7.1 Introduction to Number Theory

We've spent the first six chapters of these notes studying programming in Python. We've been mainly focused on how we represent data and designing functions to operate on this data. Up to this point, the *ideas* behind the functions that we've written have been relatively straightforward, and the challenge has been in implementing these ideas correctly using appropriate data types and programming language features like comprehensions, if statements, and for loops. We are now ready to move onto algorithms where the ideas themselves will be more complex. It won't be "obvious" how or why these algorithms work, and so to convince ourselves that these algorithms are correct, we'll need to study the mathematical theory that underlie them.

You may recall that we previewed how mathematical *proof* could be used to develop algorithms in the latter half of Chapter 4. One notable example 4.7 Proofs and Programming II: Prime Numbers, in which we used mathematical proof to justify the correctness of a faster version of a function that checks whether an integer is prime:

```
from math import floor, sqrt


def is_prime_v2(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(2, floor(sqrt(p)) + 1)
    return (
        p > 1 and
        all({not divides(d, p) for d in possible_divisors}
    )
```

It turns out that **number theory**, the branch of mathematics concerned with properties of integers such as divisibility and primality, is a wonderful educational domain for us for two reasons. First, many definitions in number theory (like divisibility) are familiar and/or intuitive, and so developing proofs and algorithms in this domain doesn't require significant mathematical build up. Second, our study of number theory and its applications to computer science will lead us to understand–and be able to implement for ourselves—the **RSA cryptosystem**, consisting of a pair of algorithms that are central to modern Internet security. If you've never heard of RSA, cryptosystems, or ever thought about security, don't worry! Over the next two chapters we're going to build of all of the mathematical theory and algorithms from scratch, without expecting any prior knowledge of the subject matter. What will set this work apart from what we've done so far is that to understand what these algorithms do and why they work, we'll need to step away from code and into the realm of mathematics.

We'll start our journey into number theory with a few key definitions, some of which you've seen before defined formally in this course, and others that you might have heard about before, but not seen a formal definition.

Divisibility, primality, and the greatest common divisor

Here are our first two definitions; these are repeated from Chapter 4.

Definition. Let $n, d \in \mathbb{Z}$. We say that d **divides** n when there exists a $k \in \mathbb{Z}$ such that $n = dk$. We use the notation $d \mid n$ to represent the statement "d divides n".

The following phrases are synonymous with "d divides n":

- n **is divisible by** d
- d is a **factor** of n
- n is a **multiple** of d

As a mathematical predicate in formal logic:

$$d \mid n : \mathbb{Z} \exists k \in \mathbb{Z},\ n = dk \qquad \text{where } n, d \in \mathbb{Z}$$

Definition. Let $p \in \mathbb{Z}$. We say p is **prime** when it is greater than 1 and the only natural numbers that divide it are 1 and itself.

As a mathematical predicate in formal logic:

$$\mathit{IsPrime}(p) : p > 1 \wedge \left(\forall d \in \mathbb{N},\ d \mid p \Rightarrow d = 1 \vee d = p\right), \qquad \text{where } p \in \mathbb{Z}$$

The next few definitions introduce and expand on the notion of common divisors between two numbers.

Definition. Let $m, n, d \in \mathbb{Z}$. We say that d is a **common divisor** of m and n when d divides m and d divides n.

We say that d is the **greatest common divisor** of m and n when it the largest number that is a common divisor of x and y, or 0 when m and n are both 0.[1] We can define the function $\gcd : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$ as the function which takes numbers m and n, and returns their greatest common divisor.

[1] According to this definition, what is $\gcd(0, n)$ when $n > 0$?

For example, $\gcd(10, 4) = 2$ and $\gcd(-30, 18) = 6$.

You might wonder whether this definition makes sense in all cases: is it possible for two numbers to have no divisors in common? One proprerty of divisibility is that 1 divides every integer (we'll prove this at the bottom of this section). So at the very least, 1 is a common divisor between any two natural numbers. We give a name to the special case when 1 is the *only* positive divisor between two numbers.

Definition. Let $m, n \in \mathbb{Z}$. We say that m and n are **coprime** when $\gcd(m, n) = 1$.

Quotients, remainders, and modular arithmetic

In 4.6 Proofs and Programming I: Divisibility, we introduced one of the central theorems of number theory, called the *Quotient-Remainder Theorem*.

Theorem. (Quotient-Remainder Theorem) For all $n \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$, there exist $q \in \mathbb{Z}$ and $r \in \mathbb{N}$ such that $n = qd + r$ and $0 \leq r < d$. Moreover, these q and r are *unique* for a given n and d.

We say that q is the **quotient** when n is divided by d, and that r is the **remainder** when n is divided by d.[2]

[2] Recall that in Python, we can compute the quotient using `//` and the remainder using `%`. You can compute both at the same time using the built-in function `divmod`, which returns a tuple containing the quotient and remainder. Try it!

Often when we are dealing with relationships between numbers, divisibility is too coarse a relationship: as a predicate, it is constrained by the binary nature of its output. Instead, we often care about the *remainder* when we divide a number by another. The following definition and notation gives us an elegant to compare remainders.

Definition. Let $a, b, n \in \mathbb{Z}$ and assume $n \neq 0$. We say that a is **equivalent to** b **modulo** n when $n \mid a - b$. In this case, we write $a \equiv b \pmod{n}$.

So for example, $10 \equiv 2 \pmod{4}$ and $9 \equiv -111 \pmod{5}$.

There are two related reasons why this notation is so useful in number theory. The first is that modular equivalence can be used to divide up numbers based on their remainders when divided by n:

Theorem. Let $a, b, n \in \mathbb{Z}$ with $n \neq 0$. Then $a \equiv b \pmod{n}$ if and only if a and b have the same remainder when divided by n.

To use our above examples, 10 and 2 have the same remainder when divided by 4, and 9 and -111 have the same remainder when divided by 5. We can represent this theorem symbolically as follows:

$$\forall a, b, n \in \mathbb{Z},\ n \neq 0 \Rightarrow \left(a \equiv b \pmod{n}\right) \Leftrightarrow \left(a \,\%\, n = b \,\%\, n\right)$$

$

One warning: students often confuse the notation $a \equiv b \pmod{n}$ with the % operator. The former states a relationship between three integers (a, b, n) and is fundamentally a predicate, analogous to $=$. The latter is a mathematical operator that *returns* an integer, and is analogous to $+$ or $\times$.

The second reason modular equivalent is useful is that almost all of the "standard" intuitions we have about equality transfer over this new notation, making it pretty easy to work with right at the very start.

Theorem. Let $a, b, c, n \in \mathbb{Z}$ with $n \neq 0$. Then the following hold:

1. $a \equiv a \pmod{n}$.
2. If $a \equiv b \pmod{n}$ then $b \equiv a \pmod{n}$.
3. If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ then $a \equiv c \pmod{n}$.

Theorem. Let $a, b, c, d, n \in \mathbb{Z}$ with $n \neq 0$. If $a \equiv c \pmod{n}$ and $b \equiv d \pmod{n}$, then the following hold:

1. $a + b \equiv c + d \pmod{n}$.
2. $a - b \equiv c - d \pmod{n}$.
3. $a \times b \equiv c \times d \pmod{n}$.

Note that this second theorem shows that the familiar addition, subtraction, and multiplication operations preserve modular equivalence relationships. However, as we'll study further in this chapter, this is *not* the case with division!