

Ch13-Recursion

September 10, 2025

1 Recursion

- defining something in terms of itself usually at some smaller scale, perhaps multiple times, to achieve your objective
 - e.g., a human being is someone whose mother is a human being
 - directory is a structure that holds files and (smaller) directories, etc.
 - fractals is a drawing which also has self-similar structure, where it can be defined in terms of itself
- in programming, functions can generally call themselves to solve similar but smaller subproblems
 - this technique is called recursion

1.1 Definitions

- **Recursion:** The process of solving a problem by reducing it to smaller versions of itself
- **Recursive definition:** a definition in which something is defined in terms of smaller version of itself
- **Recursive algorithm:** an algorithm that finds a solution to a given problem by reducing the problem to smaller versions of itself
- **Infinite recursion:** recursive call that never stops

1.1.1 general construct of recursive algorithms

- recursive algorithms/solutions have base case(s) and general case(s):
 1. must have one or more base case(s)
 - provides direct answer that makes the recursion stop
 - answer to the smallest subproblem
 2. must have one or more general case(s)
 - functions recursively reduce themselves to one of the base case(s)

1.1.2 Python Recursion Depth Limit

- by default Python limits recursion depth to something < 3000
- you can set it up higher if needed

```
[1]: import sys
```

```
[2]: sys.getrecursionlimit()
```

```
[2]: 3000
```

```
[3]: sys.setrecursionlimit(1_000_000)
```

```
[4]: sys.getrecursionlimit()
```

```
[4]: 1000000
```

1.1.3 recursive countdown example

```
[5]: # Recursively print countdown from 10-1 and blast off!  
# Run it as a script  
import time  
def countDown(n):  
    if n == 0:  
        print('Blast Off!')  
        time.sleep(1)  
    else:  
        print(n)  
        time.sleep(1)  
        countDown(n-1)  
        #print(n)
```

```
[6]: countDown(10)
```

```
10  
9  
8  
7  
6  
5  
4  
3  
2  
1  
Blast Off!
```

1.1.4 Sum of First N positive integers

- use recursion to find the sum of first N positive integers

first_sum(1) = 1 (base case)

first_sum(n) = n + first_sum(n-1) for n > 1 (general case)

```
[1]: def first_sum(n):  
    if n == 1:  
        return 1  
    else:
```

```
return n + first_sum(n-1)
```

```
[2]: first_sum(100)
```

```
[2]: 5050
```

```
[3]: # use tail recursion optimization  
# Python doesn't optimize tail recursion  
# read the blog by Guido van Rossum - http://neopythonic.blogspot.com/2009/04/  
↪tail-recursion-elimination.html  
def sum_tail(n, running_tot=1):  
    if n == 1:  
        return running_tot  
    else:  
        return sum_tail(n-1, running_tot+n)
```

```
[6]: sum_tail(100, 1)
```

```
[6]: 5050
```

1.2 Fibonacci numbers

- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- devised by Fibonacci (1170-1250), who used the sequence to model the breeding of (pairs) of rabbits
- say in generation 7 you had 21 pairs in total, of which 13 were adults, then in next generation the adults will have bred new children, and the previous children will have grown up to become adults. So, in generation 8, you'll have 13+21=34 rabbits, of which 21 are adults.

1.2.1 Fibonacci number definition

fib(0) = 0 (base case 1)

fib(1) = 1 (base case 2)

fib(n) = fib(n-1) + fib(n-2) for n >= 2 (general case)

```
[8]: # Finding Fibonacci number in series  
count = 0  
def fib(n):  
    global count  
    count += 1  
    if n <= 1:  
        return n  
    f = fib(n-1) + fib(n-2)  
    return f
```

```
[9]: fib(32)  
print(f'fib was called {count} times.')
```

```
assert fib(8) == 21
```

```
assert fib(10) == 55
```

fib was called 7049155 times.

1.2.2 visualize fib(4) using pythontutor.com

- <https://goo.gl/YNizhH>

1.2.3 You can timeit

- use timeit function to see how long fib(n) takes

```
[10]: from timeit import timeit
```

```
[11]: timeit("fib(32)", globals=globals(), number=1)
```

```
[11]: 1.1521884420071729
```

1.2.4 how many times is fib() called for fib(4)?

- Modify fib to count the number of times fib gets called.
- Time complexity of recursive fib function is $O(2^n)$ or precisely $O(1.6180^n)$
 - 1.6180 is also called golden ratio

```
[12]: def fib_tail(n, a, b):  
        if n == 0:  
            return a  
        if n == 1:  
            return b  
        return fib_tail(n-1, b, a+b)
```

```
[13]: print(fib_tail(32, 0, 1))
```

2178309

```
[14]: timeit("fib_tail(32, 0, 1)", globals=globals(), number=1)
```

```
[14]: 1.3885000953450799e-05
```

1.2.5 Factorial definition

$1! = 1$ (base case)
 $n! = n * (n-1)!$ for $n > 1$ (general case)

- Exercise - Implement factorial recursive solution; see homework assignment!

1.3 Drawing Fractals

- for our purposes, a fractal is a drawing which also has self-similar structure, where it can be defined in terms of itself.
- see Koch fractal example in text book section 18.1

1.3.1 an animated fractal, using PyGame library

- install PyGame library using the following bash script!
- if PyGame is already installed and is the current version, there's no harm!
- we use pip to install Python packages; so update pip just in case

```
[ ]: %%bash
pip install --upgrade pip
pip install pygame
```

```
[ ]: # animated fractal; when done just close the window or force kill if not
↳responding!
# this program doesn't run in colab or online services. You must run locally
↳from notebook or as a script!

import pygame, math
pygame.init()           # prepare the pygame module for use

# Create a new surface and window.
surface_size = 1024
main_surface = pygame.display.set_mode((surface_size,surface_size))
my_clock = pygame.time.Clock()

def draw_tree(order, theta, sz, posn, heading, color=(0,0,0), depth=0):

    trunk_ratio = 0.29      # How big is the trunk relative to whole tree?
    trunk = sz * trunk_ratio # length of trunk
    delta_x = trunk * math.cos(heading)
    delta_y = trunk * math.sin(heading)
    (u, v) = posn
    newpos = (u + delta_x, v + delta_y)
    pygame.draw.line(main_surface, color, posn, newpos)

    if order > 0:    # Draw another layer of subtrees

        # These next six lines are a simple hack to make the two major halves
        # of the recursion different colors. Fiddle here to change colors
        # at other depths, or when depth is even, or odd, etc.
        if depth == 0:
            color1 = (255, 0, 0)
            color2 = (0, 0, 255)
        else:
            color1 = color
            color2 = color

        # make the recursive calls to draw the two subtrees
        newsz = sz*(1 - trunk_ratio)
```

```

        draw_tree(order-1, theta, newsz, newpos, heading-theta, color1, depth+1)
        draw_tree(order-1, theta, newsz, newpos, heading+theta, color2, depth+1)

def gameloop():

    theta = 0
    while True:

        # Handle evente from keyboard, mouse, etc.
        ev = pygame.event.poll()
        if ev.type == pygame.QUIT:
            break;

        # Updates - change the angle
        theta += 0.01

        # Draw everything
        main_surface.fill((255, 255, 0))
        draw_tree(9, theta, surface_size*0.9, (surface_size//2,
↪surface_size-50), -math.pi/2)

        pygame.display.flip()
        my_clock.tick(120)

gameloop()
pygame.quit()

```

1.3.2 exercise 1

Write a recursive fact(n) function that takes a positive integer n and returns its factorial.

1.3.3 exercise 2

Write a recursive function – gcd(a, b) – that finds the greatest common divisor of two given positive integers, a and b. - see algorithm [in Khan Academy](#)

1.3.4 exercise 3

Write a program that simulates the steps required to solve the “Tower of Hanoi” puzzle for some disks n. - <https://www.mathsisfun.com/games/towerofhanoi.html>

- Recursive algorithm
- If there are 1 or more disks to move:
 1. Move the top n-1 disks from needle 1 (source) to needle 2 (helper), using needle 3 (dest) as the intermediate needle
 2. Move disk number n from needle 1 (src) to needle 3 (dest)

3. Move the top $n - 1$ disks from needle 2 (helper) to needle 3 (dest), using needle 1 (src) as the intermediate needle

```
[ ]: def moveDisks(n, src, helper, dst):  
    if n > 0:  
        moveDisks(n-1, src, dst, helper)  
        print('Move disk #{} from {} to {}'.format(n, src, dst))  
        moveDisks(n-1, helper, src, dst)
```

```
[ ]: moveDisks(3, 'needle1', 'needle2', 'needle3')
```

1.4 Kattis problems

- the following Kattis problems can be solved using recursion:
 1. Last Factorial Digit: <https://open.kattis.com/problems/lastfactorialdigit>
 2. Watch Out For Those Hailstones! - <https://open.kattis.com/problems/hailstone>
 3. Of of Sorts - <https://open.kattis.com/problems/outofsorts>
 - Hint: implement recursive binary search and apply to the generated unsorted sequence

```
[ ]:
```