Recursion

All Versions

Examples

The What, How, and When of Recursion

Recursion occurs when a function call causes that same function to be called again before the original function call terminates. For example, consider the well-known mathematical expression x! (i.e. the factorial operation). The factorial operation is defined for all nonnegative integers as follows:

- If the number is 0, then the answer is 1.
- Otherwise, the answer is that number times the factorial of one less than that number.

In Python, a naïve implementation of the factorial operation can be defined as a function as follows:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Recursion functions can be difficult to grasp sometimes, so let's walk through this step-by-step. Consider the expression factorial(3) . This and *all* function calls create a new **environment** . An environment is basically just a table that maps identifiers (e.g. n , factorial , print , etc.) to their corresponding values. At any point in time, you can access the current environment using locals() . In the first function call, the only local variable that gets defined is n = 3 . Therefore, printing locals() would show {'n': 3} . Since n == 3 , the return value becomes n * factorial(n - 1) .

At this next step is where things might get a little confusing. Looking at our new expression, we already know what n is. However, we don't yet know what factorial(n - 1) is. First, n - 1 evaluates to 2 . Then, 2 is passed to factorial as the value for n . Since this is a new function call, a second environment is created to store this new n . Let *A* be the first environment and *B* be the second environment. *A* still exists and equals {'n': 3} , however, *B* (which equals {'n': 2}) is the current environment. Looking at the function body, the return value is, again, n * factorial(n - 1) . Without evaluating this expression, let's substitute it into the original return expression. By doing this, we're mentally discarding *B* , so remember to substitute n accordingly (i.e. references to *B* 's n are replaced with n - 1 which uses *A* 's n). Now, the original return expression becomes n * ((n - 1) * factorial((n - 1) - 1)) . Take a second to ensure that you understand why this is so.

Now, let's evaluate the factorial((n - 1) - 1)) portion of that. Since *A* 's n == 3 , we're passing 1 into factorial . Therefore, we are creating a new environment *C* which equals {'n': 1} . Again, the return value is n * factorial(n - 1) . So let's replace factorial((n - 1) - 1)) of the “original” return expression similarly to how we adjusted the original return expression earlier. The “original” expression is now n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1))) .

Almost done. Now, we need to evaluate factorial((n - 2) - 1) . This time, we're passing in 0 . Therefore, this evaluates to 1 . Now, let's perform our last substitution. The “original” return expression is now n * ((n - 1) * ((n - 2) * 1)) . Recalling that the original return expression is evaluated under *A* , the expression becomes 3 * ((3 - 1) * ((3 - 2) * 1)) . This, of course, evaluates to 6. To confirm that this is the correct answer, recall that 3! == 3 * 2 * 1 == 6 . Before reading any further, be sure that you fully understand the concept of environments and how they apply to recursion.

The statement if n == 0: return 1 is called a base case. This is because, it exhibits no recursion. A base case is absolutely required. Without one, you'll run into infinite recursion. With that said, as long as you have at least one base case, you can have as many cases as you want. For example, we could have equivalently written factorial as follows:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

You may also have multiple recursion cases, but we won't get into that since it's relatively uncommon and is often difficult to mentally process.

You can also have “parallel” recursive function calls. For example, consider the Fibonacci sequence which is defined as follows:

- If the number is 0, then the answer is 0.
- If the number is 1, then the answer is 1.
- Otherwise, the answer is the sum of the previous two Fibonacci numbers.

We can define this is as follows:

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

I won't walk through this function as thoroughly as I did with factorial(3) , but the final return value of fib(5) is equivalent to the following (*syntactically* invalid) expression:

```
(
  fib((n - 2) - 2)
  +
  (
    fib(((n - 2) - 1) - 2)
    +
    fib(((n - 2) - 1) - 1)
  )
)
+
(
  (
    fib(((n - 1) - 2) - 2)
    +
    fib(((n - 1) - 2) - 1)
  )
  +
  (
    fib(((n - 1) - 1) - 2)
    +
    (
      fib((((n - 1) - 1) - 1) - 2)
      +
      fib((((n - 1) - 1) - 1) - 1)
    )
  )
)
```

This becomes (1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1))) which of course evaluates to 5 .

Now, let's cover a few more vocabulary terms:

- A **tail call** is simply a recursive function call which is the last operation to be performed before returning a value. To be clear, return foo(n - 1) is a tail call, but return foo(n - 1) + 1 is not (since the addition is the last operation).
- **Tail call optimization** (TCO) is a way to automatically reduce recursion in recursive functions.
- **Tail call elimination** (TCE) is the reduction of a tail call to an expression that can be evaluated without recursion. TCE is a type of TCO.

Tail call optimization is helpful for a number of reasons:

- The interpreter can minimize the amount of memory occupied by environments. Since no computer has unlimited memory, excessive recursive function calls would lead to a **stack overflow** .
- The interpreter can reduce the number of **stack frame** switches.

Python has no form of TCO implemented for a number of a reasons . Therefore, other techniques are required to skirt this limitation. The method of choice depends on the use case. With some intuition, the definitions of factorial and fib can relatively easily be converted to iterative code as follows:

```
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product

def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

This is usually the most efficient way to manually eliminate recursion, but it can become rather difficult for more complex functions.

Another useful tool is Python's lru_cache decorator which can be used to reduce the number of redundant calculations.

You now have an idea as to how to avoid recursion in Python, but when *should* you use recursion? The answer is “not often”. All recursive functions can be implemented iteratively. It's simply a matter of figuring out how to do so. However, there are rare cases in which recursion is okay. Recursion is common in Python when the expected inputs wouldn't cause a significant number of a recursive function calls.

If recursion is a topic that interests you, I implore you to study functional languages such as Scheme or Haskell. In such languages, recursion is much more useful.

Please note that the above example for the Fibonacci sequence, although good at showing how to apply the definition in python and later use of the lru cache, has an inefficient running time since it makes 2 recursive calls for each non base case. The number of calls to the function grows exponentially to n . Rather non-intuitively a more efficient implementation would use linear recursion:

```
def fib(n):
    if n <= 1:
        return (n,0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

But that one has the issue of returning a *pair* of numbers. This emphasizes that some functions really do not gain much from recursion.

Increasing the Maximum Recursion Depth

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a RuntimeError exception is raised:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Here's a sample of a program that would cause this error:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))
cursing(0)
# Out: I recursed 1083 times!
```

It is possible to change the recursion depth limit by using

```
sys.setrecursionlimit(limit)
```

You can check what the current parameters of the limit are by running:

```
sys.getrecursionlimit()
```

Running the same method above with our new limit we get

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

From Python 3.5, the exception is a RecursionError, which is derived from RuntimeError.

Sum of numbers from 1 to n

If I wanted to find out the sum of numbers from `1` to `n` where `n` is a natural number, I can do `1 + 2 + 3 + 4 + ... + (several hours later) + n`. Alternatively, I could write a `for` loop:

```
n = 0
for i in range (1, n+1):
    n += i
```

Or I could use a technique known as recursion:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

Recursion has advantages over the above two methods. Recursion takes less time than writing out `1 + 2 + 3` for a sum from 1 to 3. For `recursion(4)`, recursion can be used to work backwards:

Function calls: (4 -> 4 + 3 -> 4 + 3 + 2 -> 4 + 3 + 2 + 1 -> 10)

Whereas the `for` loop is working strictly forwards: (1 -> 1 + 2 -> 1 + 2 + 3 -> 1 + 2 + 3 + 4 -> 10). Sometimes the recursive solution is simpler than the iterative solution. This is evident when implementing a reversal of a linked list.

Tail Recursion - Bad Practice

When the only thing returned from a function is a recursive call, it is refered to as tail recursion.

When the only thing returned from a function is a recursive call, it is refered to as tail recursion.

Here's an example countdown written using tail recursion:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Any computation that can be made using iteration can also be made using recursion. Here is a version of find_max written using tail recursion:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

Tail recursion is considered a bad practice in Python, since the Python compiler does not handle optimization for tail recursive calls. The recursive solution in cases like this use more system resources than the equivalent iterative solution.

Tree exploration with recursion

Say we have the following tree:

```
root
- A
  - AA
  - AB
- B
  - BA
  - BB
    - BBA
```

Now, if we wish to list all the names of the elements, we could do this with a simple for-loop. We assume there is a function `get_name()` to return a string of the name of a node, a function `get_children()` to return a list of all the sub-nodes of a given node in the tree, and a function `get_root()` to get the root node.

```
root = get_root(tree)
for node in get_children(root):
    print(get_name(node))
    for child in get_children(node):
        print(get_name(child))
        for grand_child in get_children(child):
            print(get_name(grand_child))
# prints: A, AA, AB, B, BA, BB, BBA
```

This works well and fast, but what if the sub-nodes, got sub-nodes of its own? And those sub-nodes might have more sub-nodes... What if you don't know beforehand how many there will be? A method to solve this is the use of recursion.

```
def list_tree_names(node):
    for child in get_children(node):
        print(get_name(child))
        list_tree_names(node=child)

list_tree_names(node=get_root(tree))
# prints: A, AA, AB, B, BA, BB, BBA
```

Perhaps you wish to not print, but return a flat list of all node names. This can be done by passing a rolling list as a parameter.

```
def list_tree_names(node, lst=[]):
    for child in get_children(node):
        lst.append(get_name(child))
        list_tree_names(node=child, lst=lst)
    return lst

list_tree_names(node=get_root(tree))
# returns ['A', 'AA', 'AB', 'B', 'BA', 'BB', 'BBA']
```

Tail Recursion Optimization Through Stack Introspection

Tail Recursion Optimization Through Stack Introspection

By default Python's recursion stack cannot exceed 1000 frames. This can be changed by setting the sys.setrecursionlimit(15000) which is faster however, this method consumes more memory. Instead, we can also solve the Tail Recursion problem using stack introspection.

```
#!/usr/bin/env python2.4
# This program shows off a python decorator which implements tail call optimization. It
# does this by throwing an exception if it is it's own grandparent, and catching such
# exceptions to recall the stack.

import sys

class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs

def tail_call_optimized(g):
"""
This function decorates a function with tail call
optimization. It does this by throwing an exception
if it is it's own grandparent, and catching such
exceptions to fake the tail call optimization.

This function fails if the decorated
function recurses in a non-tail context.
"""

    def func(*args, **kwargs):
        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
            raise TailRecurseException(args, kwargs)
        else:
            while 1:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException, e:
                    args = e.args
                    kwargs = e.kwargs
    func.__doc__ = g.__doc__
    return func
```

To optimize the recursive functions, we can use the @tail_call_optimized decorator to call our function. Here's a few of the common recursion examples using the decorator described above:

Factorial Example:

```
@tail_call_optimized
def factorial(n, acc=1):
  "calculate a factorial"
  if n == 0:
    return acc
  return factorial(n-1, n*acc)

print factorial(10000)
# prints a big, big number,
# but doesn't hit the recursion limit.
```

Fibonacci Example:

```
@tail_call_optimized
def fib(i, current = 0, next = 1):
  if i == 0:
    return current
  else:
    return fib(i - 1, next, current + next)

print fib(10000)
# also prints a big number,
# but doesn't hit the recursion limit.
```

Syntax

Parameters

Remarks

Recursion needs a stop condition `stopCondition` in order to exit the recursion.

The original variable must be passed on to the recursive function so it becomes stored.