COMP10001 Foundations of Computing
Recursion

Semester 1, 2019

Tim Baldwin, Nic Geard, Farah Khan, and Marion Zalk

THE UNIVERSITY OF MELBOURNE

— version: 1449, date: March 1, 2019 —

Reminders

- Grok Worksheets 13 & 14 due 11:59pm Monday 13 May
- Project 2 due 11:59pm Thursday 9 May
- Project 3 opens this Thursday 9 May

Lecture Agenda

- Last lecture:
  - Iteration
- This lecture:
  - `itertools`
  - Recursion

Iterators vs. Sequences

Iterators

- no random access
- "remembers" last item seen
- no `len()`
- can be infinite
- traverse exactly once (forwards)

Sequences

- supports random access
- doesn't track last item
- has `len()`
- must be finite
- "traverse" multiple times (fwd/rev/mix)

The `itertools` Module

- Implements a number of iterator "building blocks"
- Inspired by other programming languages (APL, Haskell, SML)
- Standardises a set of fast, memory efficient tools
- Each tool can be used alone or in combination
- Forms an "iterator algebra"

`product`: Cross-product of Sequences

```
from itertools import product
def get_deck():
    """Create a list of 52 cards."""
    suits = 'CDHS'
    values = '234567890JQKA'
    deck = product(values, suits)
    return ([''.join(c) for c in deck])
```

cycle: Repeating Items Indefinitely

```
from itertools import cycle
def deal():
    """Put cards in 4 equal piles."""
    deck = get_deck()
    hands = [[], [], [], []]
    players = cycle(hands)
    for card in deck:
        player = next(players)
        player.append(card)
    return(hands)
print(deal()[0])
print(deal()[1])
print(deal()[2])
print(deal()[3])
```

Example Uses

```
>>> deck = get_deck()
>>> len(deck)
52
>>> deck[:7]
['3C', '3S', '3D', '3H', '4C', '4S', '4D']
>>> hands = deal()
>>> hand1 = sorted(hands[0])
>>> hand1
['2C', '3H', '3S', '4C', '4S', '5S', '6D',
 '7D', '8D', 'AC', 'AD', 'AH', 'AS']
```

groupby: Group Items by Some Criterion

```
>>> from itertools import groupby
>>> def first(x): return(x[0])
>>> for rank, group in groupby(hand, first):
...    print("{} {}".format(rank, list(group)))
2 ['2C']
3 ['3H', '3S']
4 ['4C', '4S']
5 ['5S']
6 ['6D']
7 ['7D']
8 ['8D']
A ['AC','AD','AH', 'AS']
```

combinations: *n* Choose *k*

```
>>> from itertools import combinations
>>> aces = ['AC','AD','AH', 'AS']
>>> combinations(aces,2)
<itertools.combinations object at 0x1794998>
>>> list(combinations(aces,2))
[('AC', 'AD'), ('AC', 'AH'), ('AC', 'AS'),
 ('AD', 'AH'), ('AD', 'AS'), ('AH', 'AS')]
```

A Recursive Mindset

- Imagine there was no iteration
  - No `for` or `while` loops (or `iter` and `next`)
  - No list comprehensions
  - No builtins like `min`, `sum`, `len`
- Count the area of The Great Barrier Reef that is bleached. You have data as a list of True and False for a 1 km^2 grid of The Reef.

```
data = [True, False, False, True, ...]

def count(lst):
  '''Return the number of True's in a list.'''
```

No loops...

```
def count(lst):
  '''Return the number of True's in a list.'''
  n = len(lst)
  if n==0: return 0
  if n==1: return lst[0]
  if n==2: return lst[0] + lst[1]
  if n==3: return lst[0] + lst[1] + lst[2]
  if n==4: return lst[0] + lst[1] + lst[2] + lst[3]
  ...
```

- Q: when you see repeated code, what do you do?
- A: ???

No loops...

```python
def count(lst):
  '''Return the number of True's in a list.'''
  n = len(lst)
  if n==0: return 0
  if n==1: return lst[0]
  if n==2: return lst[0] + lst[1]
  if n==3: return lst[0] + lst[1] + lst[2]
  if n==4: return lst[0] + lst[1] + lst[2] + lst[3]
  ...
```

- Q: when you see repeated code, what do you do?
- A: make a function ("modularise" or "factorise")

Modularise

```python
def count(lst):
    '''Return the number of True's in a list.'''
    n = len(lst)
    if n == 0: return 0
    return lst[0] + count_list(lst[1:])

def count_list(lst):
    '''Return the number of True's in a list.'''
```

- Hold on, `count_list` looks very familar...
- We already have a function that does this: `count`
- Can a function call itself?

Modularise

```python
def count(lst):
    '''Return the number of True's in a list.'''
    if len(lst) == 0:
        return 0

    return lst[0] + count(lst[1:])
```

- Python cannot tell the difference between a function calling itself, and calling some other function.
- But, because it is often a useful way of thinking about problems in computing/maths it has a special name: recursion.

A Recursive Mindset II

- How can we break the problem into an instance of the same problem, but on a smaller input?
- What happens on the smallest case (the "base case")?
- `len(lst) = 1 + len(lst[1:])`
- `my_max(lst) = max(lst[0], my_max(lst[1:]))`
- `my_min(lst) = min(lst[0], my_min(lst[1:]))`

Class Exercise

- Write a function to sum all elements in a list without using iteration.
- Hint: think recursively. How can you break down the problem of adding up n elements in a list into one of adding up one element and $n-1$ elements?

The Elements of Recursion

- "Recursive" function definitions are often use to solve problems in a "divide-and-conquer" manner, breaking the problem down into smaller sub-problems and solving them in the same way as the big problem
- They are generally made up of two parts:
  - recursive function call(s) on smaller inputs
  - a (reachable) base case to ensure the calculation halts
- Recursion is closely related to "mathematical induction"

Class Exercise

- Write a function to compute $n!$ without using iteration.
- Hint: think recursively. How can you compute $n!$ based on $(n-1)!$? What is the base case?

But why?

- Defining answers recursively (in terms of instances of the same problem on a smaller input) is common in maths
- Simple to translate to Python

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{, if } n > 2 \\ 1 & \text{, otherwise} \end{cases}$$

$$Q(n) = \begin{cases} Q(n - Q(n-1)) + Q(n - Q(n-2)) & , n > 2 \\ 1 & , n \leq 2 \end{cases}$$

But why? II

- Cast your mind back to Lecture 3a, second last slide...
  - Assuming an unlimited number of coins of each of the following denominations:

    $$(1, 2, 5, 10, 20)$$

    calculate the number of distinct coin combinations which make up a given amount N (in cents).
- We answered this with 5 nested for loops

Coins I

```python
'''Count the number of combinations of
   (1,2,5,10,20) that sum to N
'''
answer = 0
for a in range(N+1):
  for b in range(N//2+1):
    for c in range(N//5+1):
      for d in range(N//10+1):
        for e in range(N//20+1):
          if a+2*b+5*c+10*d+20*e == N:
            answer += 1
```

An iterative solution. But what if there were 6 denominations, or 7, or 8, or k?

Coins

- Think recursively. How many ways can we put in the first coin, and then work out all the combinations for the rest.

`answer(N, (1,2,5,10,20) )`
Put in zero 1's, then need `answer(N, (2,5,10,20) )`
Put in one 1, then need `answer(N-1, (2,5,10,20) )`
Put in two 1's, then need `answer(N-2, (2,5,10,20) )`
Put in three 1's, then need `answer(N-3, (2,5,10,20) )`
...

```
answer(N, coins) = sum answer(N-i*coins[0], coins[1:])
                   for i in 0,1,2,...N//coins[0]
```

Coins

```
answer(N, coins) = sum answer(N-i*coins[0], coins[1:])
                   for i in 0,1,2,...N//coins[0]
```

What's the base case?

```
    answer(N, single_coin) =
```

How many ways can you make up N with only one coin denomination?

Coins III

```
def answer(N, coins):
    if len(coins) == 1:
        if N % coins[0] == 0:
            return 1
        else:
            return 0

    c = coins[0]
    count = 0
    for i in range(0, N//c+1):
        count += answer(N-i*c, coins[1:])

    return(count)
```

The problem is difficult with iteration.

The Powerset Problem

Given a set, S, compute the powerset $\mathcal{P}(S)$ of that set (a set of all subsets, including $\{\}$).
Think recursively: construct the powerset of $n-1$ items, and add first item to each of them.

```
def power_set(lst): # lists easier than sets
    if lst == []:
        return [[]]
    rest = power_set(lst[1:])
    result = []
    for item in rest:
        result.append(item)
        result.append([lst[0]] + item)
    return result
```

`index` - Linear Search

- Input: sorted `list` of numbers
- Output: the index of a given number `x`, or `None` if it's not in the list
- Thinking recursively:

$$index(x, lst) = \begin{cases} None & \text{if lst is empty} \\ 0 & \text{if lst[0] == x} \\ 1 + index(x, lst[1:]) & \text{otherwise} \end{cases}$$

`index` - Binary Search

- Input: sorted `list` of numbers
- Output: the index of a given number `x`, or `None` if it's not in the list
- Thinking recursively and cleverly (`n=len(lst)`):

$$index(x, lst) = \begin{cases} None & \text{if lst is empty} \\ n/2 & \text{if lst[n/2] is x} \\ index(x, lst[:n/2]) & \text{if x < lst[n/2]} \\ n/2 + index(x, lst[n/2:]) & \text{otherwise} \end{cases}$$

0	1	2	3	4	5	6	7
1	3	10	12	15	45	86	91

Binary Search: Recursive Solution

```
def bsearch(val,nlist):
    return bs_rec(val,nlist,0,len(nlist)-1)

def bs_rec(val,nlist,start,end):
    if start > end:
        return None
    mid = start+(end-start)//2
    if nlist[mid] == val:
        return mid
    elif nlist[mid] < val:
        return bs_rec(val,nlist,mid+1,end)
    else:
        return bs_rec(val,nlist,start,mid-1)
```

Binary Search: Iterative Solution

... but again, there's an equally elegant iterative solution:

```
def bs_it(val,nlist):
    start = 0
    end = len(nlist) - 1
    while start < end:
        mid = start+(end-start)//2
        if nlist[mid] == val:
            return mid
        elif nlist[mid] < val:
            start = mid + 1
        else:
            end = mid - 1
    return None
```

So When *Should* You Use Recursion?

Recursion comes to its fore when an iterative solution would involve a level of iterative nesting proportionate to the size of the input, e.g.:

- the powerset problem: given a list of items, return the list of unique groupings of those items (each in the form of a list)
- the change problem: given a list of different currency denominations (e.g. `[5,10,20,50,100,200]`), calculate the number of distinct ways of forming a given amount of money from those denominations

Making Head and Tail of Recursion

- Recursion occurs in two basic forms:
  1. **head recursion:** recurse first, then perform some local calculation

```
def counter_head(n):
    if n < 0: return
    counter_head(n-1)
    print n
```

  2. **tail recursion:** perform some local calculation, then recurse

```
def counter_tail(n):
    if n < 0: return
    print n
    counter_tail(n-1)
```

Recursion: A Final Word

- Recursion is very powerful, and should always be used with caution:
  - function calls are expensive, meaning deep recursion comes at a price
  - always make sure to catch the base case, and avoid infinite recursion!
  - there is often a more efficient iterative solution to the problem, although there may not be a general iterative solution (esp. in cases where the obvious solution involves arbitrary levels of nested iteration)
  - recursion is elegant, but elegance $\neq$ more readable or efficient

Lecture Summary

- What is recursion? What two parts make up a recursive function?
- What is the difference between head and tail recursion?
- What is binary search, and how does it work?
- In what cases is recursion particularly effective?
- Why should recursion be used with caution?