Lab 11: More functions

Mar 6, 2018

Main Event

1. Number machines

Write the following functions:

- **Multiplier** Given a list, returns a new list with each element of the original list duplicated:

```
>>> multiplier([1, 2, 3])
[1, 1, 2, 2, 3, 3]
```

- **Consumer** Given a list, returns a new list with the first instance of the largest element removed:

```
>>> consumer([1, 2, 5, 5, 3])
[1, 2, 5, 3]
```

- **Blender** Given a list, returns a list with a single item: the sum of all numbers:

```
>>> blender([1, 2, 3])
[6]
```

Given a particular input, use these functions produce the following output:

1. `[1, 2, 3, 4, 5, 6]` → `[50]`
2. `[2, 4, 5]` → `[121]`
3. `[1, 1, 1, 1, 5]` → `[26]`

Note that for all functions, the return values and the parameters are of the same type. Thus, these tasks can be accomplished by “chaining” the functions; for example,

```
>>> blender(consumer(multiplier([1, 2, 3])))
>>> [9]
```

For longer chains of function applications, this may create difficult code to read; in these cases creating intermediate variables is a better approach. Also, if you can identify places to use loops, you are encouraged to do so.

If you are stuck, a good strategy is to first map the individual functions to mathematical concepts. Next, understand what each produces given the original list. Finally, with this knowledge, work backwards from the expected output to come up with the sequence of function calls to get the job done. If you are keen, try to come with solutions that use fewer function calls than your neighbor.

Solution:

```
def multiplier(values):
    return values * 2

def consumer(values):
    i = values.index(max(values))
    return values[0:i] + values[i+1:]

def blender(values):
    return [ sum(values) ]

# 1
l = [1, 2, 3, 4, 5, 6]
l = blender(multiplier(consumer(multiplier(consumer(l)))))
assert(l == [50])

# 2 (soln 1)
l = [2, 4, 5]
l = multiplier(consumer(multiplier(l)))
for i in range(2):
    l = consumer(multiplier(l))
l = blender(l)
assert(l == [121])

# 2 (soln 2)
l = [2, 4, 5]
l = blender(l)
for i in range(4):
```

```
    l = multiplier(l)
for i in range(5):
    l = consumer(l)
l = blender(l)
assert(l == [121])

# 3
l = [1, 1, 1, 1, 5]
l = blender(multiplier(consumer(multiplier(l))))
assert(l == [26])
```

2. Rock-paper-scissors

To most, rock-paper-scissors is child's play. However, it's very real. Today we will teach Python to play and, for those who are diligent, answer once and for all how the game should be played.

1. Write a function that takes two hands—rock, paper, or scissors—and returns an integer signifying whether hand 1 is "better-than" hand 2.

   A *hand* is an abstract concept—it is up to you to decide which data-type to use to represent it (hint: there's no wrong answer). The function declaration, and what it returns, should obey:

   $$rps(h_1, h_2) = \begin{cases} 1 & \text{if } h_1 > h_2, \\ -1 & \text{if } h_2 > h_1, \\ 0 & \text{otherwise.} \end{cases}$$

   Here, greater-than implies that one hand is better-than the other.

2. Write another function that allows for interactive play. The function should take a single argument *rounds* denoting the number of games to be played.

   ```
   def play(rounds)
   ```

   For each game, Player 1's decision about whether to use rock, paper, or scissors should come from user input. Player 2's decision should be randomly generated. Once the specified number of games have been completed, the function should return a list of three elements, where index

   - zero is the number of games that were a tie,
   - one is the number of games won by Player 1, and
   - two is the number of games won by Player 2.

3. In the global scope, interpret the return value and print a message about the result of the game.

Solution:

```
import random

def rps(h1, h2):
    choices = [ 'rock', 'paper', 'scissors' ]
    assert(h1 in choices)
    assert(h2 in choices)

    assignment = {}
    for i in range(len(choices)):
        c = choices[i]
        assignment[c] = i

    n = len(choices)
    outcome = (assignment[h1] - assignment[h2] + n) % n
    if outcome == 2:
        outcome = -1

    return outcome

def play(rounds):
    choices = [ 'rock', 'paper', 'scissors' ]
    scoreboard = [0] * len(choices)

    i = 0
    while i < rounds:
        print('[round', i + 1, ']', end=' ')
        h1 = input('rock, paper, or scissors? ')
        if h1 not in choices:
            continue
        h2 = random.choice(choices)

        scoreboard[rps(h1, h2)] += 1

        i += 1

    return scoreboard

if __name__ == '__main__': # https://tinyurl.com/yad5jve3
```

```
    result = play(3)
    print('Player 1 won', result[0], 'times')
    print('Player 2 won', result[1], 'times')
    print(result[2], 'ties')
```

Additional Practice

1. Rock-paper-scissors: tournament

Players

Update `play` to take two "players," along with the number of rounds

```
  def play(p1, p2, rounds)
```

One way to represent players are via "function pointers." In Python, functions are first-class citizens, meaning they can be passed as arguments to other functions, returned as the values from other functions, and assigned to variables. For example:

```
>>> def f(g):
...     g('Hello')
...
>>> h = f
>>> h(print)
Hello
```

Thus, if you create two functions—one that requests user input, and another than chooses a random hand—you can effectively recreate the game-play from the previous exercise:

```
play(interactive, random_selection, 7)
```

where `interactive` and `random_selection` are functions that ask for input, and return a random hand, respectively (the choice of seven for the number of rounds was arbitrary).

Strategy

Write several functions that represent various playing styles. Remember, a “player” just a function that returns some value that corresponds to the data-type you’ve selected as hand.

Different playing styles might include

- Always playing a particular hand.
- Giving preference to a particular hand. The choice function in Python’s random number library should help with this.
- Only playing two of the three possible hands.

Tournament

What we want to find out in this exercise is whether there is a particular strategy that wins more often than others. To find out, each strategy will compete against the other in two tournament styles:

- **Round-robin** Each player plays every other player in a best-of series. Once the tournament is finished, keep some number of top-performing players.
- **Knock-out** Each player plays their neighbor; the winner moves on (stays in the list). You should be able to use, as input, the list that was returned from the round-robin tournament.

These styles should be represented as functions that take a list of strategies. Remember, functions are first-class, so you can make a list as follows:

```
def strategy_1():
    # ...

def strategy_2():
    # ...

# ...
def strategy_n():
    # ...

strategies = [
   strategy_1,
   strategy_2,
```

```
    # ...
    strategy_n
]
```

Note that during the tournament, ties do not count! In the event of a tie, the game should be played again.

Solution:

```
import random
import numpy

import rps

# updated play
def play(p1, p2, rounds):
    choices = [ 'rock', 'paper', 'scissors' ]
    scoreboard = [0] * len(choices)

    for i in range(rounds):
        result = rps.rps(p1(choices), p2(choices))
        scoreboard[result] += 1

    return scoreboard

#
# Strategy
#

def interactive(choices):
    while True:
        h1 = input('rock, paper, or scissors? ')
        if h1 in choices:
            return h1

def random_selection(choices):
    return random.choice(choices)

def always_a(choices):
    return choices[0]

def always_b(choices):
    return choices[1]
```

```
def always_c(choices):
    return choices[2]

def favor_two(choices):
    return random.choice(choices[1:])

def weighted(choices):
    return random.choices(choices, weights=[ 0.2, 0.3, 0.5 ]).pop()

#
# Tournament
#

def round_robin(players, rounds):
    wins = [0] * len(players)

    for i in range(len(players)):
        j = (i + 1) % len(players)

        p1 = players[i]
        p2 = players[j]

        result = play(p1, p2, rounds)
        for (a, b) in zip((i, j), result[:2]):
            wins[a] += b

    keep = round(len(players) / 2)
    if keep % 2:
        keep -= 1

    best_players = []
    for i in range(keep):
        x = numpy.argmax(wins)
        best_players.append(players[x])
        wins[x] = -numpy.inf

    return best_players

def knock_out(players, rounds):
    assert(len(players) % 2 == 0)

    contestants = list(range(len(players)))
```

```
    while len(contestants) > 1:
        advance = []
        stop = len(contestants) # length changed during iteration

        for i in range(0, stop, 2):
            j = i + 1
            if j > len(players) - 1:
                break
            p1 = players[i]
            p2 = players[j]

            r = rounds
            while True:
                result = play(p1, p2, r)
                if result[0] > result[1]:
                    advance.append(i)
                    break
                elif result[1] > result[0]:
                    advance.append(j)
                    break
                r = 1 # begin sudden death

        contestants = advance
    winner = contestants.pop()

    return players[winner]

players = [
    # interactive,
    random_selection,
    always_a,
    always_b,
    always_c,
    favor_two,
    weighted,
]

champion = knock_out(round_robin(players, 100), 1000)
print(champion.__name__, 'is best')
```

2. Generators

Recall that in a typical function, once `return` is encountered that function is done. Calling the function again will start the function from the `def`-ining line.

Generators are functions whose return value doesn't necessarily kill the function completely. Instead, control is returned to the caller while a pointer is maintained within the function. When the function is called again, it is run from this saved point. To differentiate this "sort of" return from an actual return, the `yield` argument is used:

```
def f():
  return [1, 2, 3]

def g():
  for i in [1, 2, 3]:
     yield i
```

Note that in the first function a list is returned. In the second, each value of the list is returned one-by-one. This makes it particularly useful in a loop:

```
  for i in g():
     print(i)
```

The advantage is that memory needs to be allocated for list members rather than the entire list, which is the case when returning a single value. This is particularly beneficial if there is a chance that iterating through the entire sequence will not be necessary. Consider a sequence with several million items, when, in most cases, only the first few thousand are of interest.

One downside, however, is that you cannot determine the length of a generator without running it to completion. Converting the generator to a list is one short cut to doing so:

```
>>> len(list(g()))
3
```

As an aside, hopefully it is now clear why directly printing the return value of range and dictionary keys do not actually return items.

To practice, develop a function that, given a CSV file, *yield*s dictionaries for each row.

Solution:

```
def lines(csv_file):
    with open(csv_file) as fp:
        header = fp.readline().strip().split(',')
        for line in fp:
            row = line.strip().split(',')
            yield dict(zip(header, row))
```

3. Static variables

Recall that we said local variables are those local to a function. Once a function is finished, those variables are gone. A static variable, on the other hand, is a variable local to a function, but whose value remains even after the function exits. They are well defined concepts in other programming languages, but not (necessarily) within Python.

However, because of the way Python “thinks of” functions, we can fake it. Notably, Python supports function definitions *inside* of other functions:

```
def f():
   def g():
     return "World"
   return "Hello " + g()
```

The concept, more generally, is known as a closure. In practice, it is more common that the outer function returns a reference to the inner function:

```
def f():
   def g():
     print("Hello World")
   return g
```

Now

```
>>> x = f()
>>> x()
Hello World
```

Update this example by making `f` and `g` accept arguments. Notice how those arguments retain their values as `f` and `g` are called. Use this knowledge to mimic static variables as defined above.

Solution:

```
def f(x):
    def g(y):
        print(x, y)
    return g

h = f('static')
for i in range(5):
    h(i)
```
