COMP10001 Foundations of Computing
Semester 2, 2022

Tutorial Solutions: Week 10

— VERSION: 1478, DATE: OCTOBER 3, 2022 —

Discussion

1. What is an "iterator"? What are some helpful methods in the `itertools` library?

   A: *An iterator is an object that keeps track of the traversal of a container. It is used by loops to keep track of iteration through a list, set, dictionary, tuple or string (these are objects we say are "iterable"). Iterators allow use of the* `next(<iterator>)` *function to progress to the next item in the iterator, and will raise a* `StopIteration` *exception if the end is reached. Note that iterators, unlike any container types, can be infinite in length.*
   `itertools` *provides many methods to construct iterators. They include* `cycle` *which produces an iterator to cycle through a container, looping from the end back to the beginning infinitely.* `product` *will combine two containers into one tuple, with each element from the first one combined with each element from the second.* `combinations` *will produce a sequence of every possible combination of elements from a container, while* `permutations` *will include combinations with different pair orderings too.* `groupby` *will group elements of a container together in particular categories based on a function parameter.*

   Now try Exercises 1

2. What is "recursion"? What makes a function recursive?

   A: *Recursion is where a function calls itself repeatedly to solve a problem. Rather than using a loop to iterate through a sequence or repeat an action, a recursive function usually calls itself with a smaller or broken-down version of the input until it reaches the answer.*

3. What are the two parts of a recursive function?

   A: *Recursive functions include a "recursive case", where the function calls itself with a reduced or simpler input; and a "base case" where the function has reached the smallest input or simplest version of the problem: it stops recursing and returns an answer.*

4. In what cases is recursion useful? Where should it be used with caution?

   A: *Recursion is useful where an iterative solution would require nesting of loops proportionate to the size of the input, such as the powerset problem or the change problem from lectures. Otherwise, there will often be an equally elegant iterative solution, and since function calls are expensive, it's often more efficient to use the iterative approach. Some algorithms you will learn about in future subjects depend on recursion, and it can be a powerful technique when trying to sort data.*

   Now try Exercise 2

Exercises

1. What output does the following code print?

```
import itertools
beatboxer = itertools.cycle(['boots', 'and', 'cats', 'and'])

for count in range(39):
    print(next(beatboxer))
```

   A: *This code will print ten iterations of* `boots and cats and` *which will end with* `cats`*:*

```
boots
and
cats
and
boots
...
and
boots
and
cats
```

   Try changing the `for` *loop to* `while True:` *(an infinite loop) to see this cycle print infinitely!*

2. Study the following mysterious functions. For each one, answer the following questions:

- Which part is the base case?
- Which part is the recursive case?
- What does the function do?

(a)
```
def mystery(x):
    if len(x) == 1:
        return x[0]
    else:
        y = mystery(x[1:])
        if x[0] > y:
            return x[0]
        else:
            return y
```

A: *The* `if` *block is the base case, and the* `else` *block is the recursive case. The function returns the largest element in the list/tuple.*

(b)
```
def mistero(x):
    a = len(x)
    if a == 1:
        return x[0]
    else:
        y = mistero(x[a//2:])
        z = mistero(x[:a//2])
        if z > y:
            return z
        else:
            return y
```

A: *The* `if` *block is the base case, and the* `else` *block is the recursive case. Like (a), this function returns the largest element in the list/tuple. This function uses two recursive calls, while the first uses one. There's no difference in the calculated output.*

Project 2

Today we'll be looking at two key functions in project 2: `proliferate` and `breed`. Let's start with a population of two wugs:

```
characteristics = ["intelligence", "beauty", "strength", "speed"]
superwug_genome = [1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
gene_zones      = [2, 1, 2, 3, 3, 1, 3, 3, 0, 0, 2, 2, 0, 1, 0, 1]

wug1            = ([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
wug2            = ([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1], 'M')
population = [wug1, wug2]
```

1. What superior characteristics do these wugs have? What `rank` would we assign to each wug?

   A: `wug1` *is a superwug with features* `[True, True, True, True]`*, therefore rank 4. This means that it has superior intelligence, beauty, strength and speed.*
   `wug2` *is has features* `[True, False, True, True]`*, therefore rank 3. This means that it has superior intelligence, strength and speed and normal beauty.*

2. The first step in `proliferate` is to work out how to clone each wug. What would the 16 clones of `wug1` look like? What are their ranks?

A: *The 16 clones are displayed below - each with 1 "genetic bit" flipped from the original genome. This means that each clone must have one less superior characteristic than the original wug: They are all of rank 3.*

```
([0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0], 'F')
([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1], 'F')
```

3. Now we call `proliferate` and observe the resulting population. Can you see which clones were added to the population?

```
proliferate(population, limit=5)
for wug in population:
    print(wug, rank(wug))

>>> ([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F') 4
>>> ([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'M') 4
>>> ([1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1], 'M') 3
>>> ([0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F') 3
>>> ([1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], 'F') 3
```

A: *`wug1` and `wug2` remain in the population. This is not always the case, but simply because they had high ranks (4 and 3 respectively), and population insertion respects chronological order. This means that when the population reached the limit (of 5), any new clones with rank $\leq$ to the the lowest ranking wug would not be added to the population. After all of the `wug1` clones were inserted, then the `wug2` clones were inserted. Note that one of the male clones became a superwug and was also inserted into the population due to its high rank, the other males were not added.*

4. The first step in the `breed` function is to score the suitability of mates. From the perspective of `wug1`, the suitability of `wug2` is calculated as `wug2`'s rank plus a `coincidence_bonus` times the number of features they share. Using `coincidence_bonus = -2`, what would this score be?

   A: `rank(wug2) + coincidence_bonus*overlapping_features = 3 - 2*3 = -3`

Problems

1. Write a recursive function which takes an integer `n` and calculates the n^{th} fibonacci number. The 0^{th} fibonacci number is 0, the 1^{st} fibonacci number is 1 and all following fibonacci numbers are defined as the sum of the preceding two fibonacci numbers. `fib(10)` should return `55`

   A:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

2. Write a function which takes two strings as input and uses an `itertools` iterator to find whether the first word is an anagram of the second word. This might not be a very efficient way to find an anagram but it will help us work with iterators! `anagram('astronomer', 'moonstarer')` should return `True`

A:

```
from itertools import permutations

def anagram(word1, word2):
    for ordering in permutations(word1, len(word1)):
        if "".join(ordering) == word2:
            return True
    return False
```

3. Write a function which takes a lowercase string as input and prints the frequency of each vowel in the string using a dictionary. `vowel_counts('i love python')` should print:

   ```
   i 1
   e 1
   o 2
   ```

   A:

   ```
   def vowel_count(text):
       vowel_counts = {}
       # Count vowel frequencies
       for letter in text:
           if letter in 'aeiou':
               if letter in vowel_counts:
                   vowel_counts[letter] += 1
               else:
                   vowel_counts[letter] = 1

       for vowel, count in vowel_counts.items():
           print(vowel, count)
   ```

4. Write a function which takes two lists of integers and returns the average of the numbers which they both have in common. `in_common_average([1, 2, 3, 4, 5], [0, 2, 4, 6])` should return `3.0`

   A:

   ```
   def in_common_average(list1, list2):
       common = set(list1) & set(list2)
       return sum(common) / len(common)
   ```