Lab Exercise #4

Assignment Overview

This lab exercise provides practice with functions.

You will work with a partner on this exercise during your lab session. Two people should work at one computer. Occasionally switch the person who is typing. Talk to each other about what you are doing and why so that both of you understand each step.

Mimir testing: There is no Mimir testing for Part A. For B through E create one file named `lab04.py` and keep adding your functions to the same file, but comment out any instructions outside of functions when you run Mimir tests.

PART A: EXECUTION VISULIZATION FOR DEBUGGING

Making the connection between the text of a program and what it does is the key to learning to program. `Pythontutor.com` (http://www.pythontutor.com/visualize.html) makes this much easier as it actually shows you what your program is doing as it runs.
When you first learn to program you have to master the art of seeing what your program does to the contents of the variables you have declared. It gets even more difficult when you are working with references to storage, as you will see later in the semester. You can type in a Python program straight into a browser and run it at once. As you step through it, an illustration shows what is stored in each location and the variables that reference the location.

The following exercise is designed to help you understand, using a visualizer, how functions works. You will explore when variables are created, how long they exist in computer memory, and what happens when two variables have the same name and when there are two different functions.

Note: There is no coding and no Mimir testing for this part.

Copy the content of the program `example.py` into the text editor in `Pythonturor.com` and click on the "*Visualize Execution*" button. When you hit visualize execution, you are brought to an interactive visualization of what goes on in your program step by step. Press '`Next`' to execute the next line of code. You can even step backwards by pressing '`Prev`'.

There are two functions `convert_to_minutes` which converts a number of hours to an equivalent number of minutes and another function `convert_ to_seconds`, which converts a number of hours to an equivalent number of seconds. `Convert_to_seconds` does its work by calling `convert_to_minutes` and then multiplying the result by 60. Do the following:

1- Step quickly over the function definitions: two function objects are created. These objects contains the memory address of a function object. That function object contains all information about the function including any code that needs to be executed, the parameter, and the doc strings.
2- Notice the variable "`seconds`" has not been created yet, that will happen, after we finish executing the assignment statement.

3- Visualize the code step by step and see what happen: The first step is to evaluate the right-hand side which is a function call. Python first evaluates the argument and creates an object for that value.
4- During execution of a function call, a new frame is created. Whenever a function exits, the current frame is erased and execution of the previous function continues.

5- There are two variables called `num_hours`. One of them is in the frame for the call to `convert_to_minutes`, and the other is in the frame for the call to `convert_to_seconds`. Python keeps these two running functions in separate areas of memory so it does not get confused about which variable to use.

6- The first line in the body of `convert_to_minutes` is an assignment state (line 5). On the right hand side, there is an expression `num_hours` x 60. Which `num_hours` do we use?
Hint: python looks in the current frame.

7- The second step of the assignment is to assign the result to variable `minutes`. If minutes does not exist in the current frame, then it will be created. Each frame contains its own set of variables.
8- In line 6, we are now about to return the value of variable `minutes,` and exit the current function. But where do we return to?
Hint: the answer is always, to the next frame on the call stack.

9- When this call is complete, the current frame is erased, and Python produces the return value as the value of this function call expression. This call is on the right-hand side of an assignment statement. To complete the assignment, variable `minutes` will be created, and Python will store the memory address of the return value.

10- Continue visualizing the code and see that every time we exit from a function the current frame is erased and execution of the previous function continues.

PART B: LEAP YEAR

A leap year in the Gregorian calendar system is a year that is divisible by 400 or a year that is divisible by 4 but not by 100. Write a function `named leap_year` that takes one **string** parameter. It returns a **Boolean** `True` if the string represents a leap year, and returns a **Boolean** `False` otherwise.

For example, 1896, 1904, and 2000 are leap years, but 1900 is not. Therefore,

```
leap_year('1896')
```

returns `True`.
(Optional challenge: write the function suite as one line.)

★ **Demonstrate your completed program to your TA. On-line students should submit the completed program (named "lab04.py") for grading via the Mimir system.**

PART C: ROTATE

Write a function `rotate(s,n)` that has one **string** parameter `s` followed by a positive **integer** parameter `n`. It returns a rotated string such that the last `n` characters have been moved to the beginning. If the string is empty or a single character, the function should simply return the string unchanged. Assume that `n` is less than or equal to the length of `s` and that `n` is a positive intger.

For example:
`rotate('abcdefgh',3)` returns `'fghabcde'`
(Optional challenge: write the function to handle `n` larger than the length of `s`.)

★ **Demonstrate your completed program to your TA. On-line students should submit the completed program (named "lab04.py") for grading via the Mimir system.**

PART D: DIGIT COUNT

Write a function named `digit_count` that takes one parameter that is a number (**int** or **float**) and returns a count of even digits, a count of odd digits, and a count of zeros that are to the left of the decimal point. Return the three counts in that order: `even_count, odd_count, zero_count`.

For example:
`digit_count(1234567890123)` returns `(5, 7, 1)`
`digit_count(123400.345)` returns `(2, 2, 2)`
`digit_count(123.)` returns `(1, 2, 0)`
`digit_count(0.123)` returns `(0, 0, 1)`

★ **Demonstrate your completed program to your TA. On-line students should submit the completed program (named "lab04.py") for grading via the Mimir system.**

PART E: FLOAT CHECK

String has a method `s.isdigit()` that returns `True` if string `s` contains only digits and `False` otherwise, i.e. `s` is a string that represents an integer. Write a function named `float_check` that takes one parameter that is a **string** and returns `True` if the string represents a `float` and `False` otherwise. For the purpose of this function we define a `float` to be a string of digits that has at most one decimal point. Note that under this definition an integer argument will return `True`. Remember "edge cases" such as "45." or ".45"; both should return `True`.

For example:
`float_check('1234')` returns `True`
`float_check('123.45')` returns `True`
`float_check('123.45.67')` returns `False`
`float_check('34e46')` returns `False`
`float_check('.45')` returns `True`
`float_check('45.')` returns `True`
`float_check('45..')` returns `False`

(Optional challenge: write this function suite in one line.)

★ **Demonstrate your completed program to your TA. On-line students should submit the completed program (named "lab04.py") for grading via the Mimir system.**