Lecture 10.1 : Recursion

Introduction ¶

- A *recursive* solution is one where the solution to a problem is expressed as an operation on a *simplified* version of the *same* problem.
- For certain problems, recursion may offer an intuitive, simple, and elegant solution. The ability to both recognise a problem that lends itself to a recursive solution and to implement that solution is an important skill that will make you a better programmer. Furthermore, some programming languages, such as Prolog (which you will meet in second year), make heavy use of recursion.
- We introduce recursion below and implement, in Python, recursive solutions to a selection of programming problems.

What is recursion?

- Any function that calls itself is *recursive* and exhibits *recursion*.

```
def foo(n):
    return foo(n-1)
```

- The function `foo()` above is recursive. *It calls itself*. Let's try calling `foo()` and see what happens:

```
>>> from recursion import foo
>>> foo(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./recursion.py", line 2, in foo
    return foo(n-1)
  File "./recursion.py", line 2, in foo
    return foo(n-1)
  File "./recursion.py", line 2, in foo
    return foo(n-1)
  File "./recursion.py", line 2, in foo
    return foo(n-1)
  File "./recursion.py", line 2, in foo
    return foo(n-1)
  File "./recursion.py", line 2, in foo
    return foo(n-1) # etc. etc. etc.
  File "./recursion.py", line 2, in foo
    return foo(n-1)
RuntimeError: maximum recursion depth exceeded
```

- Hmm. Our program crashed! What's going on? Well we initially invoke `foo(10)`, which invokes `foo(9)` which invokes `foo(8)` which invokes `foo(7)` which invokes `foo(6)` which invokes…

- Thus our initial `foo(10)` call is the first in an *infinite* sequence of calls to `foo()`. Computers do not like an infinite number of anything. For each of our `foo()` function invocations Python instantiates a data structure to represent that particular call to the function. That data structure is called a *stack frame*. A stack frame occupies memory. Our program attempts to create an infinite number of stack frames. That would require an infinite amount of memory. Our computer does not have an infinite amount of memory. So our program crashes (after a while).

- The problem with our recursive function is that it *never* fails to invoke itself and thus exhibits *infinite recursion*.
- To prevent infinite recursion we need to insert a *base case* into our function. Let's rewrite our function as `bar()` but this time cause it to stop once its parameter hits zero:

```
def bar(n):
    if n == 0: # base case : no more calls to bar()
        return 0
    return bar(n-1)
```

- Let's try calling `bar()` and see what happens:

```
>>> from recursion import bar
>>> bar(10)
0
```

- Why does bar return zero? Well `bar(10)` calls `bar(9)` which calls `bar(8)` … which calls `bar(0)`. The base case is `bar(0)`. It returns zero to `bar(1)` which returns zero to `bar(2)` which returns zero to `bar(3)` … which returns zero to `bar(10)` which returns zero which is our answer.
- That's how recursion works. So far so good. But can we use recursion to do something useful?

Summing the numbers 0 through N

- Assume we have a function `sum_up_to()`. Given an argument `N` `sum_up_to(N)` returns the sum all of the integers 0 through N. For example `sum_up_to(10)` sums the sequence 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0.
- Let's look at the `sum_up_to()` function in action:

```
>>> from recursion import sum_up_to
>>> sum_up_to(10)
55
>>> list(range(11)) # let's verify we got the right answer
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> sum(list(range(11)))
55
```

- Let's try some more examples:

```
>>> from recursion import sum_up_to
>>> sum_up_to(0)
0
>>> sum_up_to(1)
1
>>> sum_up_to(2)
3
>>> sum_up_to(3)
6
>>> sum_up_to(4)
```

```
10
>>> sum_up_to(5)
15
>>> sum_up_to(6)
21
>>> sum_up_to(7)
28
>>> sum_up_to(8)
36
>>> sum_up_to(9)
45
>>> sum_up_to(10)
55
```

- Do you notice anything recursive about the above sequence? Let's annotate each line to make the recursion obvious:

```
>>> from recursion import sum_up_to
>>> sum_up_to(0) # base case: returns zero
0
>>> sum_up_to(1) # returns 1 + sum_up_to(0)
1
>>> sum_up_to(2) # returns 2 + sum_up_to(1)
3
>>> sum_up_to(3) # returns 3 + sum_up_to(2)
6
>>> sum_up_to(4) # returns 4 + sum_up_to(3)
10
>>> sum_up_to(5) # returns 5 + sum_up_to(4)
15
>>> sum_up_to(6) # returns 6 + sum_up_to(5)
21
>>> sum_up_to(7) # returns 7 + sum_up_to(6)
28
>>> sum_up_to(8) # returns 8 + sum_up_to(7)
36
>>> sum_up_to(9) # returns 9 + sum_up_to(8)
45
>>> sum_up_to(10) # returns 10 + sum_up_to(9)
55
```

- For any argument `N`, `sum_up_to(N)` is equal to `N + sum_up_to(N-1)`. This is the essence of a recursive solution. The solution to the problem `sum_up_to(N)` is broken down into the operation `N` + on a simpler version of the same problem `sum_up_to(N-1)`. For example `sum_up_to(10)` is `10 + sum_up_to(9)`. The base case ensures recursion stops at some point. It encodes the fact that `sum_up_to(0)` is zero.

- Let's write the Python code that implements the `sum_up_to()` function:

```
def sum_up_to(n):
    if n == 0: # base case : no more calls to sum_up_to()
        return 0
    return n + sum_up_to(n-1)
```

- Why does `sum_up_to(10)` return 55? Well `sum_up_to(10)` calls `sum_up_to(9)` which calls `sum_up_to(8)` ... which calls `sum_up_to(0)`. The base case is `sum_up_to(0)`. It returns zero to `sum_up_to(1)` which returns 1 (1+0) to `sum_up_to(2)` which returns 3 (2+1) to `sum_up_to(3)` which returns 6 (3+3) to `sum_up_to(4)` which returns 10 (4+6) to `sum_up_to(5)` which returns 15

(5+10) to `sum_up_to(6)` … which returns 45 (9+36) to `sum_up_to(10)` which returns 55 (10+45) which is our answer.

Recursive factorial

- Factorial 4 or 4! = 4 * 3 * 2 * 1 and in general N! = N * (N-1) * (N-2) * (N-3) * … 2 * 1.
- 1! is defined as 1.
- Let's look at some examples of factorial in action:

```
>>> from recursion import factorial
>>> factorial(1)
1
>>> factorial(2)
2
>>> factorial(3)
6
>>> factorial(4)
24
>>> factorial(5)
120
>>> factorial(6)
720
>>> factorial(7)
5040
>>> factorial(8)
40320
>>> factorial(9)
362880
>>> factorial(10)
3628800
```

- Do you notice anything recursive about the above sequence? Let's annotate each line to make the recursion obvious:

```
>>> from recursion import factorial
>>> factorial(1) # base case: returns 1
1
>>> factorial(2) # returns 2 * factorial(1)
2
>>> factorial(3) # retunrs 3 * factorial(2)
6
>>> factorial(4) # returns 4 * factorial(3)
24
>>> factorial(5) # returns 5 * factorial(4)
120
>>> factorial(6) # returns 6 * factorial(5)
720
>>> factorial(7) # returns 7 * factorial(6)
5040
>>> factorial(8) # returns 8 * factorial(7)
40320
>>> factorial(9) # returns 9 * factorial(8)
362880
>>> factorial(10) # returns 10 * factorial(9)
3628800
```

- For any argument `N`, `factorial(N)` is equal to `N * factorial(N-1)`. This is the essence of a recursive solution. The solution to the problem `factorial(N)` is broken down into the operation `N *` on a simpler version of the same problem `factorial(N-1)`. For example `factorial(10)` is `10 * factorial(9)`. The base case ensures recursion stops at some point. It encodes the fact that `factorial(1)` is 1.

- Let's write the Python code that implements the `factorial()` function:

```
def factorial(n):
    if n == 1: # base case : no more calls to factorial()
        return 1
    return n * factorial(n-1)
```

Our old friend Fibonacci

- The Fibonacci sequence of numbers is given by: 1, 1, 2, 3, 5, 8, 13, etc. The first two numbers of the sequence are both defined to be 1 and thereafter each number in the sequence is defined as the sum of the previous two.

- Let's look at some examples of `fibonacci()` in action:

```
>>> from recursion import fibonacci
>>> fibonacci(0)
1
>>> fibonacci(1)
1
>>> fibonacci(2)
2
>>> fibonacci(3)
3
>>> fibonacci(4)
5
>>> fibonacci(5)
8
>>> fibonacci(6)
13
>>> fibonacci(7)
21
>>> fibonacci(8)
34
>>> fibonacci(9)
55
>>> fibonacci(10)
89
>>> fibonacci(11)
144
>>> fibonacci(12)
233
```

- Do you notice anything recursive about the above sequence? Let's annotate each line to make the recursion obvious:

```
>>> from recursion import fibonacci
>>> fibonacci(0) # base case: returns 1
1
>>> fibonacci(1) # base case: returns 1
1
```

```
>>> fibonacci(2) # fibonacci(1) + fibonacci(0)
2
>>> fibonacci(3) # fibonacci(2) + fibonacci(1)
3
>>> fibonacci(4) # fibonacci(3) + fibonacci(2)
5
>>> fibonacci(5) # fibonacci(4) + fibonacci(3)
8
>>> fibonacci(6) # fibonacci(5) + fibonacci(4)
13
>>> fibonacci(7) # fibonacci(6) + fibonacci(5)
21
>>> fibonacci(8) # fibonacci(7) + fibonacci(6)
34
>>> fibonacci(9) # fibonacci(8) + fibonacci(7)
55
>>> fibonacci(10) # fibonacci(9) + fibonacci(8)
89
>>> fibonacci(11) # fibonacci(10) + fibonacci(9)
144
>>> fibonacci(12) # fibonacci(11) + fibonacci(10)
233
```

- In general, `fibonacci(N) = fibonacci(N-1) + fibonacci(N-2)`. Our base cases are `fibonacci(0) = 1` and `fibonacci(1) = 1`.
- Let's translate this into Python…

Reversing a list

- Let's try to come up with a recursive implementation of a function that reverses a list.
- Let's look at some examples of `reverse_list()` in action:

```
>>> from recursion import reverse_list
>>> reverse_list([])
[]
>>> reverse_list([5])
[5]
>>> reverse_list([4,5])
[5, 4]
>>> reverse_list([3,4,5])
[5, 4, 3]
>>> reverse_list([2,3,4,5])
[5, 4, 3, 2]
>>> reverse_list([1,2,3,4,5])
[5, 4, 3, 2, 1]
```

- Do you notice anything recursive about the above sequence? Let's annotate each line to make the recursion obvious:

```
>>> from recursion import reverse_list
>>> reverse_list([]) # base case : returns []
[]
>>> reverse_list([5]) # returns reverse_list([]).append(5)
[5]
>>> reverse_list([4,5]) # returns reverse_list([5]).append(4)
[5, 4]
>>> reverse_list([3,4,5]) # returns reverse_list([4,5]).append(3)
[5, 4, 3]
```

```
>>> reverse_list([2,3,4,5]) # returns reverse_list([3,4,5]).append(2)
[5, 4, 3, 2]
>>> reverse_list([1,2,3,4,5]) # returns reverse_list([2,3,4,5]).append(1)
[5, 4, 3, 2, 1]
```

- Let’s translate this into Python…