this week's

assignment.

Part B: Exercises

This section will give you some practice in raising exceptions and working with Python's facilities for handling unusual argument lists in functions.

1. **[15]** Write a function called `mySum` that takes an arbitrary number of arguments, all of which should be integers greater than zero, and returns their sum. If an argument isn't an integer, raise a `TypeError` (along with a meaningful error message). If an argument is `<= 0`, raise a `ValueError` (along with a meaningful error message).

   Examples:

   ```
   >>> mySum(1, 2, 3, 4, 5)
   15
   >>> mySum()
   0
   >>> mySum(2, -1, 3)
   # raises ValueError with message
   >>> mySum([1, 2, 3])
   # raises TypeError with message
   ```

2. **[25]** Write a more general version of `mySum` called `myNewSum` which will also be able to take a single list of positive integers as its only argument and which will return the sum of the list. This function will also be able to take an arbitrary number (zero or more) of individual values (positive integers), but it won't accept both individual values and a list, nor will it accept multiple lists (it will raise a `TypeError` in such cases). This function will also raise a `TypeError` if the list's elements are not integers and will raise a `ValueError` if any of the supplied integers (in a list or outside of a list) are smaller than 1.

   Hint: Handle the list case first. Most of this function is just checking :rgument types and values.

   Examples:

   ```
   >>> myNewSum(1, 2, 3, 4, 5)
   15
   >>> myNewSum(1)
   1
   >>> myNewSum()
   0
   >>> myNewSum(2, -1, 3)
   # raises ValueError with message
   >>> myNewSum(0, 2, 3)
   ```

```
# raises ValueError with message
>>> myNewSum('foo', 'bar')
# raises TypeError with message
>>> myNewSum([1, 2, 3])
6
>>> myNewSum(['foo', 'bar'])
# raises TypeError with message
>>> myNewSum([1, 0, -1])
# raises ValueError with message
>>> myNewSum(1, 2, 3, [4, 5, 6])
# raises TypeError with message
>>> myNewSum([1, 2, 3], [4, 5, 6])
# raises TypeError with message
```

3. **[25]** Write a function called `myOpReduce` which will take one required argument (a list of integers) and one keyword argument called `op`, whose value should be a string. If `op` is `'+'`, then the function will sum all the integers; if it's `'*'`, it will multiply them all together, and if it's `'max'`, it will return the maximum of the numbers. You can ignore `TypeErrors` and `ValueErrors` on the list argument, but if there is no keyword argument, more than one keyword argument, or no valid keyword argument (not one of the three allowable strings), a `ValueError` should be raised (unless there is a keyword argument whose value has a non-string type, in which case a `TypeError` should be raised). All exceptions raised should have a reasonable error message.

   Examples:

```
>>> myOpReduce([1, 2, 3, 4, 5], op='+')
15
>>> myOpReduce([], op='+')
0
>>> myOpReduce([1, 2, 3, 4, 5], op='*')
120
>>> myOpReduce([], op='*')
1
>>> myOpReduce([1, 2, 3, 4, 5], op='max')
5
>>> myOpReduce([], op='max')
0
>>> myOpReduce([1, 2, 3, 4, 5])
# raises ValueError: no keyword argument
>>> myOpReduce([1, 2, 3, 4, 5], op='foo')
# raises ValueError: invalid keyword argument
>>> myOpReduce([1, 2, 3, 4, 5], op=123)
# raises TypeError: value for keyword argument 'op' must be a string
>>> myOpReduce([1, 2, 3, 4, 5], bla='+')
# raises ValueError: invalid keyword argument
>>> myOpReduce([1, 2, 3, 4, 5], op='+', bla='foo')
# raises ValueError: too many keyword arguments
```

Part C: Pitfalls: Exception handling

Exception handling is a tricky subject. Although there are no absolute guidelines on the best way to handle exceptions, there are some ways of doing it that are clearly wrong. In this section we'll present you with some incorrect ways of handling exceptions. In each case you should identify what the problem is (write this in a comment) and then write the code the way it should be written (you will not have to write any extra functions beyond the ones shown here). There may be more than one correct way to rewrite the code; justify what you are doing in a comment. Feel free to type the code in to a Python interpreter and experiment with it to see what happens.

While working through this section, think about which parts of the code should be responsible for notifying the rest of the program of an error, and which parts should be responsible for handling errors that have been raised elsewhere.

1. **[10]**

```
import sys

def sum_of_key_values(dict, key1, key2):
    '''Return the sum of the values in the dictionary stored at key1 and key2.'''
    try:
        return dict[key1] + dict[key2]
    except KeyError:
        quit()
```

2. **[10]**

```
import sys

def sum_of_key_values(dict, key1, key2):
    '''Return the sum of the values in the dictionary stored at key1 and key2.'''
    try:
        return dict[key1] + dict[key2]
    except KeyError:   # raised if a key isn't in a dictionary
        print >> sys.stderr, 'key not found!'
```

3. **[10]**

```
def sum_of_key_values(dict, key1, key2):
    '''Return the sum of the values in the dictionary stored at key1 and key2.'''
    try:
        return dict[key1] + dict[key2]
    except KeyError:   # raised if a key isn't in a dictionary
        raise KeyError
```

4. **[10]**

```
def sum_of_key_values(dict, key1, key2):
```

```
    '''Return the sum of the values in the dictionary stored at key1 and key2.'''
    try:
        val1 = dict[key1]
    except KeyError, e:
        raise e

    try:
        val2 = dict[key2]
    except KeyError, e:
        raise e

    return val1 + val2
```

5. **[10]**

```
import sys

def fib(n):
    '''Return the nth fibonacci number.'''
    if n < 0:
        raise ValueError
        print >> sys.stderr, 'n must be >= 0'
    elif n < 2:
        return n  # base cases: fib(0) = 0, fib(1) = 1.
    else:
        return fib(n-1) + fib(n-2)

# NOTE: This is a very inefficient recursive function.  The
# inefficiency is not the problem we want you to identify.
```

6. **[10]**

```
import sys

def fib(n):
    '''Return the nth fibonacci number.'''
    if n < 0:
        print >> sys.stderr, 'n must be >= 0'
        raise ValueError
    elif n < 2:
        return n  # base cases: fib(0) = 0, fib(1) = 1.
    else:
        return fib(n-1) + fib(n-2)

# NOTE: This is a very inefficient recursive function.  The
# inefficiency is not the problem we want you to identify.
```

7. **[10]**

```
from math import exp

def exp_x_over_x(x):
    '''Return the value of e**x / x, for x > 0 and
    e = 2.71828... (base of natural logarithms).'''
```

```
    if x < 0:
        raise TypeError('x must be >= 0.0')
    return (exp(x) / x)
```

8. **[10]**

```
from math import exp

def exp_x_over_x(x):
    '''Return the value of e**x / x, for x > 0 and
    e = 2.71828... (base of natural logarithms).'''
    if type(x) is not float:
        raise Exception('x must be a float')
    elif x <= 0:
        raise Exception('x must be > 0.0')
    return (exp(x) / x)
```

Part D: Miniproject: An Interactive Sudoku Program

Introduction

For this week's miniproject, you will be writing a program which will help you to solve Sudoku puzzles interactively. If you aren't familiar with Sudoku, this article is a good description of Sudoku puzzles, so you should read that article before continuing, since we will assume you know how to play Sudoku in the following problem description.

Many programmers have written programs to automatically solve Sudoku problems. These are interesting and fun programs to write, but this week's miniproject is a bit different. For this program, we assume that you have some Sudoku problems that you would like to solve yourself, but you don't want to write the solution on a piece of paper. (One problem with solving Sudoku puzzles on paper is that it's hard to undo what you've written if you make a mistake; in contrast, your program will make it easy to undo your moves.) Your program will be an *interactive Sudoku program*; it will present you with a Sudoku puzzle and ask *you* to solve it. It will allow you to enter moves as well as to undo moves if you find that you get stuck. (Therefore, you may find that you will enjoy solving Sudoku problems using your program more than you do with pencil and paper.) However, your program will *not* solve the Sudoku puzzle for you; all the moves have to come from you (much like the Mastermind game you wrote way back in lab 2). Your program will, however, keep track of the current state of the grid of numbers (which we will refer to as the Sudoku "board"), what moves have been made and in what order, and it will not allow you to make an invalid move.

This program will also give you an opportunity to work with Python classes on a more realistic scale. You will see that your objects can store all the information about the state of the puzzle, including the entire history of what moves were played and in what order.

Note: The CS 1 final exam will contain a miniproject of about the same size as this one. You should consider this as a warmup for the final exam!

The program

1. **[180]** Write an interactive Sudoku puzzle game. The main part of the game will be a class called `Sudoku`, which will have the following methods:

   - [`__init__`] The `__init__` method will create the board and store it as a field of the `Sudoku` object. The board will be a list of lists of numbers (a list of 9 lists of length 9 each), representing the 9x9 grid of numbers in the Sudoku board. An empty square is represented by the number 0. In addition, `Sudoku` instances will also have a field representing a list of moves, which is initially empty; this field is initialized in this method.

     Be careful! When creating the board, it's very easy to create a list of lists where the lists are not actually copies of each other but references to the same list, and this may result in some very peculiar bugs due to aliasing. **On the other hand, we don't want you to just manually write out a 9x9 list of lists!** (You won't get any credit for this method if you do that!) If you're having problems with this, ask your TA.

     Note that each instance of the `Sudoku` class will be a fully-functional Sudoku game object, even though you won't need more than one instance.

   - [`load`] The `load` method will take one (non-`self`) argument (a filename) and try to load the contents of that file into the object's board representation. For this to work, the file should consist of 9 lines, each of which contains exactly 9 numbers in the range from 0 to 9 (not counting the newline at the end of each line). Don't catch any exceptions which may be raised in this method (*e.g.* an `IOError` if the file isn't found). If the file exists, but it has the wrong number of lines, or the lines are not exactly 9 characters long (not counting the newline at the end), or the lines have characters other than the digits 0 to 9 (spaces are not acceptable), then you should raise an `IOError` exception with an error message appropriate for the specific error. Again, don't catch this error in this method.

If the file is OK, then all the digits in the file will get copied into the board representation. The board will then be a list of list of integers in the range 0 to 9. This method should also clear the list of moves, since you are starting a new game.

A sample file might look like this:

```
060000010
007060000
000072000
304086900
200000001
005130406
000540000
000010600
090000080
```

Notice that each line consists of 9 digits. The 0s are blanks, so the real puzzle would be:

```
.6.....1.
..7.6....
....72...
3.4.869..
2.......1
..513.4.6
...54....
....1.6..
.9.....8.
```

where the `.` represents a blank space.

- [`save`] The `save` method takes one non-`self` argument, which is the name of a file to save the board representation to. This method will write the board representation to the file with the given name. Once written, the contents of the file should be in exactly the same format as the format that the `load` method can read: 9 lines of 9 digits, with no spaces. You don't have to raise or catch any exceptions in this method. If you do this right, the output file saved by the `save` method could be successfully loaded by the `load` method later on.

- [`show`] The `show` method will take no (non-`self`) arguments, and will print out the board in the exact format shown below in the example. For the board shown above, the `show` method would display it as follows:

```
   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  6  |     |  1  |
2 |    7|  6  |     |
```

```
3 |     |  7 2|     |
  +-----+-----+-----+
4 |3   4|  8 6|9    |
5 |2    |     |    1|
6 |    5|1 3  |4   6|
  +-----+-----+-----+
7 |     |5 4  |     |
8 |     |  1  |6    |
9 |  9  |     |  8  |
  +-----+-----+-----+
```

This method has been supplied for you in the template code, though you are free to change it if you really want to.

- [`move`] The `move` method is the most important method in the program. It takes three non-`self` arguments: the row coordinate, the column coordinate, and the number to place at those coordinates. The row and column coordinates are integers which range from 1 to 9, and the value is also an integer ranging from 1 to 9. The `move` method has these responsibilities:

  1. It checks that the inputs are valid coordinates.
  2. It checks that the move is a valid move on this board.
  3. It makes the move by writing the value into the board representation.
  4. It appends the move into the list of moves stored in the object.

  Let's expand on this a bit.

  If the inputs are not valid row/column coordinates (integers between 1 and 9) then the method should raise a `SudokuMoveError` exception with an error message which describes the exact problem which gave rise to the error. The `SudokuMoveError` exception class is defined for you in the template code, but you have to raise these exceptions in the code you write.

  For the move to be considered valid, the location in which the number is stored must be empty, which means that it must contain the number 0. In addition, there must be no number of that value anywhere in the same row, column, or box. Checking the row and column for the number is fairly easy, but checking the box is a bit tricky. What you should do is compute what the ranges of row and column coordinates are for the box in which the location specified is contained, and then look for the number in that box. If you find the number in the box, it's an error.

Making the move, once you have determined that it's a valid move, is very simple. If the move is not valid, the method should raise a `SudokuMoveError` exception with an error message which describes what the problem is (*e.g.* occupied space, row conflict, column conflict, box conflict).

When you write the move into the move list, you should represent the move as a tuple containing three numbers: the row coordinate, the column coordinate, and the number that was written into that location. So the move "at row 2, column 6, place a 1" would be represented as the tuple `(2, 6, 1)`. Each new move should be appended to the end of the move list.

- [`undo`] The `undo` method is quite simple. It should remove the last entry in the stored list of moves, and it should erase the contents of the board at the coordinates of the last move (which means storing a 0 at that location). One handy method that you may not be familiar with is the `pop` method of lists. It removes the last element of a list and returns it.

- [`solve`] The `solve` method is the method that handles user interaction (playing the game). It runs an infinite loop in which it asks the user for a command using `raw_input`, executes the command (if possible) by calling one of the other methods, asks for another command, and so on. Here are the commands that `solve` understands:

  - `q`: This causes the `solve` method to return. `q` stands for "quit" although typing `q` doesn't actually make the program exit (it just exits the `solve` method).

  - A three-digit sequence where all the digits are in the range 1 to 9. This causes a move to be made by calling the `move` method. The first two digits represent a row and a column coordinate on the board, and the last digit is the number to place at that location.

  - `u`: This undoes the last move.

  - `s <filename>`: This saves the current state of the board to the file named `<filename>`.

  Any other command is an error, and the `solve` method should raise a `SudokuCommandError` exception, giving the bad command as an argument to the exception. The `SudokuCommandError` exception class is also defined for you in the template code.

After every new move or undo command, the board should be printed using a call to the `show` method.

There are two kinds of exceptions that can be raised as a result of running this method: the `SudokuCommandError` exceptions (raised in this method) and the `SudokuMoveError` exceptions (raised in the `move` method). You should put all of the code inside the loop into a `try` block and have two `except` blocks to handle both kinds of exceptions. In each `except` block you should print out the exception and ask the user to try again. This is a nice example of exception handling as you are handling both "local" exceptions (the `SudokuCommandError` exceptions raised in this method) and "remote" exceptions (the `SudokuMoveError` exceptions raised in the `move` method).

You can write other methods as well, if you find that to be helpful.

To help you get started, we are supplying you with a template file called `Sudoku.py` which provides a skeleton for your code. You need to fill in the parts marked `# TODO`, which means that you must supply the bodies of several methods (including docstrings). Please remove the `# TODO` comments before submitting the completed version.

Example interaction

Here is a sample interaction with the Sudoku program, starting from the terminal. The file `easy.su` is a Sudoku puzzle with the following contents:

```
010089000
600200708
070003510
000010054
095030670
120040000
056100080
904008005
000390060
```

As we mentioned above, the `0`s represent unfilled spaces, so this is really:

```
.1..89...
6..2..7.8
.7...351.
....1..54
.95.3.67.
12..4....
.561...8.
9.4..8..5
...39..6.
```

(with . being a blank space.)

Running the program on this file produces this output, with someone (the user of the program) inputting commands at the sudoku> prompt:

```
% python Sudoku.py
Enter the sudoku filename: easy.su

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2    |7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7  |
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|    5|
9 |     |3 9  |  6  |
  +-----+-----+-----+

sudoku> 315
Invalid move: row conflict; please try again.

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2    |7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7  |
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|    5|
9 |     |3 9  |  6  |
  +-----+-----+-----+

sudoku> 319
Invalid move: column conflict; please try again.

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2    |7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7  |
6 |1 2  |  4  |     |
  +-----+-----+-----+
```

```
7 |  5 6|1    |  8  |
8 |9   4|    8|    5|
9 |     |3 9  |  6  |
  +-----+-----+-----+

sudoku> 349
Invalid move: box conflict; please try again.

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2    |7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7  |
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|    5|
9 |     |3 9  |  6  |
  +-----+-----+-----+

sudoku> 261

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2   1|7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7  |
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|    5|
9 |     |3 9  |  6  |
  +-----+-----+-----+

sudoku> 591

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2   1|7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7 1|
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|    5|
9 |     |3 9  |  6  |
```

```
  +-----+-----+-----+

sudoku> 931

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2   1|7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7 1|
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|    5|
9 |    1|3 9  |  6  |
  +-----+-----+-----+

sudoku> 871

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2   1|7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7 1|
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|1   5|
9 |    1|3 9  |  6  |
  +-----+-----+-----+

sudoku> 856

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2   1|7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7 1|
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|  6 8|1   5|
9 |    1|3 9  |  6  |
  +-----+-----+-----+

sudoku> u
Undoing last move...
```

```
   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2   1|7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7 1|
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|1   5|
9 |    1|3 9  |  6  |
  +-----+-----+-----+

sudoku> s easy1.out

   1 2 3 4 5 6 7 8 9
  +-----+-----+-----+
1 |  1  |  8 9|     |
2 |6    |2   1|7   8|
3 |  7  |    3|5 1  |
  +-----+-----+-----+
4 |     |  1  |  5 4|
5 |  9 5|  3  |6 7 1|
6 |1 2  |  4  |     |
  +-----+-----+-----+
7 |  5 6|1    |  8  |
8 |9   4|    8|1   5|
9 |    1|3 9  |  6  |
  +-----+-----+-----+

sudoku> q
```

Of course, we could have continued further, but this gives you the idea of what we want. After this code has executed, the file `easy1.out` will have the following contents:

```
010089000
600201708
070003510
000010054
095030671
120040000
056100080
904008105
001390060
```

Our solution is less than 200 lines long.
