Computer Science I Recursion to Iteration

CSCI-141 Lecture

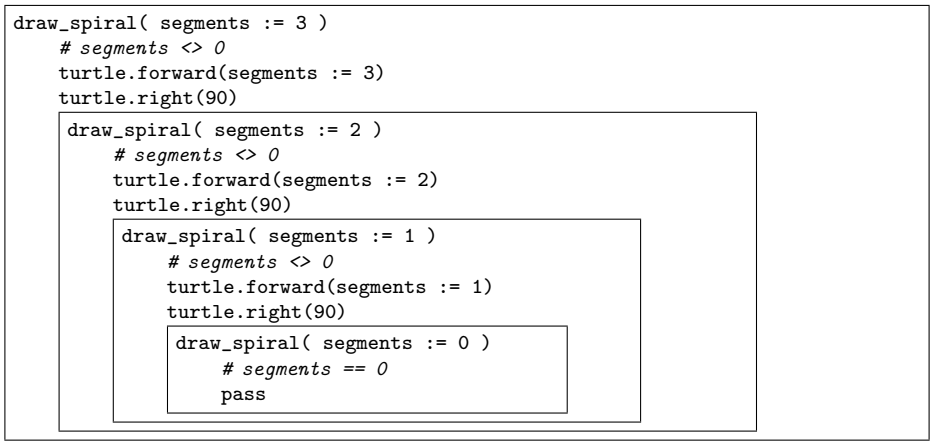
09/16/2018

1 Tail-Recursion

Consider a recursive function, `draw_spiral`, which draws lines at right angles to each other that spiral inward.

```
def draw_spiral(segments):
    if segments == 0:
        pass
    else:
        turtle.forward(segments)
        turtle.right(90)
        draw_spiral(segments - 1)
```

Here is a corresponding execution diagram.



Note that there is nothing more to execute after the function makes the recursive call. The call boxes collapse one after another. We are simply executing the code over and over again with different values for the parameter `segments`.

The `draw_spiral` function exhibits *tail-recursion*. We refer to this kind of recursion as such because the recursive call is the *last* thing to occur in the function; it occurs at the *tail* end of the function.

Why is tail-recursion so important? The reason is that if a function is tail recursive, the computer does not have to go back to the earlier invocations of the function to finish the work. It tends to be far more efficient for a computer to work in this way.

2 Tail-Recursion and Fruitful Functions

Recall the factorial function.

```
def factorial( number ):
    """ Compute the factorial of a number.
        :param number: The number for which Factorial is to be
                       computed.
        :preconditions: number is non-negative
    """
    if number == 0:
        return 1
    else:
        return number * factorial( number - 1 )
```

This function is *not* tail recursive. Even though it may at first appear that the recursive call is made last, upon closer inspection we realize that the recursive call must be made *before* the result is multiplied by the `number`. We can make this even more obvious by expanding the last line of the function into two discrete statements:

```
result = factorial( number - 1 ) # we must compute this...
return number * result           # ...before we can do this
```

There is nothing inherently wrong with this implementation, but we can modify the `factorial` function so that it *is* tail-recursive, which will enable us to experiment with alternate implementations.

Below is a tail-recursive formulation of the factorial function using *accumulation*; an additional parameter called an *accumulator* is added. In the previous implementation, each call to the `factorial` function waits for the next call to return a result before computing its part of the final result. The addition of an accumulator reverses this process; each call to the `factorial` function computes its part of the final result and then passes it in to the recursive call using the `accumulator`. This allows the recursive call to be made last. Ultimately the base case is modified to return the final result. Observe that `factorial_accum` is indeed tail-recursive; the recursive call is at the tail end of the function.

```
def factorial_accum(number, accumulator):
  if number == 0:
    # we are done; return the accumulated result
    return accumulator
  else:
    # use the accumulator to compute one part
    accumulator = number * accumulator
    # pass it into the recursive call
    return factorial_accum( number - 1, accumulator )
```

The idea is that, at each step of execution, the accumulator contains the "result so far".

For example, if the call `factorial_accum(5,42)` executes, we can assume that 42 is the accumulated result of multiplying together all of the integers greater than 5 in the original problem. The job of this specific call then is to "add" in the remaining factors (5, 4, 3, 2, and 1) to the product. Like all recursive functions, this function does this by doing a single unit of work (i.e. "add" the next factor 5: $5 \times 42 = 210$), and then making a recursive call to handle the remaining factors: `factorial_accum(4,210)`.

Now we can understand the base case. When `number` reaches 0, all of the integers from the original problem have been multiplied together, and the answer has been stored in `accumulator`, its *accumulation parameter*. Thus, the base case returns the accumulated product. We observe that

$$\texttt{factorial\_accum}(\textit{number}, \textit{accumulator}) = \textit{number}! \times \textit{accumulator}$$

but that is not enough to guarantee the necessary precondition that `accumulator` starts out with the correct initial value of `1`. If any other initial value is used, the function will not calculate the proper result. From the perspective of the client calling the function, this is more than a little strange. Not only does the client need to specify a second parameter when all they really want is to calculate the factorial for some `number`, they must *always* be sure to specify a value of 1, or they won't get the correct answer.

Thankfully, Python provides two simple solutions to this problem. The first is to treat the `factorial_accum` function as a "private" function that is not meant to be called directly by the client. Instead, a "public" function is provided for the client that takes only one parameter, `number`, and it calls the `factorial_accum` function with the correct initial value for the accumulator[1].

```
def factorial_user(number):
    return factorial_accum(number, 1)
```

An alternative solution is to assign a default value to the `accumulator` parameter. In the event that the client omits the corresponding argument when calling the function, the default parameter value is used instead. In Python, default parameter values are specified as follows:

```
def factorial_accum(number, accumulator=1):
    ...
```

The client may call the function with or without an argument for the `accumulator` parameter. Therefore, both of the following invocations are valid for calculating the factorial of 100.

```
result_1 = factorial_accum(100, 1)
result_2 = factorial_accum(100)
```

The solution that a programmer chooses is usually a matter of personal preference.

---

1. While some programming languages include syntax for hiding functions from clients by making them explicitly private, Python does not. Here "private" is used to indicate "not intended for clients to call directly."

2.1 Example: Substitution Trace of Tail-Recursive Factorial

$$\begin{aligned}
\texttt{factorial_user(3)} &= \texttt{factorial_accum(3, 1)} \\
&= \texttt{factorial_accum(2, 3)} \\
&= \texttt{factorial_accum(1, 6)} \\
&= \texttt{factorial_accum(0, 6)} \\
&= 6
\end{aligned}$$

3 Reassignment, or, Changing Variable Values

Tail-recursion can always be understood as *iteration* or repetition. Again consider the `draw_spiral` algorithm, which can legitimately be expressed as follows.

```
repeatedly execute until we are done:
    if segments == 0:
        we are done
    else:
        move the turtle forward segments
        turn right 90 degrees
        change segments to the value of segments - 1
```

The algorithm leaves us with a puzzle:

- How can we change the value associated with a parameter in Python?
- What does it mean to change the value?

The statement "`change segments to the value of segments - 1`" is expressed succinctly in Python as `segments = segments - 1`. However, the Python statement may be even more confusing! It looks like an equation, but it's not. It cannot be an equation, because the equation $s = s - 1$ implies $0 = -1$, which is a contradiction.

In computer programming, a *variable* is a symbolic name that is used to reference some information. As the name *variable* suggests, the information bound to a variable can change, and a variable name can be used in code independently of the information it represents.

A function parameter (such as `segments`) is a special case of a variable. It is a variable that is associated with an input to a function.

Until now, it has appeared that variables referred to specific values such as the number 5 or the string "abc". However we now see that the variable `segments` cannot and does not refer to a number, because numbers don't change. If variables don't refer to numbers, to what do they refer?

A variable refers to an *address*; an address is a name for a location in computer memory. Python tracks which variable is associated with which address in a table. The computer memory itself also resembles a table. Thus the following sequence of assignments leads to the following characterization involving the variable table and memory.

```
x = 15
y = 18
z = 20
```

Variable Table
`x` : a_0
`y` : a_1
`z` : a_2

Memory	
a_0	15
a_1	18
a_2	20

Most expressions, including the expression on *the right-hand side of an assignment*, evaluate variables by looking up the variable in the variable table, finding its associated address, and then using the address to look at the appropriate place in memory to find the value.

In contrast to a mathematical equation, *the left-hand side of an assignment* is treated differently from the right-hand side. When evaluating the variable on the left-hand side, the assignment looks up only the address. Execution replaces the contents at that address with the value that was computed by evaluating the right-hand side. Thus, when executing `x = x + z`, the right-hand side evaluates to 35, the left-hand side evaluates to a_0, 35 is stored in the memory location named a_0, and we have the following characterization involving the variable table and memory.

Variable Table
`x` : a_0
`y` : a_1
`z` : a_2

Memory	
a_0	35
a_1	18
a_2	20

Now let's return to the reason we needed to change the value of a variable.

4 Iteration Using the while Loop

Recall the expression of the tail recursive implementation of the `draw_spiral` function:

```
repeatedly execute until we are done:
    if segments == 0:
        we are done
    else:
        Move the turtle forward segments
        Turn right 90 degrees
        Change segments to the value of segments - 1
```

The phrase "`repeatedly execute ...`" can be translated into Python code using a special statement: `while`. `while` is similar to `if` in that it evaluates some boolean expression

and is followed by a *body* comprising a series of indented statements. Unlike `if`, which will execute the statements in its body *once* if and only if the boolean expression evaluates to `True`, `while` will *repeatedly* execute the statements in its body as long as the boolean expression *remains* `True`. And so, the phrase "`repeatedly execute`" may be implemented in Python as `while True`.

This kind of statement is referred to as a *loop* because, once the last statement in the body is executed, control *loops back* to the top, at which point the `while` statement will reevaluate the boolean expression. If the boolean expression is still `True`, the statements in the body will execute again. This continues until the expression evaluates to `False`.

When using recursion, repetition occurs in the recursive case when the function calls itself. But repetition occurs in a `while` loop without making an additional function call. Thus, the above algorithm may be implemented as follows:

```
while True: # repeatedly execute...
    turtle.forward(segments)
    turtle.right(90)
    segments = segments - 1
```

Of course this presents a problem: there is no "`we are done`" that stops the repetition. The boolean expression `True` will never evaluate to `False`, and execution will continue forever (or, more likely, until the user terminates the program). In recursion, repetition stops ultimately at the *base case*, usually when an `if` condition determines that it should not make another recursive call. In Python, the `break` statement can be used inside of a `while` loop to similar effect; a `break` will immediately stop execution of a loop and control will flow to the first statement immediately following the body (i.e. the first unindented statement following the loop).

```
while True: # repeatedly execute...
    if segments == 0:  # ... until we are done
        break
    else:
        turtle.forward(segments)
        turtle.right(90)
        segments = segments - 1
```

The above code is not a typical way to utilize a `while` loop construct. It was only written in that way in order to make clear the parallels between the tail recursive algorithm and the iterative (looping) algorithm. If the `while` statement evaluates the boolean expression before each iteration, why not leverage this to validate the number of segments? Below is the code as it would be written if it was decided from the start to use a `while` loop.

```
def draw_spiral( segments ):
    while segments != 0: # repeatedly execute until we are done
        forward(segments)
        right(90)
        segments = segments - 1
```

Note that the `break` statement is no longer needed[2]. This is because the `while` statement will re-evaluate its boolean expression before each new iteration. With each iteration, `segments` is *reassigned* to the value of `segments - 1`. When the value of `segments` reaches 0, the boolean expression `segments != 0` will evaluate to `False`. This will terminate the iteration, and the flow of control will move to the first statement following the body of the `while` loop[3].

4.1 Tail Recursion and Iteration

So why discuss tail recursion and iteration together?

Any algorithm that may be implemented using tail recursion may also be implemented using a `while` loop. There are incentives for choosing iteration over tail recursion, not the least of which is avoiding the danger of exceeding the limit on the number of functions allowed on the call stack. Recall that even a simple recursive function may "blow up the stack" if the input is large enough:

```
def count_down(number):
    if number < 0:
        pass
    else:
        print(number)
        number = number - 1
        count_down(number)

# this will exceed the maximum depth on the call stack
count_down(10000)
```

Iteration may be used to accomplish the same task without adding additional stack frames to the call stack. The number of the calls on the stack does not increase because no additional function calls are made. Therefore, there is no danger of a stack overflow, which Python reports as a `RecursionError`.

```
def count_down(number):
    while number >= 0:
        print(number)
        number = number - 1

count_down(10000)
```

---

2. In fact, use of break statements is often considered a "code smell," meaning that there is often a better, more elegant way to solve the problem.
3. In this example, there are no statements following the while loop. In this case the function will finish executing and return.

5 Re-Implementing Tail Recursive Functions with Accumulation and Iteration

Any tail-recursive function may be rewritten to use iteration. Let's examine two more examples of functions that we have implemented previously: Factorial and the Fibonacci Sequence[4].

5.1 Factorial

Let's begin again by expressing the factorial algorithm using repetition:

```
repeatedly execute until we are done:
    if number == 0:
        we are done, return accumulator
    else:
        change accumulator to the value of accumulator * number
        change number to the value of number - 1
```

As before, we can implement this algorithm using a `while` loop.

```
def factorial_loop(number):
    accumulator = 1 # accumulator begins at 1
    while True: # repeatedly execute...
        if number == 0: # ... until we are done
            return accumulator
        else:
            accumulator = accumulator * number
            number = number - 1
```

Notice the use of a `return` statement rather than a `break` statement in the body of the loop. This will have the desired effect of terminating the iteration because, any time a `return` statement is executed by a Python program, the function will immediately stop executing and return control to the caller. This is no exception. However, we can leverage the `while` loop's boolean expression to implement the loop in a different way that is perhaps more understandable to the reader[5].

```
def factorial_loop(number):
    accumulator = 1         # accumulator begins at 1
    while number != 0:      # repeatedly execute...
        accumulator = accumulator * number
        number = number - 1
    return accumulator
```

4. Wouldn't "Factorial and the Fibonacci Sequence" be a great name for a band?
5. It is a simple truth that, all things being equal, it is better to write programs that are easier to understand so that other programmers can more easily grasp what it is that the program is meant to do.

Notice that, as with the previous example, the boolean expression used by the base case is inverted for use with the `while` loop; it has been changed from `number == 0` to `number != 0`. This makes sense because, previously, the expression was used in this way: "if this condition is True, work should stop." But the `while` loop uses it in the opposite way: "if this condition is True, work should continue." For this reason, it is very common to invert the condition used by the base case in a tail-recursive function when reimplementing the algorithm using iteration.

5.2 The Fibonacci Sequence

Let us now review the accumulator technique and iteration applied to the Fibonacci Sequence. It turns out that in this case doing so greatly speeds up the computation!

Recall the mathematical definition of the Fibonacci function.

$$
\begin{aligned}
&fibo(0) = 0 \\
&fibo(1) = 1 \\
&fibo(n) = fibo(n-1) + fibo(n-2), \quad n > 1
\end{aligned}
$$

Translating this definition directly to code results in the following function:

```
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

This is not wrong, but it is inefficient. Observe that in executing $fibo(n-1)$ we in turn must again execute $fibo(n-2)$. For example, to compute $fibo(4)$ we compute $fibo(3)$ and $fibo(2)$. But computing $fibo(3)$ computes $fibo(2)$ and $fibo(1)$, meaning that $fibo(2)$ gets executed twice. It is for this reason that we refer to this implementation of the Fibonacci function as the *naive* implementation. A better solution is to save previously computed values for future use using the accumulator pattern. We shall observe that the execution time is reduced from *exponential* time to *linear* time, with respect to the argument value.

The accumulative Fibonacci requires us to keep two accumulator values: the last two values computed for the next two lower values of n. As before, it seems more natural to work our way up – 0,1,1,2,3,5,8,...; How do we know when to stop? If we count down as the accumulated answer increases, we always stop at the same point. As the counter `n` *decreases*, we accumulate larger and larger values in the sequence. The tail-recursive solution is shown below.

```
def fibo_accum(n, fn_1=1, fn_2=0):
    """Calculates the nth number in the Fibonacci
       sequence using tail recursion.
       :param n: The index of the desired number in the sequence
       :param fn_1: The calculated value of F(n-1)
       :param fn_2: The calculated value of F(n-2)
       :preconditions: n is greater than or equal to 0.
    """
    if n == 0:
        return fn_2
    elif n == 1:
        return fn_1
    else:
        temp = fn_1
        fn_1 = fn_1 + fn_2 # new F(n-1) = old F(n-1) + old F(n-2)
        fn_2 = temp        # new F(n-2) = old F(n-1)
        return fibo_accum(n-1, fn_1, fn_2)
```

Note the use of the variable `temp` to store the old value of `fn_1`. This is because we cannot assign a new value to `fn_1` *and* assign its old value to `fn_2`. The third variable is necessary to *temporarily* store the old value of `fn_1` so that it can be used later. We refer to these local variables as *temporary variables* for what are hopefully obvious reasons.

5.2.1 Example: Substitution Trace of Tail-Recursive Fibonacci

$$\begin{aligned}
\texttt{fibo_accum(4)} &= \texttt{fibo_accum(4, 1, 0)} \\
&= \texttt{fibo_accum(3, 1, 1)} \\
&= \texttt{fibo_accum(2, 2, 1)} \\
&= \texttt{fibo_accum(1, 3, 2)} \\
&= 3
\end{aligned}$$

Some people may claim that recursion is inherently inefficient and give the Fibonacci algorithm as a classic example of that inefficiency. We see here that recursion is not the fundamental flaw; the flaw is instead mechanically translating the mathematical definition into code without considering what the most efficient algorithm would be.

As with factorial, since we can understand tail-recursion as iteration, we can replace the recursive call with a loop. Observe that we must be somewhat careful about how we go about changing the values of the variables, since in each successive iteration one of the old values is kept (the old value of `fn_1` is kept as the new value of `fn_2`), hence the need for a temporary variable. The loop "runs forever" until exiting the loop. As with the previous example, the `return` statements exit the function.

```
def fibo_loop(n):
    """Calculates the nth number in the Fibonacci
       sequence using iteration.
       :param n: The index of the desired number in the sequence
       :preconditions: n is greater than or equal to 0.
    """
    # Set up fn_1 and fn_2 to be the first two values in the
    # sequence.
    fn_1 = 1
    fn_2 = 0
    while True:
        if n == 0:
            return fn_2
        elif n == 1:
            return fn_1
        else:
            temp = fn_1
            fn_1 = fn_1 + fn_2
            fn_2 = temp

            n = n - 1
```

As in previous examples, using `while True` is useful for the initial translation of the tail recursive implementation of the Fibonacci sequence into iteration, but the loop can be changed to make use of the boolean expression.

```
fn_1 = 1
fn_2 = 0
while n > 1:
    n = n - 1
    temp = fn_1
    fn_1 = fn_1 + fn_2
    fn_2 = temp
if n == 0:
    return fn_2
elif n == 1:
    return fn_1
```

6 Tracing Iteration with Timeline Diagrams

A substitution trace is impractical for an iterative construct since the function is usually only invoked once, and therefore we can't substitute new values for the parameters each time the function is called. Instead we use a timeline. A timeline shows how each variable changes for each iteration of the loop. In the example below, each variable corresponds to a row in the table. The first column of values refers to the initial values after initialization

and before iteration begins. Each subsequent column refers to the values at the *end* of each iteration of the loop.

6.1 Timeline example for `fibo_loop(5)`

n	5	4	3	2	1
fn_1	1	1	2	3	5
fn_2	0	1	1	2	3

7 Using Loops to Enable User Interaction

Loops can also be used outside of fruitful functions. Allowing the user to interact with a program through the terminal is a classic example. Here is a program called **adder** that adds a bunch of numbers entered one per line by the user. The program knows that there are no more numbers when the user enters a blank line. A sample session would run as follows.

```
$ python3 adder.py
Enter numbers, one per line.
Entering an empty line will cause the program to
display the sum of the entered numbers, then terminate.

> 1
> 5
> 9.9
>
The sum of the numbers entered is 15.9
```

The code below achieves the requirements.

```
def adder1():
    print( "Enter numbers, one per line." )
    print( "Entering an empty line will cause the program to" )
    print( "display the sum of the numbers, then terminate." )
    print()

    sum = 0
    line = input( "> " )                    # {*}
    while line != "":
        number = float( line )
        sum = sum + number
        line = input( "> " )                # repeated line {*}
    print( "The sum of the numbers entered is", sum )
```

Note that we had to repeat one of the lines of code. This was done so that the decision to terminate or not happens at the loop boundary, where the test expression in the `while` statement is located. Sometimes one has to repeat several lines of code, which can be error-prone because the programmer has to make sure those two sequences of lines are kept up-to-date together.

An alternative is to use a `break` statement. That statement is a specialized *go to* that jumps control out of the loop to the statement beyond it. The test in the `while` statement does that, too, but a `break` can appear anywhere in the loop body.

```
def adder2():
    print( "Enter numbers, one per line." )
    print( "Entering an empty line will cause the program to" )
    print( "display the sum of the numbers, then terminate." )
    print()

    sum = 0
    while True:
        line = input( "> " )
        if line =="":
            break
        number = float( line )
        sum = sum + number
    print( "The sum of the numbers entered is", sum )
```

We have now reintroduced two other constructs that some consider troublesome.

1. The loop appears to be an *infinite loop*.
2. Execution of the loop the final time is incomplete.

The programmer has to assess which convention is preferred.

8 Sample Implementation

See `factorial.py`, `fibonacci.py`, `spiral.py`, and `adder.py` for the complete examples.