Lecture 6: Recursion and Dictionary

Recursion
- is the process of repeating items in a self-similar way.

What is Recursion then?
- Algorithmically: a way to design solutions to problems by dividing and conquering
  - reduce a problem to similar version of the same problem.
- Semantically: a programming technique where a function calls itself.
  - in programming the goal is not to have infinite recursion
    - must have one or more base cases that are easy to solve
    - must solve the problem on some other input with the goal of simplifying the larger problem input

Iterative Algorithms so far
- looping constructs (while and for loops) lead to iterative algorithms
- can capture computation in a set of state variables that update on each iteration loop

Multiplacative - Iterative Solution

- multiply a * b is equivalent to "add a to itself b times"
- capture state by
  - an iteration number (i) starts at b
    - i ← i - 1 and stop when 0
  - a current value of computation (result)
    - result ← result + a

```
>>> def mult_iter(a,b):
        result=0
        while b > 0:        → iteration
            result += a     → iterate the running sum
            b -=1           → current value of iteration variable.
        return(result)
```

Note: ## Multiplaction-Recursion Solution

- recursion step:
  - think how to reduce problem to similar/small version of the same problem
- base case:
  - keep reducing problem until reach a similar case that can be solved directly
    - when b = 1, a * b = a

```
>>> def mult_recu(a, b):
        if b == 1:                          # base case
            return(a)
        else:
            return(a + mult_recu(a, b-1))   # recursive step
```

Note

a * b = a + a + a + a + a + ... + a

= a + [a + a + a + a + ... + a] ← b - 1

= a * (b - 1) → recursion reduction

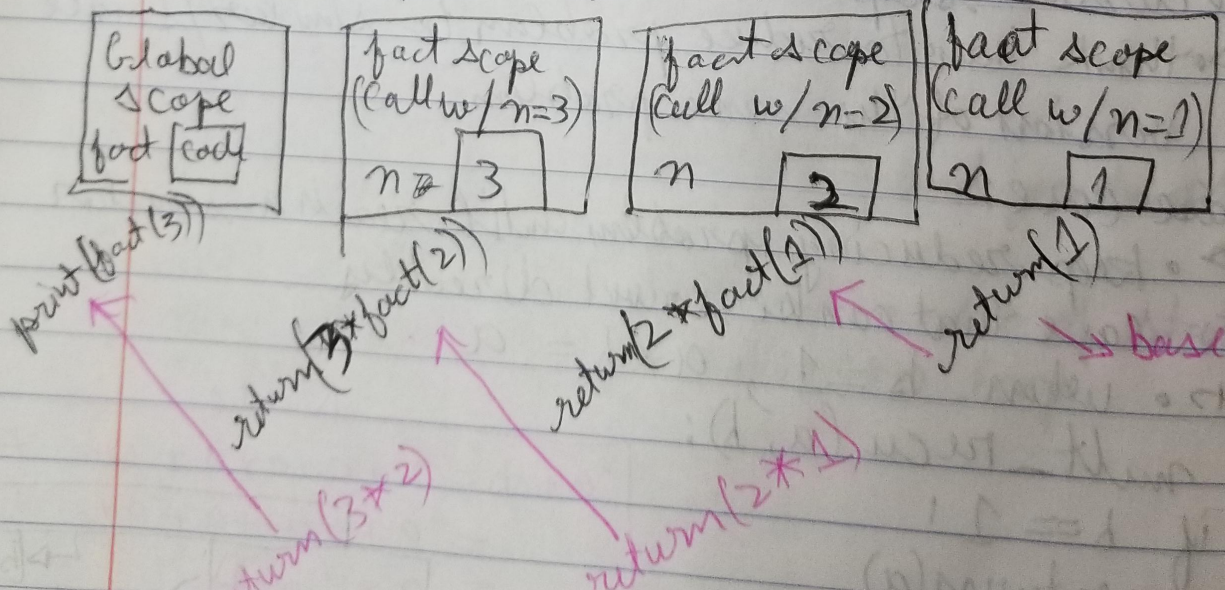
Factorial (Recursion)

$n! = n * (n-1) * (n-2) * \cdots 1$

- for what n do we know the factorial?
  - $n=1$ → if $n == 1$: return (1) — base case
- how to reduce problem?
  - rewrite in terms of something simpler to reach base case
  - $n * (n-1)!$ → else: — recursive step
    return (n * factorial (n-1))

```
def fact(n):
    if n == 1:
        return(1)
    else:
        return(n * fact(n-1))
```

- 2 x of the code::
  lets print(fact(3))



Some Observations

- each recursive call to a function creates its own scope/environment
- bindings of variables in a scope are not changed by recursive call
- flow of control passes back to previous scope once function call returns value

Mathematical Induction

- To prove a statement indexed on integers is true for all values of n:
  - Prove it is true when n is smallest value (eg: $n=0$ or $n=1$)
  - Then prove that if it is true for an arbitrary value of n, one can show that it must be true for $n+1$

Example of Induction:

- $0+1+2+3+\cdots+n = \frac{n(n+1)}{2}$

Proof: If $n=0$, then left hand side is 0 and right hand side is 0 because $\frac{0\cdot(0+1)}{2}$.

Assume true for some k, then we need to show that its true for $k+1$ case:

$1+2+3+\cdots k = \frac{1}{2}(k(k+1))$

$1+2+3+\cdots k+(k+1) = \frac{1}{2}k(k+1)+(k+1)$

$= \frac{1}{2}(k+1)(k+1+1) = \frac{1}{2}(k+1)(k+2)$ □

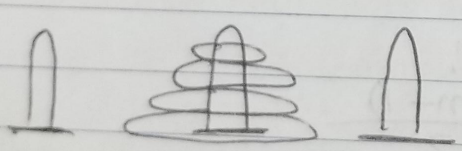
What is the Relevance to Code?

- Same logic of Induction is applied

```
>>> def mult(a,b):
        if b == 1:
            return (a)
        else:
            return(a+mult(a,b-1))
```

- Base case, we can show that mult must return correct answer.
- For recursive case, we can assume that mult correctly returns an answer for problems of size smaller than b, then by the addition step, it must also return a correct answer for problem of size b.
- Thus by induction, code correctly returns answer

Tower of Hanoi:



goal move the rings from one tower to another without putting larger ring on top of smaller ring. (takes 15 steps)

We can do this using program and recursive technique.

→

```
>>> def printMove(fr, to)
        print('move from' + str(fr) + 'to' + str(to))
>>> def Tower(n, fr, to, spare):
        if n==1:
            printMove(fr, to)
        else:
            Tower(n-1, fr, spare, to)
            Tower(1, fr, to, spare)
            Tower(n-1, spare, to, fr)
```

Recursion with Multiple Base Cases

- Fibonacci numbers
  - Leonardo de Pisa (aka Fibonacci) modeled the following:
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have one month generation period
    - Assume rabbits never die, that female always produces one more pair every month from its second month
    - How many female rabbits are there at the end of one year?

So it takes the first pair one month to mature. After 1 month they have a pair of rabbits, they take 1 month to mature, and have a pair of their own, so on and so forth.

month	Female
0	1
1	1
2	2
3	3
4	5
5	8

So;→ After one month (0); 1 female
↳ After second month; still 1 female ~~none~~ now pregnant
↳ After third month; two females and one of them pregnant

In general;
females(n) = females(n-1) + females(n-2)
↳ • every female alive at month n-2 will produce one female in month n.
↳ • these can be added to those alive in month n-1 to get total alive in month n

Programming wise:
↳ • Base Case:
Females(0) = 1
Females(1) = 1
↳ • Recursive Case:
Females(n) = Females(n-1) + Females(n-2)

```
def fib(x):
    if x==0 or x==1:
        return(1)
    else:
        return(fib(x-1)+fib(x-2))
```

Recursion on Non-Numerics

- How to check if a string is a palindrome
  - Palindrome:
    - "Able was I, ere I saw Elba"
    - Reads the same backwards and forwards
    - "Are we not drawn onward, we few, drawn onward to new era?"

To solve this Recursively:

- First convert the string to just characters, by stripping out punctuation, and converting uppercase to lower case
- Then:
  - Base case: a string of 0 or 1 is a palindrome
  - Recursive case: if first character matches last character, this is a palindrome if middle section is a palindome.

Example

- 'Able was I, ere I saw Elba' → 'ablewasiereisawelba'
- isPalindome('ablewasiereisawelba')

  is same as:

  'a' == 'a' and

  is Palindome('blewasiereisawelb')

  ⋮

code →

```
>>> def isPalindome(s):
    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return(ans)
    def isPal(s):
        if len(s) <= 1:
            return(True)
        else:
            return(s[0] == s[-1] and isPal(s[1:-1]))
    return(isPal(toChars(s)))
```

recursive step

Dictionaries

A list	
0	ele1
1	ele2
2	ele3
3	ele4

A dictionary

key 1	val 1
key 2	val 2
key 3	val 3

→ to store pairs of data: {key : value}

```
my_dict = {}
```
empty dictionary

```
grades = {'Ana': 'B', 'John': 'A+', 'Denise': 'A', 'Katy': 'A'}
```

key

value

Dictionary Lookup

- similar to indexing into a list
- looks up the key
- returns the value associated with the key
- if key isn't found, get an error

```
>>> grades['John'] -> 'A+'
>>> grades['Bob'] -> error
```

my_dict

'Ana'	'B'
'John'	'A+'
'Denise'	'A'
'katy'	'A'

Dictionary Operations

```
grades = {'Ana': 'B', 'John': 'A+', 'Denise': 'A', 'katy': 'A'}
```

- add an entry
  ```
  grades['Bob'] = 'A-'
  ```
- test if key in dictionary
  ```
  'John' in grades -> True
  'Dan' in grades -> False
  ```
- delete entry
  ```
  del(grades['Ana'])
  ```
- get an iterable that acts like a tuple of all keys
  ```
  grades.keys() -> ['Ana', 'John', 'Denise', 'katy']
  ```
- get an iterable that acts like a tuple of all values
  ```
  grades.values() -> ['B', 'A+', 'A', 'A']
  ```

order not guaranteed

Dictionary: keys and Values.

- Values:
  - any type (immutable or mutable)
  - can be duplicates
  - dictionary values can be lists, even other dictionaries
- keys:
  - must be unique
  - immutable type (int, float, string, tuple, boolean)
    - actually need an object that is hashable, but think of as immutable as all immutable types are hashable.
  - Careful with float type as a key
- No Order to key or values

Ex:: d = {4:{1:0}, (1,3):'twelve', 'const':[3.14, 1.618]}

List	vs	Dictionary
• ordered sequence of elements		• matches "key" to "value"
• look up elements by an integer index		• look up one item by another item
• indices have an order		• no order is guaranteed
• index is an integer		• key can be any immutable type

Example: 3 functions to Analyze Song Lyrics

1) creat a frequency dictionary mapping str: int

```
>>> def lyrics_to_frequency(lyrics):
        my_dict = {}
        for word in lyrics:
            if word in my_dict:
                my_dict[word] += 1
            else:
                my_dict[word] = 1
        return (my_dict)
```

2) find word that occurs at least X times
   - let the user choose "at least X times" so allow as parameter
   - return a list of tuples, each tuple is a (list, int) containing the list of words ordered by their frequency
   - IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are ~~not~~ mutating the song dictionary

```
>>> def words_often(freqs, minTimes):
        result = []
        done = False
        while not done:
            temp = most_common_words(freqs)
            if temp[1] >= minTimes:
                result.append(temp)
```

```
            for w in temp[0]:
                del(freqs[w])
        else:
            done = True
    return(result)
print(words_often(beatles, 5))
```

3) find word that occurs the most and have many times
   - use a list, in case there is more than one word
   - return a tuple(list, int) for (words_list and highest_frequency)

```
>>> def most_common_words(freqs):
        values = freqs.values()
        best = max(values)
        words = []
        for k in freqs:
            if freqs[k] = best:
                words.append(k)
        return(words, best)
```