Recursion

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Recursion
  - Recursive data structures
  - Recursive functions
  - Recursive calls
- Application examples
  - loop replacement: tail/head recursion
  - Counting elements in nested lists

Recursive data structures

- One that contains substructures of the same type as itself
- Examples
  - nested lists: `[1, [2, 3], [4, [5, 6]]]`
  - nested tuples: `(1, (2, 3), (4, (5, 6)))`
  - dictionary of dictionaries:
    `{1: {2: 3}, 4: {5: 6, 7:8}}`
  - trees, graphs, expressions, statements, ...

Recursive Function

- a function that calls itself directly or indirectly
- Why call itself?
- handling recursive data structure
- not different from any other function

a function that calls itself

```
def recursive_func(n):
    print(f'hello {n}')
    recursive_func(n+1)
```

```
>>> recursive_func(0)
hello 0
hello 1
...
hello 994
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in recursive_func
  File "<stdin>", line 3, in recursive_func
  File "<stdin>", line 3, in recursive_func

  [Previous line repeated 992 more times]

  File "<stdin>", line 2, in recursive_func

RecursionError: maximum recursion depth
exceeded while calling a Python object
```

- Like infinite loop!
- calling is easy, but how does it return?

Structure of Recursive Functions

- base case:
  - return without recursive call
  - without base case => function might never return!
  - however, having base does not guarantee it will be executed, so need to be careful
- recursive case:
  - calls itself one or more times
  - may call itself directly or indirectly (i.e., calling another function that then calls it)

Example: factorial

- mathematically defined as

$$n! = \begin{cases} 1 & \text{if } n < 2 \quad \text{=> base case} \\ n \times (n-1)! & \text{if } n \geq 2 \quad \text{=> recursive case} \end{cases}$$

n	0	1	2	3	4	5	6	7
$n!$	1	1	2	6	24	120	720	1540

Factorial in Python

- Iterative versions
  - using **`for`** loop

```
def iter_fac(n):
    s = 1
    for i in range(2, n+1):
        s = s * i
    return s
```

  - using **`while`** loop

```
def iter_fac(n):
    s = 1
    while n >= 2:
        s *= n
        n -= 1
    return s
```

- Recursive version

```
def rec_fac(n):
    if n < 2:
        return 1
    else:
        return n * rec_fac(n-1)
```

looks like the equation!

$$n! = \begin{cases} 1 & \text{if } n < 2 \\ n \times (n-1)! & \text{if } n \geq 2 \end{cases}$$

```
def rec_fac(n):
    return 1           if n < 2 \
      else n * rec_fac(n-1)
```

Tracing recursive execution: stack

```
def f(n):
    if n < 2:
        return 1
    else:
        return n * f(n-1)
```

stack trace for f(4)

<table>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>f(1)</td><td>return 1</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>f(2)</td><td colspan="2">2*_</td><td>return 2</td><td></td><td></td></tr>
<tr><td></td><td>f(3)</td><td colspan="3">3*_</td><td>return 6</td><td></td></tr>
<tr><td>f(4)</td><td colspan="5">4*_</td><td>return 24</td></tr>
</table>

Example: Fibonacci

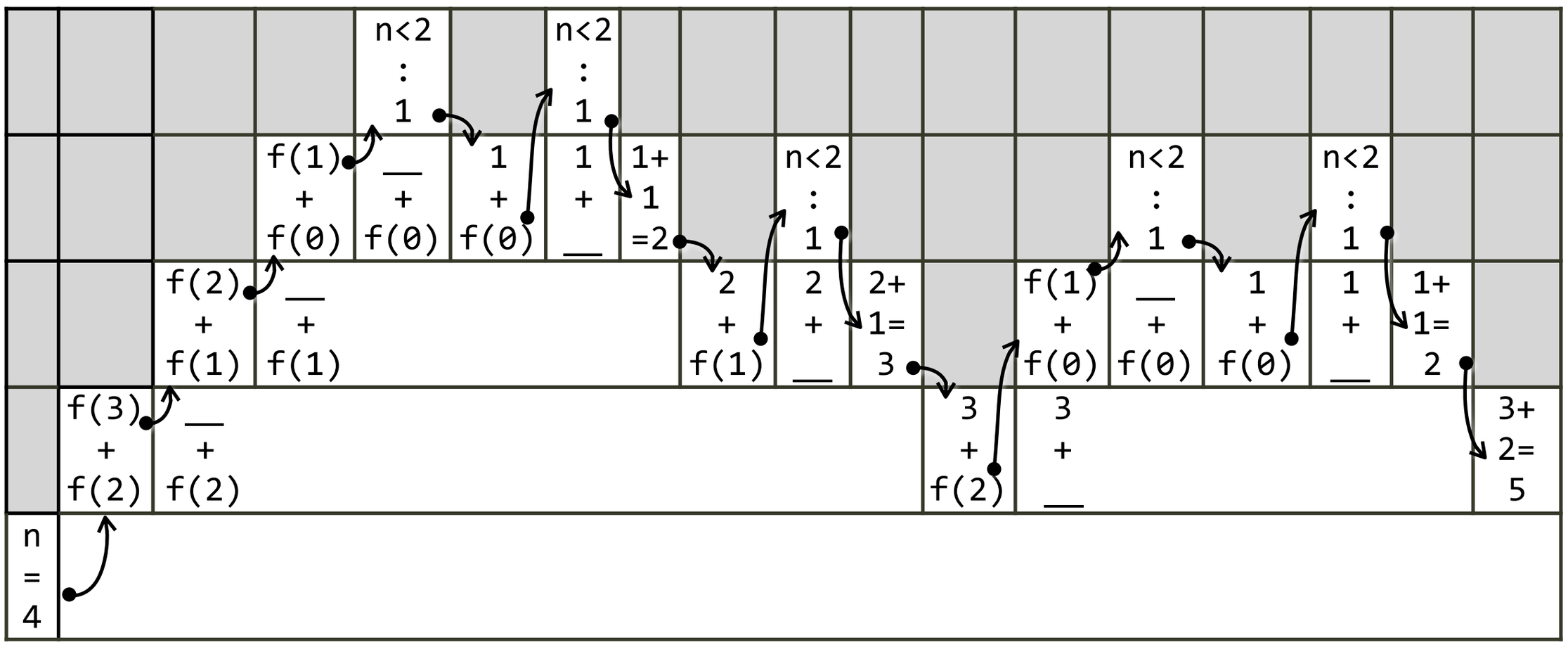
- mathematically defined as

$$fib(n) = \begin{cases} 1 & \text{if } n < 2 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9
fib(n)	1	1	2	3	5	8	13	21	34	55

fib() as recursive python code

```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



Another example: Counting elements in a list

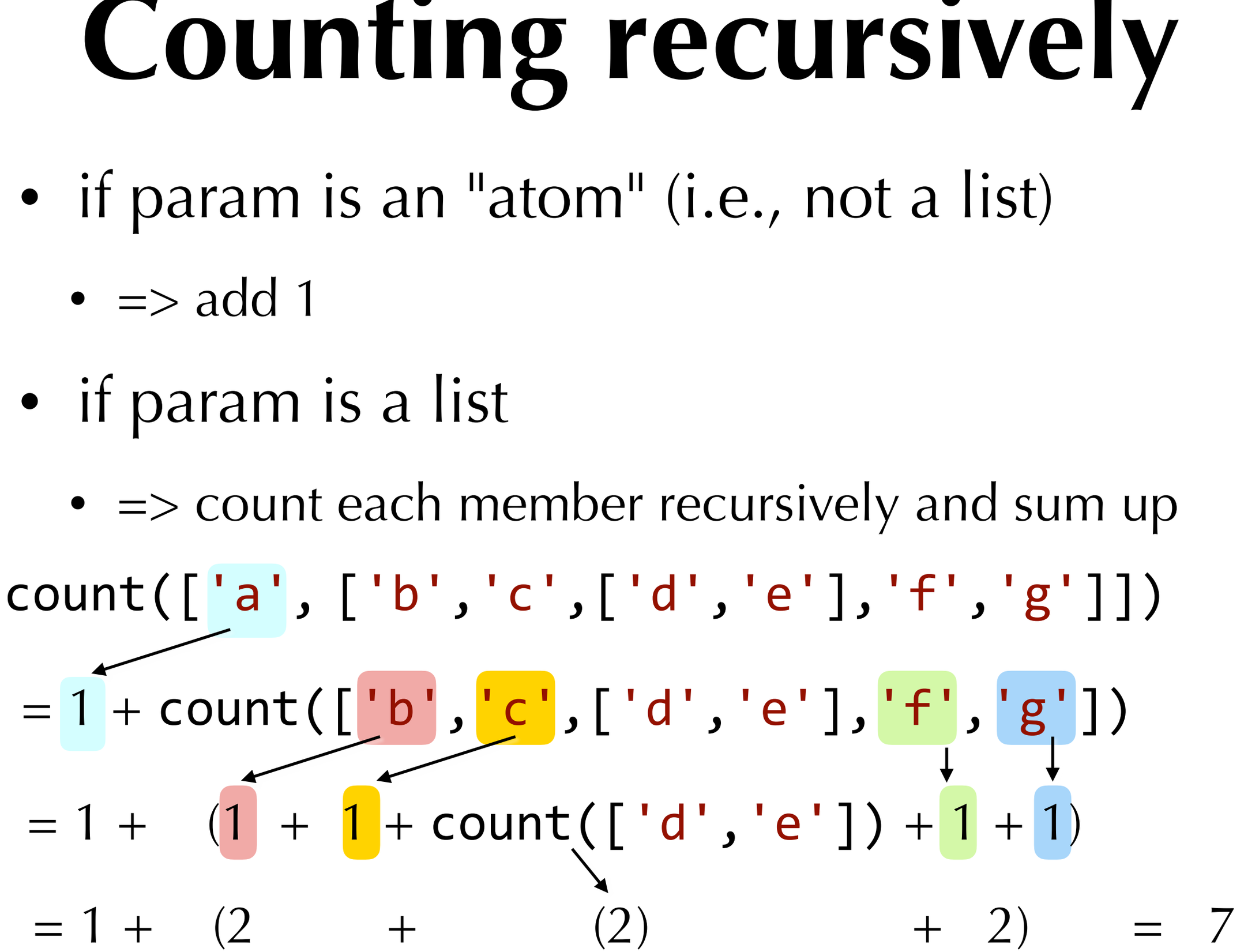
- Example:
  - `L = ['a',['b','c', ['d','e'],'f','g']]`
  - Q: How many elements are in `L`?

```
>>> L = ['a', ['b', 'c', ['d', 'e'], 'f', 'g']]
>>> len(L)
2
```

    - first item is string `'a'`,
    - second item is list `['b', 'c', ['d', 'e'], 'f', 'g']`
- But there are seven strings!!
  - Solution: Counting recursively

Counting recursively

- if param is an "atom" (i.e., not a list)
  - => add 1
- if param is a list
  - => count each member recursively and sum up



Source code for count()

- Base case
  - non-list type count as 1 item, no recursion

```
def count(x):
    if type(x) != list:
        return 1
    c = 0
    for i in x:
        c = c + count(i)
    return c
```

- Recursive case
  - each item could be a nested list, so call each recursively by calling count on it

Rewrite count() with sum of map

- **for**-loop version

```
def count(x):
    if type(x) != list:
        return 1
    c = 0
    for i in x:
        c = c + count(i)
    return c
```

- sum of map

```
def count(x):
    return 1 if type(x) != list
      else sum(map(count, x))
```

- Recursive case is sum(count(x[0]), count(x[1]), count(x[2]), ...)
- very concise!!

Alternative way to count

- if x is a list
  - if empty: return 0
  - else: recursively count(x[0]) + count(x[1:])
- otherwise, return 1
  - count([1, [2, 3, [4, 5], 6, 7]])
    = count(1) + count([[2, 3, [4, 5], 6, 7]])
    = 1 + (count(2) + count([3, [4, 5], 6, 7]))
    = 1 + (1 + (count(3) + count([[4, 5], 6, 7])))
    = 1 + (1 + (1 + (count([4, 5]) + count([6, 7]))))
    = 1 + (1 + (1 + ((count(4) + count([5])) + (count(6) + count([7])))))
    = 1 + (1 + (1 + ((1 + (1 + count([])))+ (1 + (1 + count([])))))
    = *7*

Alternative r_count: replace loop with recursion

```
def r_count(x):
    if type(x) != type([]):
        return 1
    elif len(x) == 0:
        return 0
    else:
        return r_count(x[0]) + r_count(x[1:])
```

- Two base cases
  - atom (non-list): return 1
  - empty container: return 0
- Recursive case
  - recursively r_count(x[0]) + recursively r_count(x[1:])

Other recursive structure: file system

- Directories are containers (lists)
- File are "atoms"
- Can use os package to access files

routine	purpose
`os.getcwd()`	get current working directory path (string)
`os.listdir(d)`	get list of names of files and directories in directory d
`os.path.isdir(d)`	check if d (string) is a directory

Example use of os routines

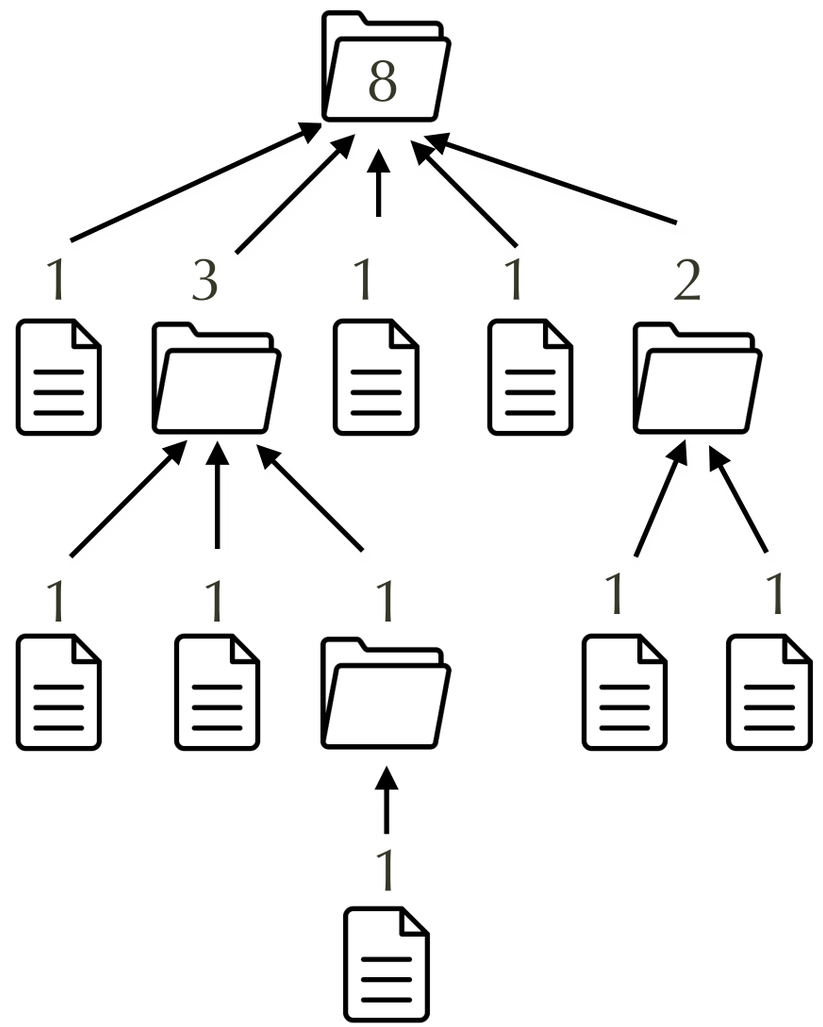
routine	purpose
`os.getcwd()`	get current working directory path (string)
`os.listdir(d)`	get list of names of files and directories in directory d
`os.path.isdir(d)`	check if d (string) is a directory

```
>>> import os
>>> cwd = os.getcwd()
>>> os.listdir(cwd)
['hello.py', 'cs1355f08-04.mp3', 'mult.py', 'out.txt', 'leap.py',
'posnegzero.py', 'result', 'myuniqOpt.py', 'prog.py',
'primeGen.py', 'index.py', ...]
>>> os.path.isdir(cwd)
True
>>> os.path.isdir(os.path.join(cwd, 'prog.py'))
False
```

How to count files recursively

- parameter: path
  - if none provided, use current directory `'.'`
- Base case:
  - if p is a file (not a directory), return 1
- Recursive case:
  - sum up the recursive count of each element in the directory

```
def count_files(p='.'):
    import os
    return 1 if not os.path.isdir(p)
      else sum(map(count_files, \
                   os.listdir(p)))
```



Example: recursive finding member

- Suppose we want a way to find the position of some data recursively

```
>>> L = [1, 2, [3, 4], 5]
>>> L.index(3)
ValueError: 3 is not in list
>>> rec_find(L, 3)
(2, 0)       # this means 3 can be found at L[2][0]
>>> M = [1, 2, [3, [4, 23], 5, 6]]
>>> rec_find(M, 23)
(2, 1, 1)    # 23 is found at M[2][1][1]
>>> rec_find(23, 23)
True
>>> rec_find(23, 45)
False
```

  - if first arg is not list/tuple, return value-to-value match

Source code for rec_find()

```
def rec_find(L, val):
   if type(L) in {list, tuple}: # if look inside members of L
        for i, v in enumerate(L):
            p = rec_find(v, val) # recursively find each member
            if p == True:        # L[i] == val, so we return (i,)
                return (i,)
            if p != False:       # L[i] recursively found val,
                return (i,)+p    # so we prepend i to its path p
   return L == val    # either L is not seq or for-loop didn't find
```

- base case:
  - L is not sequence, or val not found in L
- recursive case:
  - L is sequence, go over each member, return on first match

Example: indent_list()

- Want to print a list hierarchically
  - base case: string => print indent before string
  - recursive case: => increase indentation level by 1

```
>>> L = ['F1', ['F4', 'F5', ['F8']], 'F2', 'F3', 'D3', ['F6', 'F7']]
>>> indent_list(L)
    F1
        F4
        F5
            F8
    F2
    F3
    D3
        F6
        F7
```

Source code for indent_list()

```
def indent_list(L, level=0):
    if L == None:
        return
    if type(L) in {list, tuple}:
        for child in L:
            indent_list(child, level+1)
    else:
        print(f'{" "*4*level}{L}')
if __name__ == '__main__':
    L = ['F1', ['F4', 'F5', ['F8']], 'F2', 'F3', 'D3', ['F6', 'F7']]
    indent_list(L)
```

```
    F1
        F4
        F5
            F8
    F2
    F3
    D3
        F6
        F7
```

Example: Outline numbering from nested list of headings

sample input data

```
L=['Intro',
      ['Motivation', 'Contributions'],
   'Related Work',
      ['By Author', 'By Subject'],
   'Technical Approach',
      ['Overview',
         ['Block Diagram', 'Schematic'],
       'Algorithm',
         ['Static', 'Dynamic'] ],
   'Conclusions']
```

output

```
>>> number_outline(L)
1. Introduction
    1.1. Motivation
    1.2. Contributions
2. Related Work
    2.1. By Author
    2.2. By Subject
3. Technical Approach
    3.1. Overview
        3.1.1. Block Diagram
        3.1.2. Schematic
    3.2. Algorithm
        3.2.1. Static
        3.2.2. Dynamic
4. Conclusions
```

Technical Approach to Outline numbering

- Param: outline and a "prefix" (tuple)
  - L can be a list or a string `def number_outline(L, prefix=()):`
  - prefix is tuple. e.g., (1,) for "Introduction"
- Base case: L is a string
  - convert tuple into indentation and tiered number
  - e.g., L="Motivation", prefix=(1, 1)
    want 4 spaces indent, "1.1. ", then "Motivation"
  - indent is ' ' * 4 * (len(prefix)-1)

Technical Approach to Outline numbering (cont'd)

- Recursive case: L is a list/tuple
  - Iterate over L members, start a new tier of number
  - Initially, prefix=(), start new tier i = 0 =>
  - e.g., L=['Introduction', ['Motivation',...]..

```
        prefix=()
for v in L:
   if v is 'Introduction' => recurse(v, prefix=(1, ))
   if v is ['Motivation, ...] => recurse(v, prefix=(1,))
       and recursive call will start new tier
   if v is 'Related work' => recurse(v,prefix=(2,))
=> this pre-increment i if v is string;
      don't increment if v is a list/tuple.
```

Source code for Outline numbering

```
def number_outline(L, prefix=()):
    if type(L) in {list, tuple}:
        # keep prefix[-1], extend by new dimension, starting from 1
        i = 0
        for v in L:
            if type(v) not in {list, tuple}:
                i += 1
            number_outline(v, prefix+(i,))
            # don't increment if v is a list/tuple
# otherwise, indent and join the prefix together by '.'
    else:
        s = ' ' * 4*(len(prefix)-1)
        s += '.'.join(map(str,prefix))
        s += '. ' + L
        print(s)
```