recursive_functions

February 16, 2024

[1]:
```
# Consider wanting to queue up for a ride in a roller coaster. The queue,␣
 ↪however, is so long
# you can't determine how many there are at first glance.

# What can you do?
```

[2]:
```
#
# You could ask the last person in the queue and ask: how many are in front of␣
 ↪you?
# If there is someone in front of that person, he responds: Hold on, I am gonna␣
 ↪ask the person
# in front of me. Is there no one in front of him, he responds with: I am the␣
 ↪first in line.
# This asking the person in front of you propagates through the entire queue␣
 ↪until the person in front is found.

# The responses are now propagated backwards:

# The person in front: Here is no one in front of me.
# The second to last person responds with: there is (1 + 0) people ahead of me.
# The third to last person responds with there are (1 + 1) people ahead of me.
# ...
# Until you reach back to the end of line.
```

[3]:
```
# The asking "how many are in front of you" is our recursive function. We keep␣
 ↪calling our function on
# the remainder of the queue until the queue is empty - we have reached the␣
 ↪base case and do not invoke
# our function again.
```

[4]:
```
# Let's look at an implementation of our queue problem:
queue = ["Pete", "Alex", "George", "Mike", "Fiona", "Linn", "Thomas"]



def how_many_in_front(queue):
    if len(queue) == 0:
```

```
        return 0
    else:
        return 1 + how_many_in_front(queue[1:])

how_many_in_queue = how_many_in_front(queue)
print(how_many_in_queue)
```

```
7
```

[5]:
```
# The function is called 7 times!
# Whenever we call our function, that call gets added to the call stack.
# As long as a function call involves calling itself, that call is added on top␣
 ↪of the call stack.
# As soon as a function call does not involve calling itself, we can pop that␣
 ↪from the call stack
# and provide the return value to the function call below.
# We do this (python does this) until the call stack is empty and we get our␣
 ↪final result.
```

[6]:
```
# More recursion:

def spell(word):
    if len(word) == 1:
        print(word[0])
    else:
        print(word[0])
        spell(word[1:])

spell("Simon")

def spell_iterative(word):
    for char in word:
        print(char)

spell_iterative("Simon")
```

```
S
i
m
o
n
S
i
m
o
n
```

[ ]: