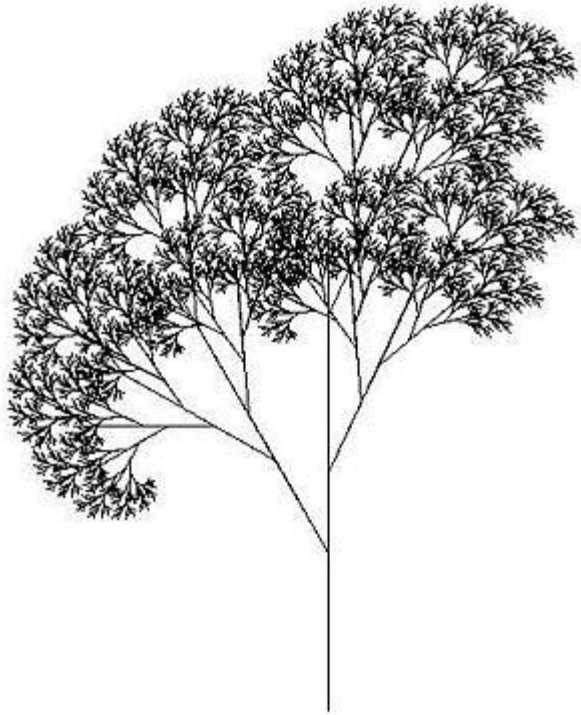# Recursion

How Python handles function calls
Recursive functions
Design and coding recursive functions
Analysing recursive functions
Example recursive functions
Python stack limits

Activation Records

When Python executes one of our functions, it creates an *activation record* that stores
- variables passed in as parameters
- local variables created inside the function
- the point in the function source code it has reached

and pushes it onto a *stack* of activation records.

When it finishes executing the function, it:
- pops the record off the stack and deletes it, remembering any return value
- moves to the record now on top of the stack, and returns to the point in the source code it had reached, and passes in the return value

```
def fa(x):
    y = x * 2
    z = fb(y)
    return x + z

def fb(w):
    t = w + 7
    q = fc(t)
    return q + w

def fc(p):
    s = p+1
    return s

fa(4)
```

If you want to use a value returned from a function, every path through the code *must* reach a return statement *in the function.*

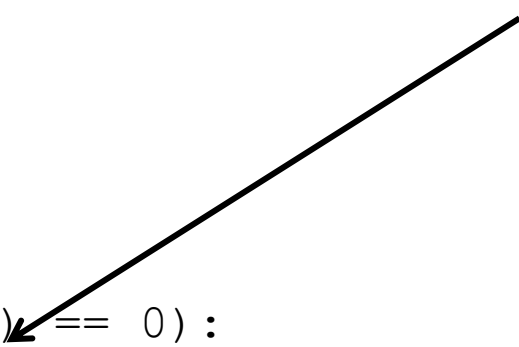Call Stack

Recursion: revision

Example: recursive function to compute sum of all values in a list

base case – a non recursive codeblock that exits the function

```
def rsum(mylist):
    if (len(mylist) == 0):
        return 0
    endsum = rsum(mylist[1:])
    return mylist[0] + endsum
```
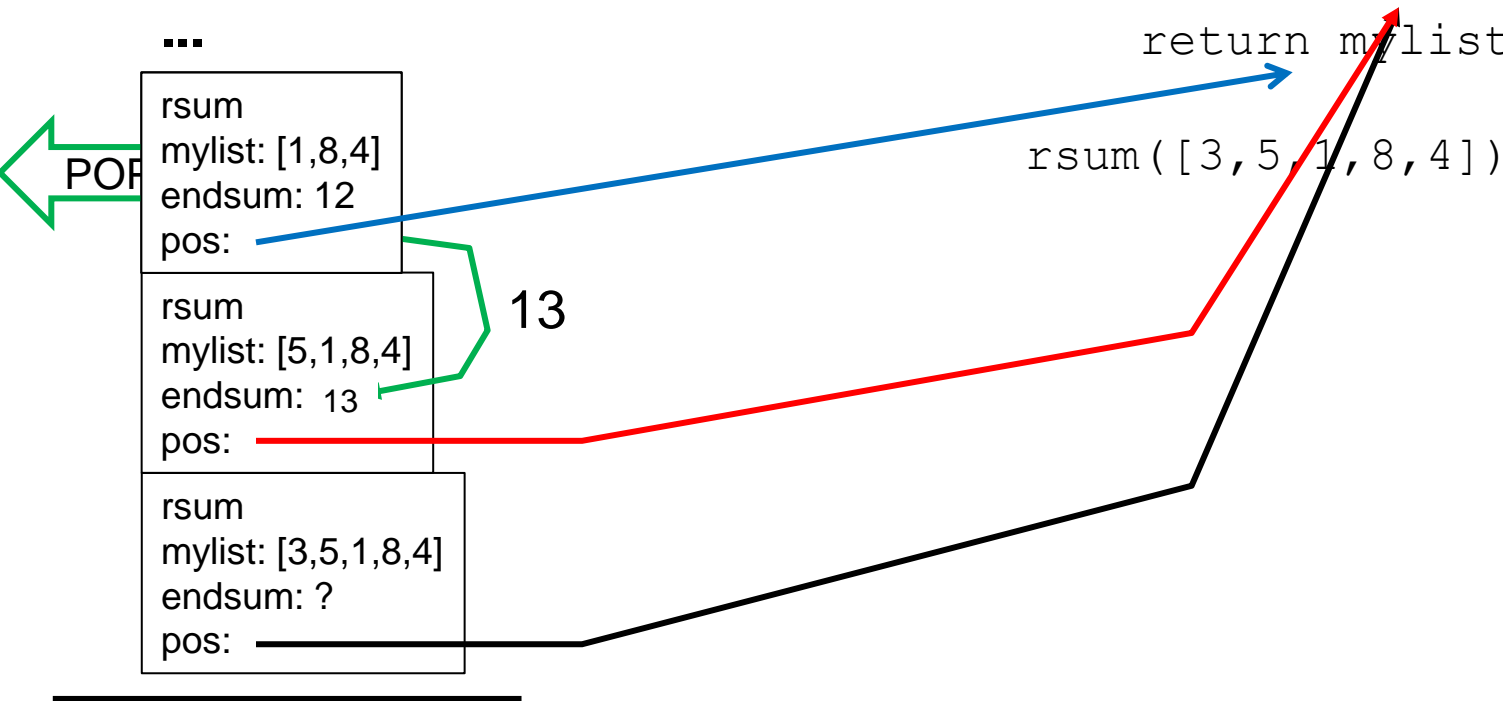
recursive call – a call to the same function, but with parameters closer to reaching a base case

Python handles recursive functions on the call stack in the same way as any other function

```
def rsum(mylist):
    if (len(mylist) == 0):
        return 0
    endsum = rsum(mylist[1:])
    return mylist[0] + endsum
```

rsum([3,5,1,8,4])

...

rsum
mylist: [1,8,4]
POP ← endsum: 12
pos:

rsum
mylist: [5,1,8,4]   } 13
endsum: 13
pos:

rsum
mylist: [3,5,1,8,4]
endsum: ?
pos:

Call Stack

What's wrong with this function (to compute the maximum value in list)?

```python
def maxlist(mylist):
    return max(mylist[0], maxlist(mylist[1:]))
```

There *must* be at least one base case to stop the recursion.

All computation paths through the recursive calls *must* lead to a base case.

What's wrong with this method (searching in a BST)?

```
def search(self, item):
    if item < self._element:
        if self._leftchild:
            self._leftchild.search(item)
    elif item > self._element:
        if self._rightchild:
            self._rightchild.search(item)
    else:
        return self
```

If you are returning a value, *every* path through the function, *at this level*, must end in a return statement

What's wrong with this method (to compute x to the power of n, where n>= 0)?

```python
def power(x,n):
    if n == 0:
        return 1
    elif n > 0:
        return x * power(x, n+1)
```

The recursive step *must* take you closer to the base case

What is the difference?

```
def recsum1(mylist):
    if (len(mylist) == 0):
        return 0
    remaindersum = recsum1(mylist[1:])
    return mylist[0] + remaindersum

def recsum2(mylist, i=0):
    if (len(mylist) == i):
        return 0
    remaindersum = recsum2(mylist, i+1)
    return mylist[i] + remaindersum
```

List slicing creates a new list, and so must copy all references from the old list into the new (sliced) list on each recursive call.

Analysing recursive functions

Basic approach:

1. count the worst-case work done by a single activation, *without* the recursive call

2. count the worst-case number of recursive calls
3. multiply the two counts together

```
def rsum(mylist, i=0):
    if (len(mylist) == i):
        return 0
    rsum = rsum(mylist, i+1)
    return mylist[i] + rsum
```
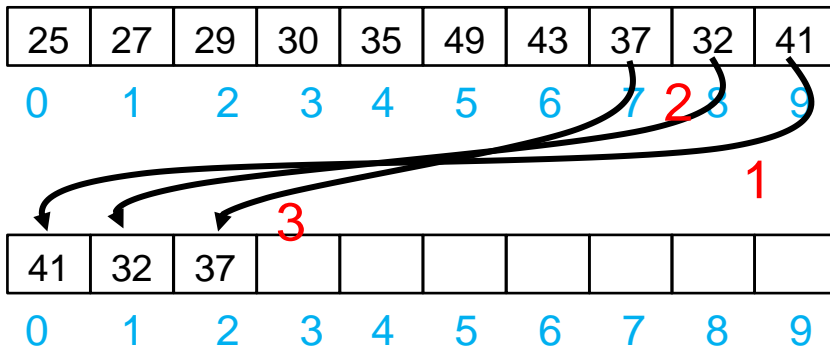
constant operations per activation (1 length check, 1 comparison, 1 lookup, 2 additions)

1 (recursive) call of rsum for each element of list

Therefore *O(n)*

Examples: reversing a list (1)

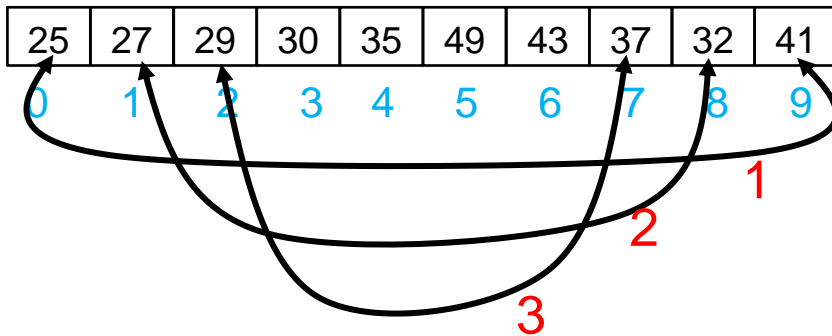Reverse a list, using recursion, method 1

Can be done with slicing, or using indices, but needs to create an extra list, so doubles the space requirement.

```
def rev1(mylist, i=0):
    if len(mylist) == i:
        return []
    retlist = rev1(mylist, i+1)
    retlist.append(mylist[i])
    return retlist
```

O(n)

Examples: reversing a list (2)

Reverse a list, using recursion, method 2

Also linear recursion.
   2 assignments,
   1 comparison
   3 arithmetic operations
per activation.
n/2 activations.
So O(n)
(and no extra space)

```
def rev4(mylist):
    return rev4plus(mylist, 0, len(mylist)-1)

def rev4plus(mylist, begin, end):
    if begin < end - 1:
        mylist[begin], mylist[end] = mylist[end], mylist[begin]
        return rev4plus(mylist, begin+1, end-1)
    return mylist
```

Examples: xⁿ

```
def power(x,n):
    if n == 0:
        return 1
    elif n > 0:
        return x * power(x, n-1)
```

Linear recursion
     2 comparisons
     2 arithmetic ops
n recursive calls
O(n)

Can we improve the efficiency?

```
def power2(x,n):
    if n == 0:
        return 1
    elif n > 0:
        y = power(x, n//2)
        y = y*y
        if n%2 == 1:
            y = y*x
        return y
```

Linear recursion
     3 comparisons
     4 arithmetic ops
log n recursive calls
O(*log n*)

Examples: BST size

```
def size(self):
    sz = 1
    if self._leftchild:
        sz += self._leftchild.size()
    if self._rightchild:
        sz += self._rightchild.size()
    return sz
```

Binary recursion (since there may be 2 recursive calls per activation)
   2 comparisons
   3 arithmetic ops
<= 2 recursive calls per activation, each node in the tree activates once
O(n)

Note: could also say each non-root node in the tree is activated once by its parent, and the root is activated at the start, so *n* activations.

Examples: File structure size (du)

```
size(directory):        #pseudocode
    sz = 0
    for each file in directory
        sz += filesize(file)
    for each sub-directory in directory
        sz += size(sub-directory)
    return sz
```

Assume n directories or sub-directories

Multiple recursion (since there may be $n-1$ recursive calls per activation)
  m calls of `filesize`, where m is max number of files in any directory
<= $n-1$ recursive calls per activation, each node in the tree activates once
O(mn^2)

That is a loose bound on the worst-case complexity. By the argument on the previous slide, there are n directories, and each directory is activated once, so O(mn). And if there are t files in total, we get O($n+t$)

This is still wrong, but what happens if we run it in Python?
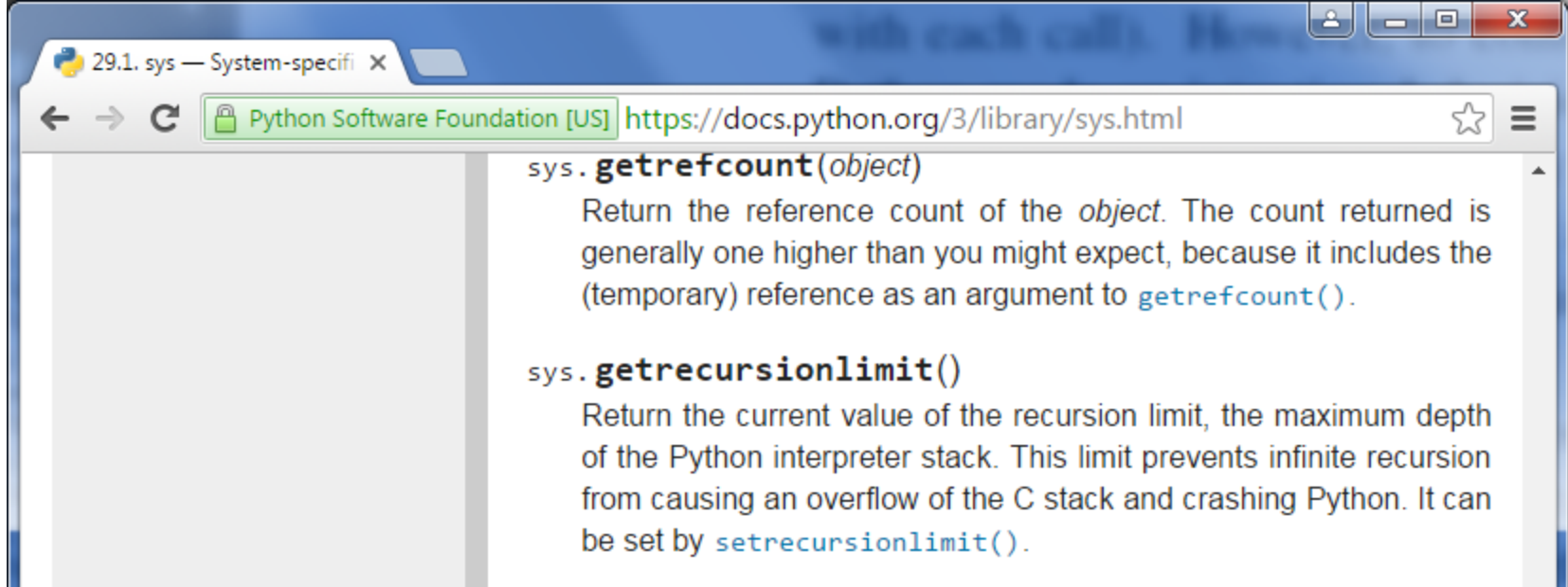
```python
def power(x,n):
    if n == 0:
        return 1
    elif n > 0:
        return x * power(x, n+1)
```

Unbounded recursion will keep adding activation records to the call stack until we run out of memory.

Memory fills so quickly, and the CPU is overworked, so the system behaviour is unspecified.

Python imposes a limit on the size of the call stack to stop this.

sys.**getrefcount**(*object*)

Return the reference count of the *object*. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to `getrefcount()`.

sys.**getrecursionlimit**()

Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by `setrecursionlimit()`.

...

sys.**setrecursionlimit**(*limit*)

Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

If the new limit is too low at the current recursion depth, a `RecursionError` exception is raised.

Changed in version 3.5.1: A `RecursionError` exception is now raised if the new limit is too low at the current recursion depth.

Note: this sets a limit on the stack size, and not just on the number of recursive calls.

Next Lecture

Complexity Analysis