STMC HKOI Training

Lesson 8: Function and Recursion

Chan Yan Mong

April 9, 2022

Goal today

String and list are similar so we can do similar operations (Slice, membership, elementwise access etc.) on it.

- Introduce function to modularize code
- Using function to make previous code cleaner
- Introduce recursion
- Use recursion to solve problems:
  - Fibonacci problem
  - Tower of Hanoi
  - Permutations

Code Reusability

- **Code reuse** is the use of existing software, or software knowledge, to build new software
- Many times in programming, certain segment of code (e.g. search for name in contact for example) will be used again and again
- We would therefore want to reuse those code without keep typing it again and again
- Here is an example:

```python
# Searching for names in contact

contact = ['Billy','Damon','Leon','Mary','Shirley']
searchName = input('Enter a name: ')
inContact = False
for name in contact:
  if searchName == name:
    inContact = True
if inContact:
  print(f'{searchName} is in contact')

# Search another one
searchName = input('Enter a name again: ')
inContact = False
for name in contact:
  if searchName == name:
    inContact = True
if inContact:
  print(f'{searchName} is in contact')
```

Code Reusability

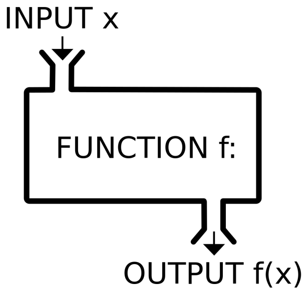
- As shown in the code, line 13 to 19 is completely identical to the above
- It would be great if we can put them into one line like this:

```python
if haveName(contact, name):
  print(f'{name} is in contact')
```

- *function* provide exactly what we want

Function

- **Function**, in the simplest term, is a machine that takes in some input x and perform some action on x, producing an output $f(x)$
- For example, we may define the following functions:
  - $f(x) = x^2$
  - $f(x) = x^x$
  - $f(x) =$ Last digit of x in decimal
  - $f(x) = \{x \quad \text{if} \quad x \geq 0, -x \quad \text{if} \quad x < 0\}$
  - $f(x, y) = xy^2 + x^2y^4$



Function

- To *evaluate* a function, we put in an input and perform the actions as described by the definition of the f. For example:

		$f(x)$	
x	x^2	x^x	x if $x \geq 0$, $-x$ otherwise
-1	$(-1)^2$	$(-1)^{(-1)}$	1
1	$(1)^2$	$(1)^{(1)}$	1
2	$(2)^2$	$(2)^{(2)}$	2
11	$(11)^2$	$(11)^{(11)}$	11

Function

- By defining functions, we can simply expressions by ”calling” them in our expressions
- For example, $\text{lerp}(a, b, t) = a(1-t) + bt \quad 0 \leq t \leq 1$ performs a linear interpolation from a to b
- By nesting lerp we can perfrom complicated interpolation

$$f(a, b, c, t) = \text{lerp}(\text{lerp}(a, b, t), \text{lerp}(b, c, t), t)$$

Functions in programming

- In programming, functions are ”self contained” modules of code that accomplish a specific task.
- A function takes in certain inputs called the **arguments** and produce outputs called **return value**
- For example, you may write a `UserLookup` function that takes in the username as argument, and return user info as return value.

Function in programming

- In python, a function have the following syntax

```
def func(arg1,arg2,...):
  # Operations
  return val1,val2,...
```

- For example, the `lerp` function just now:

```
def lerp(a,b,t):
  return (1-t)*a + t*b
```

Function Exercises

Complete the following exercises

1. Write a function `Square(x)` that square a number
2. Write a function `IsBigger(x,y)` that compares two number *x*,*y* and return True if $x > y$
3. Write a function `HaveSubstring(str,substr)` that returns True if `substr` is contained in `str` and False otherwise
4. Write a function `SumAll(num_list)` that return the sum of a list of numbers `num_list`
5. Write a function `MinMax(num_list)` that returns in maximum and mininum number from the list `num_list`

Application: Tic-Tac-Toe

- To demonstrate how function can be used to simply code, we will write a *Tic-Tac-Toe* game
- In a Tic-Tac-Toe game, people place `X` and `O` in turns until a winning position is obtained
- Let's be more specific and write down the actual game flow

Application: Tic-Tac-Toe

Here is a possible flow of the game:

1. Prepare an empty 3×3 grid. Assume the game starts with `X`
2. Each round, a player enter a pair of numbers (row, col) to specify the row and column he or she wants to place the `X` (or `O`)
3. If the position (row, col) is occupied, go back to 2 and ask for re-entry; Otherwise continue
4. Check if ending position is reached. If so declare the winner and end the game; Otherwise continue
5. Change the player from `X` to `O` (or `O` to `X`) and go back to 2

With that we can lay down the structure of our code:

```python
def main():
  grid = [[0,0,0],[0,0,0],[0,0,0]] # Empty 3x3 list
  playerNow = 'X'

  while IsEndPosition(grid) == False:
      row, col = int(input('Row: ')), int(input('Col: '))
      if not Grid_InRange(row,col):
          print(f'The position ({row},{col}) is not in range!')
      elif not Grid_IsEmpty(row,col,grid):
          print(f'The position ({row},{col}) is occupied!')
      else:
          grid[row][col] = playerNow
          playerNow = GetNextPlayer(playerNow)
          Print_Grid(grid)

  winner = IsEndPosition(grid)
  if winner == 'Tie':
      print('Tie!')
  else:
      print(f'{winner} is the winner')
  input()
```

Application: Tic-Tac-Toe

- But wait, what are `IsEndPosition`, `Grid_IsEmpty`, `GetNextPlayer`, `Print_Grid` and `Grid_InRange`
- They are *functions* to be defined
- As one can see, function allow us to break a program into smaller parts that can be implemented individually
- This makes the code cleaner and more maintainable

For example, we can implement `Print_Grid` like this

```
def Print_Grid(grid):
  for i in range(3):
      for j in range(3):
          if grid[i][j] != 0:
              print(grid[i][j],end='')
          else:
              print('-',end='')
      print('')
```

ans similarly:

```
def Grid_IsEmpty(row,col,grid):
  return grid[row][col] == 0

def Grid_InRange(row,col):
  return 0<=row and row <=2 and 0 <= col and col <= 2

def GetNextPlayer(playerNow):
  if playerNow == 'X':
      return 'O'
  else:
      return 'X'
```

For the implementation of `IsEndPosition`, please refer to the code posted online. Here we will look at some core logic. Basically, we want to check if the grid configuration is in any of these ending positions:

- 3 identical `X` or `O` along row
- 3 identical `X` or `O` along column
- 3 identical `X` or `O` along diagonal
- All cells are filled but it is not in any of the ending positions above

The first two can be done in something like this:

```
for i in range(3):
  # Check all rows
  if grid[i][0] == grid[i][1] and grid[i][0] == grid[i][2] and grid[
    i][0] != 0:
      return grid[i][0]
  # Check all columns
  if grid[0][i] == grid[1][i] and grid[0][i] == grid[2][i] and  grid
    [0][i] != 0:
      return grid[0][i]
```

The third one can be done like this:

```
# Check Diagonals
if grid[0][0] == grid[1][1] and grid[0][0] == grid[2][2] and grid[1][1] != 0:
    return grid[1][1]
if grid[0][2] == grid[1][1] and grid[0][2] == grid[2][0] and grid[1][1] != 0:
    return grid[1][1]
```

and the last one can be checked by using this code after excluding all the three cases above:

```
# Check if all cells filled
isFull = True
for i in range(3):
    for j in range(3):
        if grid[i][j] == 0:
            isFull = False
            break
```

Recursion

- Function also allow us to easily implement algorithms that involve **recursion**
- That is, solutions that depends on solutions to smaller instances of the same problem
- As we will illustrate, recursion sometimes allow us to solve complicated problems in very neat way

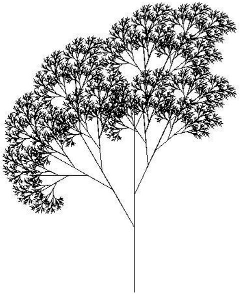


Figure 1: Recursive Tree (Source)

Example: Fibonacci Sequence

- Recall the Fibonacci sequence that we discuss long time ago

$$F(n+2) = F(n+1) + F(n)$$
$$F(1) = F(2) = 1$$

- We had implemented that using loops before, we will now try to do it in recursion

Example: Fibonacci Sequence

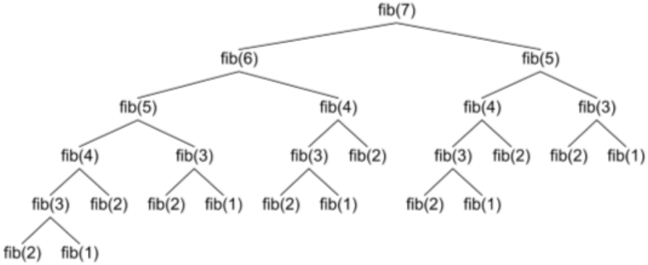
```python
# Fibonacci sequence using recursion
def Fib(n):
  if n == 1 or n == 2:
    return 1
  else:
    return Fib(n-1) + Fib(n-2)

```

Example: Fibonacci Sequence

- As shown in the code, calling $F(4)$ returns $F(3) + F(2)$
- To return $F(3) + F(2)$, we must evaluate $F(3)$ and $F(2)$
- $F(2)$ by line 3,4 of the code returns 1
- $F(3)$ on the otherhand calls $F(1) + F(2)$, which both evaluates to 1 according to line 3,4
- So we ”backsubstitute” the values layer by layer up and evaluate $F(4) = ((1 + 1)) + (1) = 3$

Example: Fibonacci Sequence

- To see what happens for larger *n*, we can consider the following figure



Source: StackOverflow

Take away

This simple exmaple in illustrate some general features of recursive algorithm

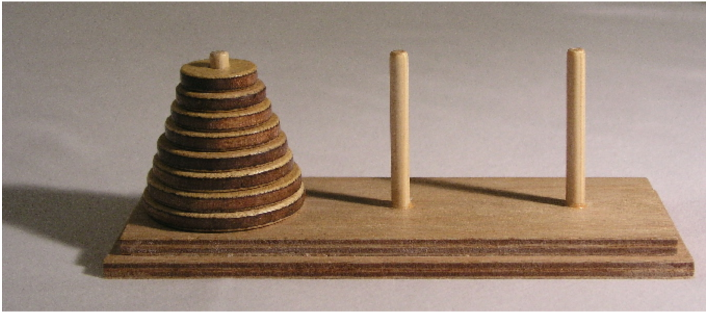
- In recursion, we often call a function inside itself (e.g. Here we call $F(3), F(2)$ inside $F(4)$)
- Solutions that depends on solutions to smaller instances of the same problem (e.g. Finding $F(4)$ involves finding $F(3)$,$F(2)$)
- We combine solutions of smaller problems to solve a larger problem

Example: Tower of Hanoi

- The Tower of Hanoi is a mathematical puzzle first introduced by Édouard Lucas in 1883
- It begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape.
- The objective of the puzzle is to move the entire stack to the last rod, obeying the following rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a disk that is smaller than it.

Example: Tower of Hanoi



Example: Tower of Hanoi

- We would like to program an algorithm that would *solves* the tower of hanoi
- Given a stack of *n* disk to begin with, derive a program that will teach you how should you move the disks to solve the puzzle
- Let's try to play around with the game to gain some insights and familiarity
- Here is a link to the game: https://www.mathsisfun.com/games/towerofhanoi.html

Example: Tower of Hanoi

Observations:

Suppose there are n disks and suppose we know how to move $n-1$ disks from rod 1 to rod 2. Then to move n disks, we just need to:

1. Move all $n-1$ disk above to the rod in the middle rod
2. Place the bottom disk to the final rod
3. Move the $n-1$ disks to the final rod

Example: Tower of Hanoi

- In other words, once we know how to move $n-1$ disks, moving n disks is easy
- But how do we know how to move $n-1$ disks?
- Well, to move $n-1$ disk, we just need to know how to move $n-2$ disks!
- Wait how about $n-2$? We don't know how to do that right?
- Well, we just have to figure out how to move $n-3$ disks!
- $\ldots$
- But how to move 2 disks?
- Well, to move 2 disks, we just need to know how to move 1 disk
- **But moving 1 disk is trivial!**

Example: Tower of Hanoi

- Now we know how to solve the problem by recursion

```python
# Solving Tower of Hanoi using recursion

def Hanoi(n,rod_from,rod_to):
  if n == 1:
    print(f'Move the disk from rod {rod_from} to rod {rod_to}')
  else:
    rod_middle = 6 - rod_from - rod_to
    Hanoi(n-1,rod_from,rod_middle,lv+1)
    print(f'Move the disk from rod {rod_from} to rod {rod_to}')
    Hanoi(n-1,rod_middle,rod_to,lv+1)
```

Example: Listing all permutations

- Recall the permutations of a set is a set of all possible rearrangements of its elements
- For example, the permutations of `{A,B,C}` are:
  `{A,B,C}`,`{A,C,B}`,`{B,A,C}`,`{B,C,A}`,`{C,A,B}`,`{C,B,A}`
- As mentioned before, the total number of permutations of a n-element set is $n!$
- Now we wish to write an algorithm that generates all the permutations

Example: Listing all permutations

- Let's see how we can approach the problem
- First, note that the permutation of 1-element set is trivial, it's just the element itself
- Second, to generate the permutations for $n + 1$ elements, we just need to make each element appear once in front, and generate permutations for the remaining n elements
- For example, Permutations of $\{1, 2, 3, 4\}$:

$$\{1, \text{Permute } 2, 3, 4\}$$
$$\{2, \text{Permute } 1, 3, 4\}$$
$$\{3, \text{Permute } 1, 2, 4\}$$
$$\{4, \text{Permute } , 1, 2, 3\}$$

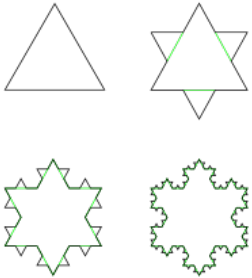
Example: Listing all permutations

- This allow us to write down a very simple recursive algorithm that does the job

```python
def swap(l,i,j): # Swap element i,j in list l
  temp = l[i]
  l[i] = l[j]
  l[j] = temp

def Permutation(start,end,arr): # In-place permutation
  if start == end:
      print(arr)
  else:
      for i in range(start,end):
          swap(arr,start,i)
          Permutation(start+1,end,arr)
          swap(arr,i,start)
```

Example: Koch Snowflake

- This example is unrelated to problem solving, but it introduce some interesting mathematics of fractals
- Koch snowflake is one of the earliest fractals to have been described



Example: Koch Snowflake

- The snowflake can be constructed using the following procedure

1. Draw an equilateral triangle with sides *s*; For each line segment:
2. Divide the line segment into three segments of equal length
3. Draw an equilateral triangle that has the middle segment from step 3 as its base and points outward
4. Remove the line segment that is the base of the triangle from step 3
5. Repeat at step 2

Example: Koch Snowflake

- The resulting animation is shown here
- Surprisingly, this figure has infinite perimeter but finite area

Theorem (Perimeter of Koch Snowflake)

Let s be the side length of the original equilateral triangle, the perimeter of the figure at the $n+1$th iteration is given as:

$$P_n = 3 \cdot s \cdot \left(\frac{4}{3}\right)^n$$

In other words, as $n \to \infty$, $P_n \to \infty$

Proof.

Observe in each iteration, we remove $1/3$ (the base of the triangle) of the perimeter and add in $2/3$ of the perimeter (the other two sides of the triangle). Hence, in each iteration the perimeter is scaled up by a factor of $4/3$. Since the perimeter of the 1st iteration (i.e. $n = 0$) is $3s$, it follows that for the $n + 1$th iteration:

$$P_n = (3s) \cdot \left(\frac{4}{3}\right)^n$$

This completes the proof $\square$

Example: Koch Snowflake

Theorem (Area of Koch Snowflake)

Let a_0 be the area of the original triangle, at $n+1$th iteration, the area of the triangle A_n is given as:

$$A_n = \frac{a_0}{5}\left[8 - 3\left(\frac{4}{9}\right)^n\right]$$

In particular when $n \to \infty$, $A_n \to \frac{8}{5}a_0$

Proof.

Let s_n be the number of sides in each iteration. Then:

$$s_{n+1} = 3s_n - s_n + 2s_n = 4s_n$$

Hence $s_n = 3 \cdot 4^n$ Now in the $n+1$th iteration we add s_n triangle to the figure, each with sides $(1/3)^{n+1}$, so the new area is:

$$\begin{aligned} A_{n+1} &= A_n + a_0 s_n \left(1/3\right)^{2(n+1)} \\ &= A_n + \frac{a_0}{3}\left(\frac{4}{9}\right)^n \end{aligned}$$

Since $A_0 = a_0$, expanding the recursion we have:

$$A_n = a_0\left[1 + \frac{1}{3} + \frac{1}{3}\left(\frac{4}{9}\right) + \frac{1}{3}\left(\frac{4}{9}\right)^2 + \cdots + \frac{1}{3}\left(\frac{4}{9}\right)^{n-1}\right]$$

Proof.

Recall the geometric sequence sum formula (Try to prove it!):

$$a_0 + a_0 r + a_0 r^2 + \cdots a_0 r^{n-1} = a_0 \left(\frac{1 - r^n}{1 - r} \right)$$

Hence the sum becomes:

$$\begin{aligned} A_n &= a_0 \left[1 + \frac{1}{3} \left(\frac{1 - (4/9)^n}{1 - (4/9)} \right) \right] \\ &= a_0 \left[1 + \frac{3}{5} \left(1 - \left(\frac{4}{9} \right)^n \right) \right] \\ &= \frac{a_0}{5} \left[8 - 3 \left(\frac{4}{9} \right)^n \right] \end{aligned}$$

This completes the proof $\square$

Example: Koch Snowflake

- We can define the dimension of an object by breaking it down in to *N* self-similar piece, each scaled down by a factor ϵ. For example, the dimension *D* of a square is 2 because we can break it down into 4 similar pieces each scaled by a factor of $1/2$.
- In general, we may define the diemnsion of a geometric object by:

$$N = \epsilon^{-D}$$

- Using this scaling law we can find that the dimension of Koch Snowflake to be: $\log 4/\log 3 \approx 1.262$