CS 115 Lecture 20

Recursion

Taken from notes by Dr. Neil Moore

Recursion

Problems – computational, mathematical, and otherwise – can be defined and solved **recursively.**

- That means, in terms of themselves
- A compound *sentence* is two *sentences* with “and” between them
- A Python expression may be two *expressions* with an operator between them (3 + 2) * (4 – 9)
- A tree is made of branches and *branches* are made of smaller *branches*
- Many mathematical structures are defined recursively
  - Fibonacci numbers, fractals, factorials, …
  - Mathematicians call this **induction** (same thing as recursion)
  - It’s also a common method of mathematical proof

Recursion in programming

- The idea behind recursion in programming
  - Break down a complex problem into a simpler version of *the same problem*
  - Implemented by functions *that call themselves*
    - **Recursive functions**
  - The same computation recurs (happens repeatedly)
    - This is not the same as iteration (looping) – the repetition is not obvious in the code
    - But it is always possible to convert iteration to recursion and vice versa
- Recursion is often the most natural way of thinking about a problem.
  - Some computations are very difficult to perform without recursion

Thinking recursively

- Suppose we want to write a function that prints a triangle of stars

  print_triangle(4)

Gives

```
*
* *
* * *
* * * *
```

- You can use nested loops to solve this, but let's try recursion instead

Thinking recursively

- Pretend someone else has already written a function to print a triangle of size 3. How would you print a triangle of size 4?
  - First call that function
  - Then print a row of four stars
- What about a triangle of size 5?
  - Print a triangle of size 4
  - Then print a row of five stars
- Recursion: use the solution to a simpler version of the same problem!

A (broken) recursive function

```
def print_triangle( side_len ):
     # first solve a simpler version of the problem
     print_triangle(side_len -1)
     # solve the original problem by drawing the last line
     print("* " * side_len)
     print()
```

- One small problem
  - It will never end!
  - To print a triangle of size 1, first print a triangle of size 0
  - To do that, you would have to print a triangle of size -1 – What???

The base case

- Every recursion must end somewhere
  - At some point the problem is so simple we can solve it directly
  - Usually that is when the problem size is zero or one
  - We call this the **base case** or **the termination condition**
  - How do you print a triangle of size zero?
    - By doing nothing!

```
def print_triangle(side_len):
    if side_len > 0:     # recursive case
        print_triangle(side_len -1)
        print("* " * side_len))
        print()
    # the "else" is the base case – do nothing!!  Fall through the if and return
```

Rules for recursion

There are three key requirements for a recursive function to work correctly

1. **Base case:** There **MUST** be a special case to handle the simplest versions of the problem directly, **without** recursion.
   - A base case does NOT call the function again!
2. **Recursive case:** there must be a case where the function DOES call itself.
3. **Simplification:** the recursive call must be performed on a simpler version of the problem. That is, it must reduce the size of the problem, bringing you *closer to the base case*
   - That means the arguments **MUST** be changed from the parameters
   - If this rule is not followed, you have **infinite recursion** and WILL crash eventually!

- A few related guidelines
  - You should check for the base case first
    - Before making any recursive calls
  - The base case is usually, though not always, a problem involving 0 or 1 or something of that size.

About the rules

- You can have multiple base cases, as long as there is at least one.
- Sometimes the base case does nothing! That’s ok!
  - You could put the recursive case in an if statement
    - “If it’s not the base case, then do something”
- If the function returns something, that something should use/involve the value of the recursive call
- The changes you make to the recursive parameters (when they become arguments) can be just about anything:
  - Often subtraction, division, shortening a list
  - But in some situations, it can be addition or multiplication
  - The important thing is that it gets closer to a base case!

About the rules

- The order of recursive calls matters!
  - What would happen if we move the `print_triangle` call so it is after the `print`?
  - The triangle is upside down!
- You can have more than one recursive call inside a recursive function.
  - Just means the function will do a LOT more work before it can return

Infinite recursion

What happens if you break one of the rules?

- You can get an **infinite recursion**
- Meaning the function just keeps calling itself “forever”
- Even worse than an infinite loop!
  - Every recursive call (like every call!) uses a little bit of memory
    - Parameters, return address, return value, local variables, …
  - Where are all these stored? On the call stack!
  - So eventually an infinite recursion will run out of memory
    - At least crashing your program
    - And possibly the whole operating system!

Infinite recursion and Python

Python has built-in checks to avoid crashing the OS with recursion

- When there are too many recursive calls, it raises an exception:

  `RuntimeError ("Maximum recursion depth exceeded…")`

- So the program crashes before the OS does
- You can change the limit with `sys.setrecursionlimit(1000)`
  - But then you risk crashing more than just your program!

Recursive definitions

When solving a problem recursively, it helps to write out the definition of the problem recursively. This is usually the hard part.

Consider the Fibonacci sequence:

Fib(0) = 1, Fib(1) = 1, Fib(2) = 2, Fib(3) = 3, Fib(4) = 5, Fib(5) = 8, …

What's the pattern?

- **Recursive case:** Fib(n) = Fib(n-1) + Fib(n-2)
- **Base case:** actually, there are two! Fib(0) = 1, Fib(1) = 1
- Each recursive call brings us closer to the base cases
  - As long as n isn't negative, anyway

The Fibonacci sequence in code

```
def fibonacci(n):
# base cases
     if n == 0 or n == 1:
          result = 1
     else: # recursive case
          result = fibonacci(n-1) + fibonacci(n-2)
     return result
```

Recursion and the call stack

- Every recursive call adds a new entry to the call stack (just like every function call!)
  - When the call returns, the entry on the stack is removed (just like every return!)
- So you'll have the same function on the call stack many times
  - Each **instance** of the function has its own parameters, local variables, return value and return address
  - Variables are local to **one call** to the function
- Let's observe the call stack in a recursive program using the debugger