Lab #2: Inheritance

In this lab you will:

- Practice inheritance by designing and implementing subclasses.
- Re-visit some things you’ve already done:
  - Design and implement classes. Consult course materials from the first two weeks, or How to Think Like a Computer Scientist as needed.
  - Use the design recipe for functions and methods.
- Learn about assert statements in Python

Task 1: Event source interface

On Assignment 1, you will be reading in strings from a file, and parsing those strings into grocery store simulation events. While files are convenient for testing purposes, there are other possible sources of events.

Your main task for this lab is to generalize the idea of event sources into an abstract interface and create subclasses to achieve the following behaviour:

- Events stored in a file, one per line
- Events entered by the user during an interactive session
- Events generated randomly

For example, a design for an abstract interface with a method next, might be used like this:

```python
events = ??? # an event source constructor
event = events.next()
while event is not None:
    ... # do something with the event
event = events.next()
```

This makes it easy to switch between different event sources; in order to change the source of the events, all you need to do is use a different constructor.

This task is quite open-ended to give you an opportunity to do some design, and also get used to researching Python functionality online. You should first follow Part 1 of the Class Design Recipe to design an excellent interface for an abstract event source, and then implement the three subclasses.
Task 2: Assert statements

Consider the following version of the event client code:

```python
event = ??? # get the next event from source
tokens = event.split()

if tokens[0] == 'ARRIVE':
    if int(tokens[2]) == 1:
        suffix = 'item'
    else:
        suffix = 'items'
    print('Customer {name} arrived with {count} {suffix}'.format(
        name=tokens[1], count=tokens[2], suffix=suffix))
elif tokens[0] == 'OPEN':
    print('Checkout line {cid} is open.'.format(
        cid=tokens[1]))
```

There is a clear assumption in this code about the format of `event`.

In general, good docstrings specify what assumptions client code can safely make. However, everyone makes mistakes, both in implementation and in documentation. Therefore, it is best to make as few unchecked assumptions as possible. That way, if there is a failure, you fail fast and avoid wasting minutes or hours looking in all the wrong parts of the code.

One way to explicitly check assumptions is through `assert` statements. These statements have the following syntax:

```python
assert <bool-exp>
```

where `<bool-exp>` is any expression (including a function call) that evaluates to a boolean.

When a program encounters an assert statement that evaluates to `False`, the program immediately halts and raises an `AssertionError`.

Assert statements are generally light-weight, but they give you guarantees about properties of the code without you having to trace through the execution of a program yourself. Use them liberally when debugging!

For practice, copy the above code into your file’s “main” block, and modify it to use your event source in a loop (we’ve deliberately used different methods here to show you more than one plausible interface for an event source). Put in as many assert statements as needed to ensure that `event` has the assumed structure. You should try out how effective your code is at quickly catching errors by giving the client code erroneous input.