hw02

November 19, 2021

[16]:
```
#Q1: Num eights
"""
Write a recursive function num_eights that takes a positive integer x and
returns the number of times the digit 8 appears in x.
Use recursion - the tests will fail if you use any assignment statements.
"""
def num_eights(x):
    """Returns the number of times 8 appears as a digit of x.

    >>> num_eights(3)
    0
    >>> num_eights(8)
    1
    >>> num_eights(88888888)
    8
    >>> num_eights(2638)
    1
    >>> num_eights(86380)
    2
    >>> num_eights(12345)
    0
    >>> from construct_check import check
    >>> # ban all assignment statements
    >>> check(HW_SOURCE_FILE, 'num_eights',
    ...       ['Assign', 'AugAssign'])
    True
    """
    "*** YOUR CODE HERE ***"
    #base case
    if x < 10:
        if x == 8:
            return 1
        else:
            return 0
    #recursive case
    return num_eights(x%10) + num_eights(x//10)
```

[17]:
```
print(num_eights(3) == 0)
print(num_eights(8) == 1)
print(num_eights(88888888) == 8)
print(num_eights(2638) == 1)
print(num_eights(86380) == 2)
print(num_eights(12345) == 0)
```

```
True
True
True
True
True
True
```

[18]:
```
#Q2: ping-pong
"""
The ping-pong sequence counts up starting from 1 and is always either counting␣
 ↪up or counting down.
At element k, the direction switches if k is a multiple of 8 or contains the␣
 ↪digit 8.
The first 30 elements of the ping-pong sequence are listed below, with␣
 ↪direction swaps marked using
brackets at the 8th, 16th, 18th, 24th, and 28th elements:
Implement a function pingpong that returns the nth element of the ping-pong␣
 ↪sequence without using
any assignment statements.
You may use the function num_eights, which you defined in the previous question.
Use recursion - the tests will fail if you use any assignment statements.
"""

def pingpong(n):
    """Return the nth element of the ping-pong sequence.

    >>> pingpong(8)
    8
    >>> pingpong(10)
    6
    >>> pingpong(15)
    1
    >>> pingpong(21)
    -1
    >>> pingpong(22)
    -2
    >>> pingpong(30)
    -2
    >>> pingpong(68)
    0
```

```
    >>> pingpong(69)
    -1
    >>> pingpong(80)
    0
    >>> pingpong(81)
    1
    >>> pingpong(82)
    0
    >>> pingpong(100)
    -6
    >>> from construct_check import check
    >>> # ban assignment statements
    >>> check(HW_SOURCE_FILE, 'pingpong', ['Assign', 'AugAssign'])
    True
    """
    "*** YOUR CODE HERE ***"
    #base case
    if n <= 8:
        return n
    #recursive base
    return direction(n) + pingpong(n-1)

def direction(x):
    if x < 8:
        return 1
    if (x-1) % 8 == 0 or num_eights(x-1) > 0:
        return -1 * direction(x-1)
    return direction(x-1)
```

[19]:
```
print(pingpong(8) == 8)
print(pingpong(10) == 6)
print(pingpong(15) == 1)
print(pingpong(21) == -1)
print(pingpong(22) == -2)
print(pingpong(30) == -2)
print(pingpong(68) == 0)
print(pingpong(69) == -1)
print(pingpong(80) == 0)
print(pingpong(81) == 1)
print(pingpong(82) == 0)
print(pingpong(100) == -6)
```

```
True
True
True
True
True
True
```

```
True
True
True
True
True
True
True
```

[20]:
```
#Q3: Missing Digits
"""
Write the recursive function missing_digits that takes a number n that is sorted in increasing order
(for example, 12289 is valid but 15362 and 98764 are not).
It returns the number of missing digits in n. A missing digit is a number between the first and last
digit of n of a that is not in n.
Use recursion - the tests will fail if you use while or for loops.
"""
def missing_digits(n):
    """Given a number a that is in sorted, increasing order,
    return the number of missing digits in n. A missing digit is
    a number between the first and last digit of a that is not in n.
    >>> missing_digits(1248) # 3, 5, 6, 7
    4
    >>> missing_digits(1122) # No missing numbers
    0
    >>> missing_digits(123456) # No missing numbers
    0
    >>> missing_digits(3558) # 4, 6, 7
    3
    >>> missing_digits(35578) # 4, 6
    2
    >>> missing_digits(12456) # 3
    1
    >>> missing_digits(16789) # 2, 3, 4, 5
    4
    >>> missing_digits(19) # 2, 3, 4, 5, 6, 7, 8
    7
    >>> missing_digits(4) # No missing numbers between 4 and 4
    0
    >>> from construct_check import check
    >>> # ban while or for loops
    >>> check(HW_SOURCE_FILE, 'missing_digits', ['While', 'For'])
    True
    """
    "*** YOUR CODE HERE ***"
    #base case
```

```
    if n < 10:
        return 0
    last, rest = n % 10, n // 10
    #recursive case
    #89: no missing number
    #88: no missing number
    #53: missing 1
    #52: missing 2
    return missing_digits(rest) + max(last - (rest % 10 + 1), 0)
```

[21]:
```
print(missing_digits(1248) == 4)
print(missing_digits(1122) == 0)
print(missing_digits(123456) == 0)
print(missing_digits(3558) == 3)
print(missing_digits(35578) == 2)
print(missing_digits(12456) == 1)
print(missing_digits(16789) == 4)
print(missing_digits(19) == 7)
print(missing_digits(4) == 0)
```

```
True
True
True
True
True
True
True
True
True
```

[22]:
```
#Q4: Count coins
"""
Given a positive integer total, a set of coins makes change for total if the␣
 ↪sum of the values of
the coins is total. Here we will use standard US Coin values: 1, 5, 10, 25 For␣
 ↪example,
the following sets make change for 15:
15 1-cent coins
10 1-cent, 1 5-cent coins
5 1-cent, 2 5-cent coins
5 1-cent, 1 10-cent coins
3 5-cent coins
1 5-cent, 1 10-cent coin
Thus, there are 6 ways to make change for 15.
Write a recursive function count_coins that takes a positive integer total and␣
 ↪returns the number
of ways to make change for total using coins.
```

```
Use the next_largest_coin function given to you to calculate the next largest␣
↪coin denomination
given your current coin. I.e. next_largest_coin(5) = 10.
"""
def next_largest_coin(coin):
    """Return the next coin.
    >>> next_largest_coin(1)
    5
    >>> next_largest_coin(5)
    10
    >>> next_largest_coin(10)
    25
    >>> next_largest_coin(2) # Other values return None
    """
    if coin == 1:
        return 5
    elif coin == 5:
        return 10
    elif coin == 10:
        return 25

def count_coins_helper(total, smallest):
    #base case
    if total < 0:
        return 0
    if total == 0:
        return 1
    if smallest == None:
        return 0

    #recursive case
    without_coin = count_coins_helper(total, next_largest_coin(smallest))
    with_coin = count_coins_helper(total-smallest,smallest)
    return without_coin + with_coin

def count_coins(total):
    """Return the number of ways to make change for total using coins of value␣
↪of 1, 5, 10, 25.
    >>> count_coins(15)
    6
    >>> count_coins(10)
    4
    >>> count_coins(20)
    9
    >>> count_coins(100) # How many ways to make change for a dollar?
    242
    >>> from construct_check import check
```

```
    >>> # ban iteration
    >>> check(HW_SOURCE_FILE, 'count_coins', ['While', 'For'])
 ↪
    True
    """
    "*** YOUR CODE HERE ***"
    return count_coins_helper(total, 1)
```

[23]:
```
print(count_coins(15) == 6)
print(count_coins(10) == 4)
print(count_coins(20) == 9)
print(count_coins(100) == 242)
```

```
True
True
True
True
```

[24]:
```
#Q5: Anonymous factorial
"""
The recursive factorial function can be written as a single expression by using
a conditional expression.

>>> fact = lambda n: 1 if n == 1 else mul(n, fact(sub(n, 1)))
>>> fact(5)
120
However, this implementation relies on the fact (no pun intended) that fact has
 ↪a name,
to which we refer in the body of fact. To write a recursive function,
we have always given it a name using a def or assignment statement so that we
 ↪can refer to
the function within its own body. In this question, your job is to define fact
 ↪recursively
without giving it a name!

Write an expression that computes n factorial using only call expressions,
conditional expressions, and lambda expressions (no assignment or def
 ↪statements).
Note in particular that you are not allowed to use make_anonymous_factorial in
 ↪your return expression. The sub and mul functions from the operator module
 ↪are the only built-in functions required to solve this problem:
"""

from operator import sub, mul

def make_anonymous_factorial():
    """Return the value of an expression that computes factorial.
```

```
    >>> make_anonymous_factorial()(5)
    120
    >>> from construct_check import check
    >>> # ban any assignments or recursion
    >>> check(HW_SOURCE_FILE, 'make_anonymous_factorial', ['Assign',␣
 ↪'AugAssign', 'FunctionDef', 'Recursion'])
    True
    """
    return lambda n: (lambda f: f(n,f))(lambda n,fact: 1 if n == 1 else mul(n,␣
 ↪fact(n-1, fact)))
```

[25]:
```
print(make_anonymous_factorial()(5) == 120)
```

```
True
```

[26]:
```
#Towers of Hanoi
def print_move(origin, destination):
    """Print instructions to move a disk."""
    print("Move the top disk from rod", origin, "to rod", destination)

def move_stack(n, start, end):
    """Print the moves required to move n disks on the start pole to the end
    pole without violating the rules of Towers of Hanoi.

    n -- number of disks
    start -- a pole position, either 1, 2, or 3
    end -- a pole position, either 1, 2, or 3

    There are exactly three poles, and start and end must be different. Assume
    that the start pole has at least n disks of increasing size, and the end
    pole is either empty or has a top disk larger than the top n start disks.

    >>> move_stack(1, 1, 3)
    Move the top disk from rod 1 to rod 3
    >>> move_stack(2, 1, 3)
    Move the top disk from rod 1 to rod 2
    Move the top disk from rod 1 to rod 3
    Move the top disk from rod 2 to rod 3
    >>> move_stack(3, 1, 3)
    Move the top disk from rod 1 to rod 3
    Move the top disk from rod 1 to rod 2
    Move the top disk from rod 3 to rod 2
    Move the top disk from rod 1 to rod 3
    Move the top disk from rod 2 to rod 1
    Move the top disk from rod 2 to rod 3
    Move the top disk from rod 1 to rod 3
```

```
    """
    assert 1 <= start <= 3 and 1 <= end <= 3 and start != end, "Bad start/end"
    if n == 1:
        print_move(start, end)
    else:
        spare = 6 - start - end
        move_stack(n-1, start, spare)
        print_move(start, end)
        move_stack(n-1, spare, end)
```

[27]:
```
move_stack(3, 1, 3)
```

```
Move the top disk from rod 1 to rod 3
Move the top disk from rod 1 to rod 2
Move the top disk from rod 3 to rod 2
Move the top disk from rod 1 to rod 3
Move the top disk from rod 2 to rod 1
Move the top disk from rod 2 to rod 3
Move the top disk from rod 1 to rod 3
```

[ ]:
```

```