Exercises

Level 3: Intermediate Python Syntax

3.1: Advanced Functions

1) Create a stored lambda function that calculates the hypotenuse of a right triangle; it should take base and height as its parameter. Invoke (test) this lambda with different arguments.

2) This exercise is a modification of Exercise 1.5.8, to use the **reduce** function. In Exercise 1.5.8, we created functions to calculate the WAM and WAR of a list of mortgage tuples.

- **a.** Create an alternate version of WAM, which takes a list of mortgage tuples; for this version, use the **reduce** function, with a *regular function* as its callable, to calculate the WAM.
- **b.** Create an alternate version of WAR, which takes a list of mortgage tuples; for this version, use the **reduce** function, with a *lambda* as its callable, to calculate the WAR of the list of rates.
- **c.** Modify your WAR and WAM functions in your **LoanPool** class to take the above **reduce** approach instead of the previous approach.

3) Create a regular function (called **reconcileLists**) that takes two separate lists as its parameters. In this example, List 1 represents risk valuations per trade (i.e. Delta) from *Risk System A* and List 2 has the same from *Risk System B*. The purpose of this function is to reconcile the two lists and report the differences between the two systems. To this end, it should return a list of True or False values, corresponding to each value in the lists (True means they match at *index*, False means they don't match at *index*).

Test the **reconcileLists** function with different lists of values (lists should be of at least length ten). Note that the assumption is that both lists are the same length (report an error otherwise).

4) To incorporate lambda into the previous exercise, do the following:

- **a.** Create a **breakAbsolute** stored lambda which takes two values and an *epsilon* parameter. This lambda should 'return' True if the two values are not within *epsilon* of each other.
- **b.** Create a **breakRelative** stored lambda which takes two values and a *percent* parameter. This lambda should 'return' True if the percent difference between the two values exceeds *percent*.
- **c.** Create a **breakAbsRelative** function which takes two values and a *percent* parameter. This should return True if the percent difference between the absolute values of the two values exceeds *percent*.
- **d.** Modify the **reconcileLists** function to take a third parameter, called *breakFn* (this represents a passed-in function or lambda). The **reconcileLists** function should utilize the passed-in *breakFn* function to build the True/False list. You will need to use **functools.partial** to specify the parameter of the *breakFn* function (i.e., epsilon or percent).
- **e.** Test **reconcileLists** with different lists of values (should be large lists of numbers) and with each of the above **break*** functions.

5) The previous exercise presents a good use-case for **functools.partial**:

   a. Create a *partial* called **reconcileListsBreakAbsolute** (which uses the **breakAbsolute** function). Test this comprehensively.
   b. Create similar *partial* functions for each of the **break*** functions in the previous exercise.

3.2: Generators 101

1) Create a list of 1000 numbers. Convert the list to an iterable and iterate through it.

2) Create a list of 1000 numbers. Convert the list to a reversed iterable and iterate through it.

3) Modify your **LoanPool** class to be an iterable. To do this, you will need to define an **__iter__** method within the class; this method should be a generator, that returns one **Loan** at a time. Effectively, the result will be that you should be able to loop over a **LoanPool** object's individual **Loan** objects. For example, the following should now work:

```
for loan in loanPool:
    print(loan.notional)
```

4) Modify the Fibonacci function from Exercise 1.3.2 to be a generator function. Note that the function should no longer have any input parameter since making it a generator allows it to return the infinite sequence. Do the following:
   a. Display the first and second values of the Fibonacci sequence.
   b. Iterate through and display the next 100 values of the sequence.

5) Generator expressions:
   a. Create a list comprehension that contains the square of all numbers from 0-5,000,000, using **range**. Sum this using the built-in *sum* function.
   b. Compare the total time taken to build and sum each. Which one is faster? What are the benefits of using the generator instead of the list comprehension? Why?

6) Create three generator expressions and use **itertools.chain** to attach them together. Print out the result as a list.

7) Create three generator expressions and zip them together. Print out the result as a list.

8) Create three generator expressions and use the appropriate **itertools** function to get all the combinations of the values. Print out the result as a list.

9) Create a list of ten names. Loop through the list and output each name in the following format:

Name 1: Henry
Name 2: Jake

3.3: Exception Handling

1) Create code that takes a numerator and denominator input from the user. Output the quotient in decimal form. Handle the divide-by-zero case gracefully, using exception handling.

2) Extend exercise 1) to handle the situation when the user inputs something other than a number, using exception handling. If the user does not enter a number, the code should provide the user with an error message and ask the user to try again.

Note that this is an example of *duck typing*.

3) Create a function that calculates the factorial of an input number. If the input value is invalid, raise an exception. Test this out in main(), and handle the exception. Provide several examples, using explicit error handling and general error handling (catching all error types).

4) Modify all the applicable **Loan** classes from Level 2 so that if an incorrect **Asset** type is passed-into the **__init__** function, an exception is raised (instead of printing the message to the user). Test this out in **main**, and handle the exception.

Note that all future exercises should utilize exception raising/handling instead of printing error messages to the user.

3.4: Context Managers

1) Open a file and write to it, using the *with* statement. Verify that the file has indeed been closed, once the *with* statement exits (check the *closed* attribute on the file handle variable).

2) Modify the Timer class to work as a context manager. Essentially, it should be possible to do the following:

```
with Timer('timerName'):
    print('Do Work Here')
```

An example output would look like: **timerName: 1.5467 seconds**

The timer class should still have a configurable display. The context manager should be coded so that the following code works, to configure the display when using the context manager:

```
with Timer('timerName') as timer:
    timer.configureTimerDisplay('hrs')
    print('Do Work Here')
```

How does this compare to the previous approach of using the regular Timer class?