12.2 Recursively-Defined Functions

In the last section, we looked at the function $f(n) = \sum_{i=0}^{n} i$, and proved using induction that $f(n) = \frac{n(n+1)}{2}$ for all $n \in \mathbb{N}$. The key insight in the inductive step was that we could break down the summation into one of the same form, but of a slightly smaller size:

$$\sum_{i=0}^{k+1} i = \left(\sum_{i=0}^{k} i\right) + (k+1)$$

We can represent this idea more precisely as the following algebraic statement: for all $k \in \mathbb{N}$, $f(k+1) = f(k) + (k+1)$. This relationship actually gives uas a different way of *defining* f, without using either a summation or the formula $\frac{n(n+1)}{2}$. Here is our definition of f:

$$f(n) = \begin{cases} 0, & \text{if } n = 0 \\ f(n-1) + n, & \text{if } n > 0 \end{cases}$$

We say that this is a **recursive** definition, meaning that f is defined in terms of itself.[1] It is

[1] Another name for this is a *self-referential definition*.

this recursive definition that formed the basis of our induction proof in the previous section: we were able to manipulate the equation $f(k+1) = f(k) + (k+1)$ to prove the inductive step.

Recursively-defined functions in Python

Much earlier in these notes, we discussed how we can turn mathematical expressions involving summations into Python expressions using the built-in `sum` function. It turns out that we can do the same with recursively-defined functions, and that this translation is a very natural one. Here is how we could represent our above function f in Python:

```
def f(n: int) -> int:
    """Return the sum of the numbers between 0 and n, inclusive.

    Preconditions:
        - n >= 0

    >>> f(4)
    10
    """
    if n == 0:
        return 0
```

```
    else:
        return f(n - 1) + n
```

Now, this implementation certainly isn't the most efficient way of representing this function,[2] but it does illustrate that Python functions can be also defined recursively.

[2] e.g., `return n * (n + 1) // 2`

More formally, let `f` be a Python function. We say that `f` is a **recursively-defined function** (or **recursive function** for short) when it contains a call to itself in its body. We use the term **recursive call** to describe this inner `f(n - 1)` call. We use the term *recursion* to describe the programming technique of defining recursive functions to perform computations and solve problems.

We use labels for the different parts of this if statement, as the structure is common for most recursive functions:

- The if branch, consisting of the statement `return 0`, is called the **base case** of the function.
- The else branch, consisting of the statement `return f(n - 1) + n`, is called the **recursive step** of the function, since it contains a recursive call.[3]

[3] We'll see later that recursive functions can have more than one base case and recursive case.

These labels are deliberately parallel to the ones we used for a proof by induction in the previous section. In both an induction proof and recursive function, the *base case* is the component that does not require any additional "breaking down" of the problem. Similarly, both the *inductive step* of a proof and the *recursive step* of a function require the problem to be broken down into an instance of a smaller size, either by using the induction hypothesis or by making a recursive call.

How does recursion "work"?

Just like induction, recursion can feel a bit like magic when we first see it. We tend to reason about our code step by step, line by line. In the case of a recursive call—`f(n - 1)` in the above example—you might wonder, how can we call a function when we haven't finished defining it yet?

There are two ways of reasoning about recursive functions. We'll start with the one that follows what the Python interpreter does, which is analogous to the "domino-style chain of reasoning" for induction:

- When we call `f(0)`, the base case executes, and `0` is returned.
- When we call `f(1)`, the recursive call `f(1 - 1) == f(0)` is made, which returns `0` (see previous point). Then `0 + 1 == 1` is returned.
- When we call `f(2)`, the recursive call `f(2 - 1) == f(1)` is made, which returns `1` (see previous point). Then `1 + 2 == 3` is returned.

In a proof by induction, we found that the induction step meant that $P(0)$ implied $P(1)$, which implied $P(2)$, which implied $P(3)$, etc. In the same way for our recursive function, `f(2)` calls `f(1)`, which calls `f(0)`, etc.

However, this "chain of reasoning" is a very long-winded way of tracing the execution of a recursive function call! Suppose we want to reason about the call `f(100)`: we'd be sitting here long time writing statements of the form "When we call `f(_)`, the recursive call `f(_)` …" as above. This type of literal tracing is what the Python interpreter does, but it's also *extremely time-consuming and error-prone* for humans to do.

The second (and better) way for humans to reason about recursive functions is the *inductive approach*: we assume that the recursive call `f(n - 1)` returns the correct result, based on the function's specification, and without relying on having explicitly traced that call. This is the exact analog of the induction hypothesis in a proof by induction!

For example, here's how we we would reason inductively about the call `f(100)`:

1. When we call `f(100)`, the the recursive call `f(100 - 1) == f(99)` is made. *Assuming* this call is correct, it returns `4950` (the sum of the numbers between 0 and 99, inclusive).
2. Then `4950 + 100 == 5050` is returned.

If you're feeling a bit uncomfortable with making the assumption that `f(99)` returns `4950`, think back to our proof by induction. We assumed $P(k)$ and used that to prove $P(k+1)$, knowing that as long as k was arbitrary we would be able to fall back on our "chain of reasoning" to get to the base case. In the same way here, we can assume `f(99)` is correct to justify the correctness of `f(100)`, knowing that if we wanted to we *could* trace all the different calls `f(99)`, `f(98)`, …, `f(1)`, `f(0)`, but would end up with the same result.

This inductive approach technique is also called *partial tracing*, where the "partial" indicates that we do not trace into any recursive calls, but instead assume that they work correctly. This is also the difference between the `next` and `step` commands in the Python debugger: we use the former to execute a function call as one step, and the latter to actually go into the body of the function being called.

The Euclidean Algorithm revisited

Recall that all the way back in Section 6.6, we studied the *Euclidean algorithm* for calculating the greatest common divisor of two numbers. This relied on the mathematical fact that for all $a, b \in \mathbb{Z}$ with $b \neq 0$, $\gcd(a, b) = \gcd(b, a \ \% \ b)$. Here is our implementation of the Euclidean algorithm, using a while loop:

```
def euclidean_gcd(a: int, b: int) -> int:
    """Return the gcd of a and b.

    Preconditions:
```

```
        - a >= 0 and b >= 0
    """
    x = a
    y = b
    while y != 0:
        r = x % y
        x = y
        y = r
    return x
```

There is another way of implementing the Euclidean algorithm in Python that uses recursion. Take another look at the key mathematical equation that makes the Euclidean algorithm work: $\gcd(a, b) = \gcd(b, a \% b)$. This equation, plus the fact that $\gcd(a, 0) = a$ for all $a \in \mathbb{N}$ is actually all we need to write an alternate, recursive definition of the gcd function over the natural numbers:

$$\gcd(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \% b), & \text{if } b > 0 \end{cases}$$

Compare this definition to the one we wrote at the start of this section. This is also a recursive definition, but now the recursive part doesn't just decrease a number by 1. Instead, the second argument decreases from b to $a \% b$—and the value of the latter could be anything between 0 and $b - 1$, inclusive. This example shows the flexibility of recursive definitions: we aren't just limited to going "from n to $n - 1$", or even unary functions. A recursive definition is valid as long as it always uses "smaller" argument values to the function in the recursive call.[4]

[4] We'll define what we mean by "smaller" more precisely in a future section.

With this in mind, we'll complete our example by translating this gcd definition directly into Python code. Notice once again how naturally the above definition translates into Python:

```
def euclidean_gcd_rec(a: int, b: int) -> int:
    """Return the gcd of a and b (using recursion!).

    Preconditions:
        - a >= 0 and b >= 0
    """
    if b == 0:
        return a
    else:
        return euclidean_gcd_rec(b, a % b)
```