STMC coding team Training

Lesson 5: Function and Recursion

Tsai Yun Chen

April 5, 2024

Goal today

Function provides us a way to write code in a clean and organized way, and it provides us ways to solve problems with few lines of code

- Introduce function to modularize code
- Using function to make previous code cleaner
- Function as parameter, Function Folding
- Introduce recursion: Divide and Conquer
- Merge Sort
- Fibonacci number
- Problem of Recursion
- Strategy of Memoization
- Fibonacci number, revised

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

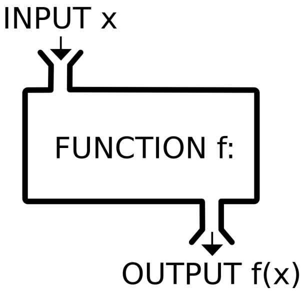
- As shown in the code, line 13 to 19 is completely identical to the above
- It would be great if we can put them into one line like this:

```
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,

$$\max(a, b) = \frac{|a-b| + (a+b)}{2}$$

gives the larger number between a and b. (Why?)

- By nesting max we can perfrom find max between more number

$$f(a, b, c, d) = \max(\max(a, b), \max(c, d))$$

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

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

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

```python
def max(a,b):
  return (abs(a-b)+(a+b))/2   #you can also use if-then-else
```

Function Exercises

Complete the following exercises

1. Write a function `Square(x)` that square a number
2. Write a function `SumAll(num_list)` that return the sum of a list of numbers `num_list`
3. Write a function `MinMax(num_list)` that returns the mininum number from the list `num_list`
4. Write a function `Reverse(str)` that returns the reverse of the string

Folding

- Observe in the last few tasks, we are repeating something similar as well!
- We walk through the list and do something to each elements
- Can we make this as a function as well?
- Yes! This trick is called function folding

Function Left Folding

```
def LeftFold(list,func):
  first=False
  for item in list:
    if first:
      res=item
      first=True
    else:
      res=func(res,item)
  return res
```

Example of using folding

```
def max(a,b):
  return (abs(a-b)+(a+b))/2

def sum(a,b):
  return a+b

list=[3,7,11,99,34,5,16]

print("The maximum of elements of the list:",LeftFold(list,max))
print("The sum of elements of the list:",LeftFold(list,sum))
```

Recursion: Divide and Conquer

- Some times problems can be divided into several parts
- Each parts are actually the same problem, but smaller
- example sorting, permutation, counting etc.
- Recursion is a kind of trick that solve the problem by first solving its smaller sub-problem

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

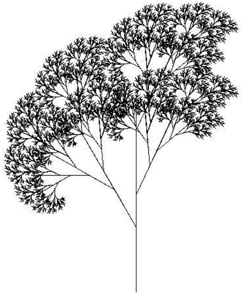


Figure 1: Recursive Tree (Source)

Example: Merge Sort

- Recall we discussed about how we can sort a list using looping
- We also discussed about the time needed for the naive soring: $O(n^2)$.
- Now we discuss a recursive approach

Example: Merge Sort

- The first step is to find the sub-problem
- Let's say we separate our array into two halves
- Then we sort them separately
- Can we merge the two sorted arrays?

Example: Merge Sort

- Let's try, consider having an array

$$\{11, 6, 4, 2, 7, 18, 1, 21\}$$

- Now separate it into two halves,

$$\{11, 6, 4, 2\}, \{7, 18, 1, 21\}$$

- sort them separately,

$$\{2, 4, 6, 11\}, \{1, 7, 18, 21\}$$

- How do we merge them? Take the smaller of them! (Greedy)

Example: Merge Sort

- But wait! How do we sort the two smaller array?
- Look back to our original problem, what are we solving? Sorting!
- Let's run the same function on the smaller array again.
- But when do we stop?
- If an array contains only one element, then it must be sorted.

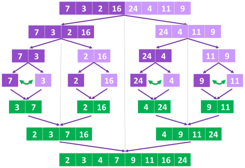


Figure 2: Merge Sort (Source)

```python
def MergeSort(list):
  length=len(list)
  if length==1:
    return list
  left=list[0:length//2]
  right=list[(length//2):length]
  LeftSorted=MergeSort(left)
  RightSorted=MergeSort(right)
  listSorted=[]
  LPointer=0
  RPointer=0
  for i in range(length):
    if RPointer==len(RightSorted):
      listSorted.append(LeftSorted[LPointer])
      LPointer=LPointer+1
    elif LPointer==len(LeftSorted) or RightSorted[RPointer]<LeftSorted[LPointer]:
      listSorted.append(RightSorted[RPointer])
      RPointer=RPointer+1
    else:
      listSorted.append(LeftSorted[LPointer])
      LPointer=LPointer+1
  return listSorted
```

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

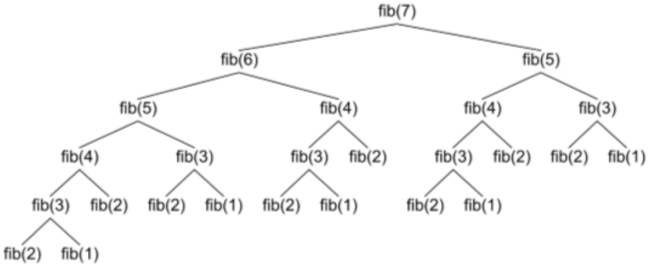
```python
# Fibonacci sequence using recursion
def Fib(n):
  if n == 1 or n == 2:
    return 1
  else:
    return Fib(n-1) + Fib(n-2)

```

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

Something is wrong…

- Look at the tree diagram on previous page, what can you observed?
- How many time does fib(5) appear? What about fib(4) and fib(3)?
- A problem of recursion is it might solve the same problem repeatly
- You might ask, why care about it?

Time complexity of naive Fibonacci recursion

- Let's see if that really matters.
- We let $T(n)$ be the time needed for solving the n-th fibonacci number
- By the formula, we know that

$$T(n) = \underbrace{T(n-1) + T(n-2)}_{\text{computing previous terms}} + \underbrace{1}_{\text{addition}}$$

- Solving it (with some tedious math), we get

$$T(n) \approx 2^n$$

Strategy of Memoization

- Exponential is almost the worst thing we can get.
- Turns out it matters, but how can we improve it?
- Simple method: remembering previous result
- This approach is called Memoization

Fibonacci number, revised

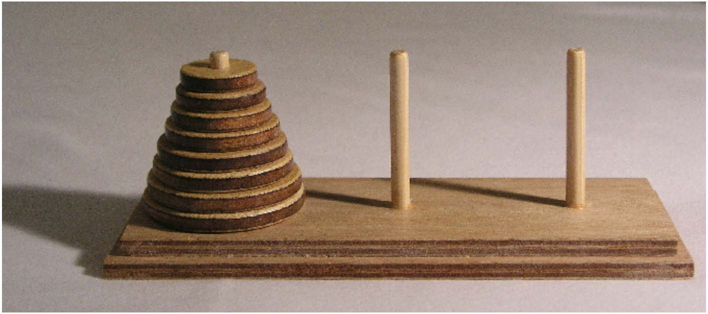
```
# Fibonacci sequence using recursion, with Memoization
def _Fib(n):
  if n == 1 or n == 2:
    return (1,1)
  else:
    prev=_Fib(n-1)
    return (prev[1],prev[0]+prev[1])

def Fib(n):
  pair=_Fib(n)
  return pair[1]
```

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.

Exercise: Tower of Hanoi



Exercise: 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

Solution: 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

Solution: 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!**

Homework

- Homework 3 is posted on the course website, namely the HW3.ipynb
- same as last time, 3 problems, sorted in ascending order of difficulty
- cover topics of looping, list and functions
- submit the homework to the same place, inside the folder of `HW3 submission`
- remember to include all your group member's name in the document
- deadline: before next lesson, i.e. 13/4
- the solution will be disclosed one week after the deadline, i.e. 20/4
- comments and solution on HW2 are released.