Lecture 9.2 : 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 i.e. *it calls itself*.
- Let's try calling `foo()` and see what happens:

```
print(foo(10))
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Input In [2], in <module>
----> 1 print(foo(10))

Input In [1], in foo(n)
      1 def foo(n):
----> 2    return foo(n-1)

Input In [1], in foo(n)
      1 def foo(n):
----> 2    return foo(n-1)

    [... skipping similar frames: foo at line 2 (2970 times)]

Input In [1], in foo(n)
      1 def foo(n):
----> 2    return foo(n-1)

RecursionError: 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 creates 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:

```
print(bar(10))
```

```
0
```

- Why does `bar(10)` 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:

```
print(sum_up_to(10))
```

```
55
```

- Let's try some more examples:

```
print(sum_up_to(0))
print(sum_up_to(1))
print(sum_up_to(2))
print(sum_up_to(3))
print(sum_up_to(4))
print(sum_up_to(5))
print(sum_up_to(6))
print(sum_up_to(7))
print(sum_up_to(8))
print(sum_up_to(9))
print(sum_up_to(10))
```

```
0
1
3
6
10
15
21
28
36
45
55
```

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

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

- 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.
- Our base case 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)`.
- The base case 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(9)` 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:

```
print(factorial(1))
print(factorial(2))
print(factorial(3))
print(factorial(4))
print(factorial(5))
print(factorial(6))
print(factorial(7))
print(factorial(8))
print(factorial(9))
print(factorial(10))
```

```
1
2
6
24
120
720
5040
40320
362880
3628800
```

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

```
factorial(1)          1        base case returns 1
factorial(2)          2        2 * factorial(1)
factorial(3)          6        3 * factorial(2)
factorial(4)         24        4 * factorial(3)
factorial(5)        120        5 * factorial(4)
factorial(6)        720        6 * factorial(5)
factorial(7)       5040        7 * factorial(6)
factorial(8)      40320        8 * factorial(7)
factorial(9)     362880        9 * factorial(8)
factorial(10)   3628800        10 * factorial(9)
```

- 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.
- The base case 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)
```

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:

```
print(fibonacci(0))
print(fibonacci(1))
print(fibonacci(2))
print(fibonacci(3))
print(fibonacci(4))
print(fibonacci(5))
print(fibonacci(6))
print(fibonacci(7))
print(fibonacci(8))
print(fibonacci(9))
print(fibonacci(10))
```

```
1
1
```

```
2
3
5
8
13
21
34
55
89
```

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

```
fibonacci(0)     1      base case returns 1
fibonacci(1)     1      base case returns 1
fibonacci(2)     2      fibonacci(1) + fibonacci(0)
fibonacci(3)     3      fibonacci(2) + fibonacci(1)
fibonacci(4)     5      fibonacci(3) + fibonacci(2)
fibonacci(5)     8      fibonacci(4) + fibonacci(3)
fibonacci(6)    13      fibonacci(5) + fibonacci(4)
fibonacci(7)    21      fibonacci(6) + fibonacci(5)
fibonacci(8)    34      fibonacci(7) + fibonacci(6)
fibonacci(9)    55      fibonacci(8) + fibonacci(7)
fibonacci(10)   89      fibonacci(9) + fibonacci(8)
```

- 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…

```
def fibonacci(n):
   if n == 0:
      return 1
   if n == 1:
      return 1
   return fibonacci(n-1) + fibonacci(n-2)
```

Reversing a list

- Let's try to come up with a recursive implementation of a function that reverses a list.