Duels and Duals, Mathematical Abstraction and Programming, 2023 Mississippi Governor's School

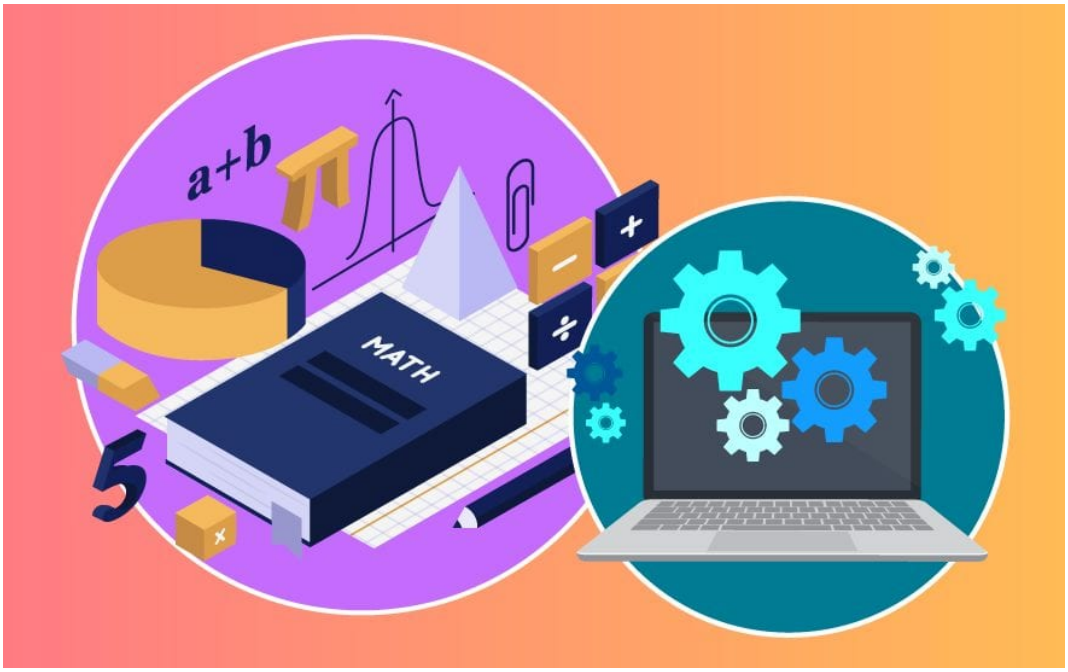
Dr. Jim Newton

June 4, 2023

Contents

Chapter 0

Introduction



https://leverageedu.com/blog/mathematics-and-computing/

0.1 Objectives

There are several objectives in this course:

Mathematics: Ameliorate your love for Mathematics.

Computer Science: Develop a sense for programming.

Communication: Confidently defend your ideas.

Why math? We assume that you already love math, and we will use examples in abstract math as source of ideas to write computer programs.

The mathematical and computer science topics have been chosen for this course to complement each other. You will enforce what you've learned by writing computer programs using the ideas and also by explaining to your peers what it is that you have done.

It is important that the leaders of tomorrow, especially the leaders in the technology sector, understand the impact of technological development on humans. We must know how to apply abstract concepts to real computation problems, and how to explain their ideas and defend their decisions to their peers.

Don't forget that you are working not only with machines but also with humans. In this course, we will maintain a respectful, inclusive, and safe environment. Remember that different people have different strengths and weaknesses. Scientific discussions can sometimes become heated and experts might disagree about the best course of action. We will keep disagreements objective and use science to test ideas, rather than relying on emotions.

Good ideas that are never explained die and are forgotten. It is my belief that you don't understand something until you can explain it. From time to time, some students will present their solutions to their peers. This means defending your choices, recognizing strengths and shortcomings, and being subjected to peer review (important in the scientific process). This exercise is intended not only to learn to accept constructive criticism but also to build confidence.

0.2 Note from your Instructor

I'm Dr. Jim Newton, and I will be your instructor and coach for two weeks as explore mathematical applications in programming.

I am a Mississippi native, and a graduate of Mississippi State University where in 1988 I received a BS in Electrical Engineering and in 1992 an MA in Mathematics. Years later, in 2015, I received my PhD in Computer Science from Sorbonne University in Paris, France. I currently live in Issy les Moulineaux, France, and work as an assistant professor and researcher at EPITA (`www.epita.fr`), a French engineering school.

In 1981, I was a scholar at the first ever Mississippi Governor's School. I've designed this course remembering my own experience at MGS, back in the day. I realize that different students coming from different high schools in Mississippi have been exposed to various levels of mathematics and computer science courses. I hope that regardless of your background and previous experience, you will find this course interesting and challenging. I am confident that if you apply yourself and remain self-confident, you will succeed and learn a lot.

0.3 Overview

We will probably proceed through several units which mix three important experiences:

- Theory — Learn mathematics and computer science concepts.
- Practice — Create code relating to the theory of various units.
- Presentation — Show and explain what you have done.

We will primarily cover an introduction to Abstract Algebra. Our practical treatment of Algebra will help you prepare for a more theoretical treatment at the university level.

0.4 Proposed Syllabus

Week	Day	Unit	Activities
1	Mon	0, 1	Hello World on the Cloud
1	Tue	2	Sets and Functions
1	Wed	2	
1	Thu	3	Logic and Loops
1	Fri	3	
2	Mon	3	Performance, refactoring
2	Tue	4,5	Monoids
2	Wed	4,5	
2	Thu	6,7	Rings and Fields
2	Fri	6,7	

Chapter 1

Environment

https://woz-u.com/wp-content

1.1 GitHub

In this course we will be experimenting with software together. You will be able to share your code with your classmates, and with the world. However, you control when you want to make your code available to the world, and when you want to incorporate your classmate's code into your development area.

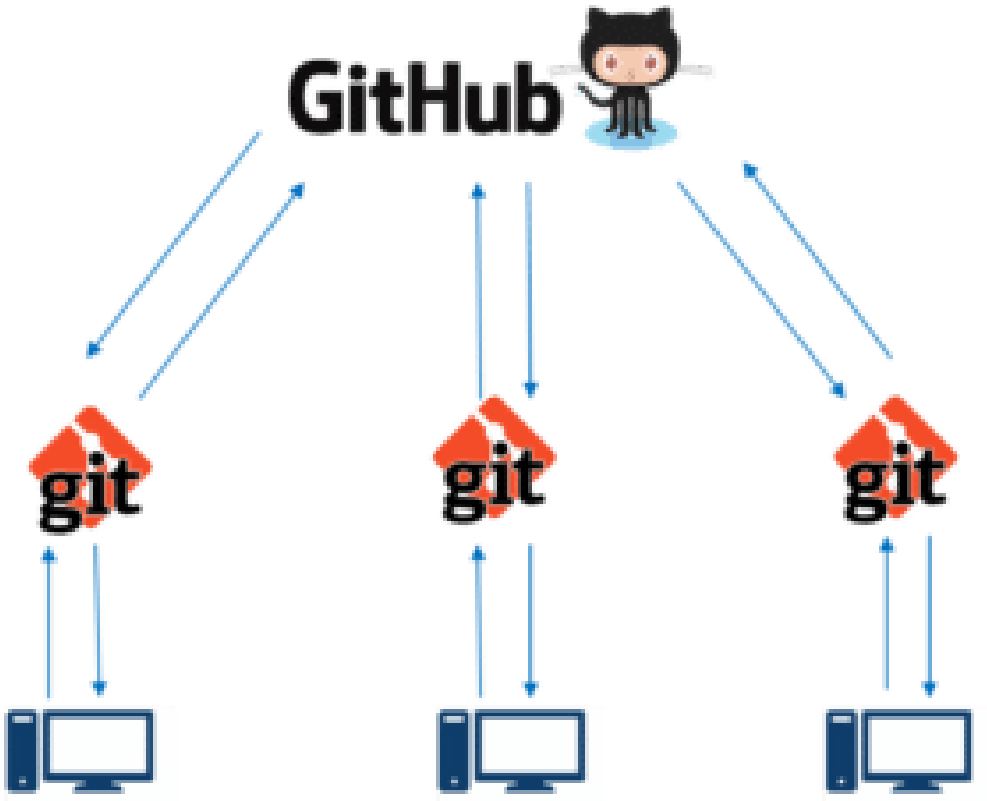


Figure 1.1: GitHub is a cloud service that hosts code repositories. You connect to GitHub either by web browser or directly from *git*.

To achieve this goal we have a central repository of code, hosted by GitHub on the cloud. You will download (clone) a copy of this repository, edit existing code, create new code, and then occasionally upload your contributions back to the central repository, making your contributions available to everyone else.

GitHub is a web based service which hosts thousands of projects, small and large, in the cloud. Typically, users have a development en-

vironment on their local computer (laptop or desktop) and periodically upload and download changes to the cloud.

The URL `https://github.com/jimka2001/mgs-2023`, is the public entry point to the GitHub project which we will work on in this MGS-2023 course. How we will edit code in the GitHub project is described in Section 1.2.

You must create an account on GitHub (if you don't have one already). This will be done in Section 1.3.

1.2 GitPod

Figure 1.2: GitPod provides a web-browser-based development environment which integrates with GitHub.

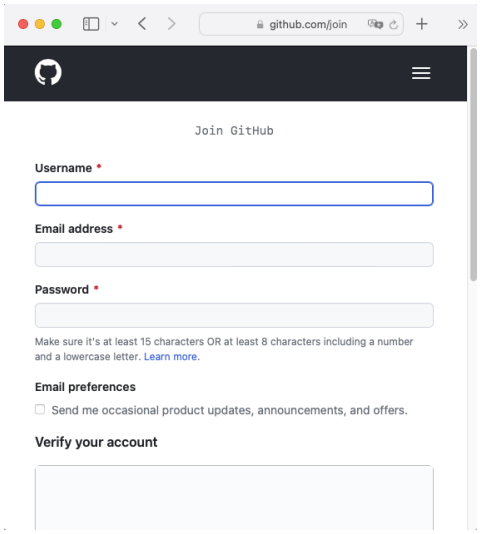
We will use a development environment entirely on the cloud— a web based service called, GitPod to provide an environment for working with GitHub. This environment and all the code you develop will still be available to you after MGS-2023 is finished.

The GitPod web site is `https://www.gitpod.io`.

1.3 Getting Started

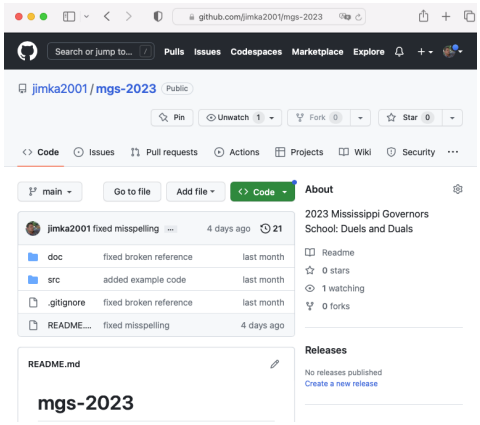
We need to set up the development environment. Before you can learn to use a piece of software, you have to install it. These steps should help.

1.3.1 Account Creation



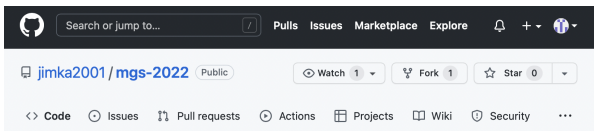
Create a GitHub account using an abstract user name. Don't use your real name. This is self explanatory at the URL `https://github.com/join`.

1.3.2 Open the Repository



Open `https://github.com/jimka2001/mgs-2023` with your web browser.

1.3.3 Fork yourself a Copy

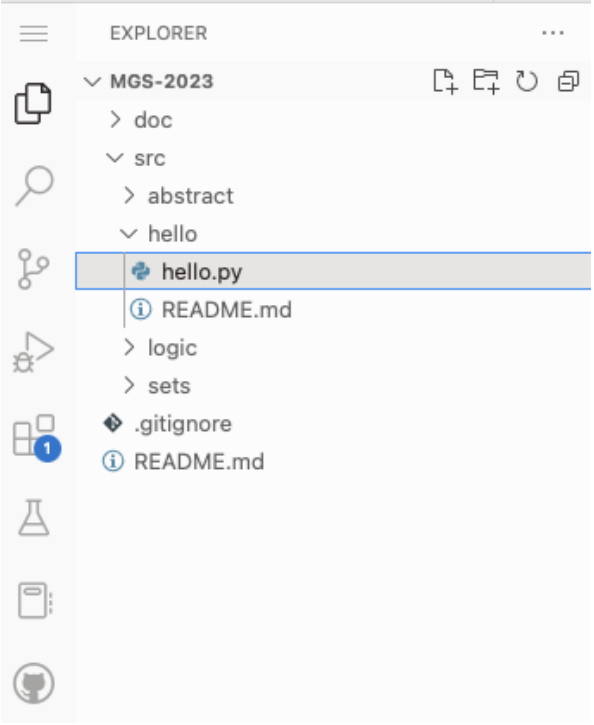


Fork the repository.

1.3.4 Open GitPod

Open the web-IDE, GitPod. You may just prepend `http://gitpod.io/#` to the URL in the web browser.

1.3.5 Open the Explorer

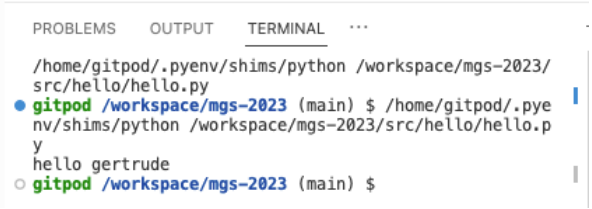


Find the Explorer. Open it to find `hello.py` at `src/hello/hello.py`.

1.3.6 Open a Python File

Click `hello.py` to open the file in an editor pane. You should see something like what is shown here:

1.3.7 Make a Sample Run



Find the definition `hello`. Find the icon  in the top-right of the editor window. Click the triangle.

1.3.8 Challenges for the Student

Understanding errors and debugging is difficult, but it is part of programming.

Experiment with the simple pieces of code in the above sections. Insert spaces and press the run button. Look at the error messages produced. Remove some quotation marks or parenthese (leaving unbalanced quotations marks or parentheses)—again look at what error messages you see when you try to run invalid code.

- remove and add some spaces at the beginning of a line
- change the indentation
- unbalance the parentheses
- unbalance the quotation marks
- put extra spaces inside the quotation marks
- change the name of the `hello` function at definition site or call site.
- figure out how to undo your changes in the test editor to make the code work again.

Question: What is the difference between a synax error and a logical error?

Chapter 2

Mathematical Sets and Functions

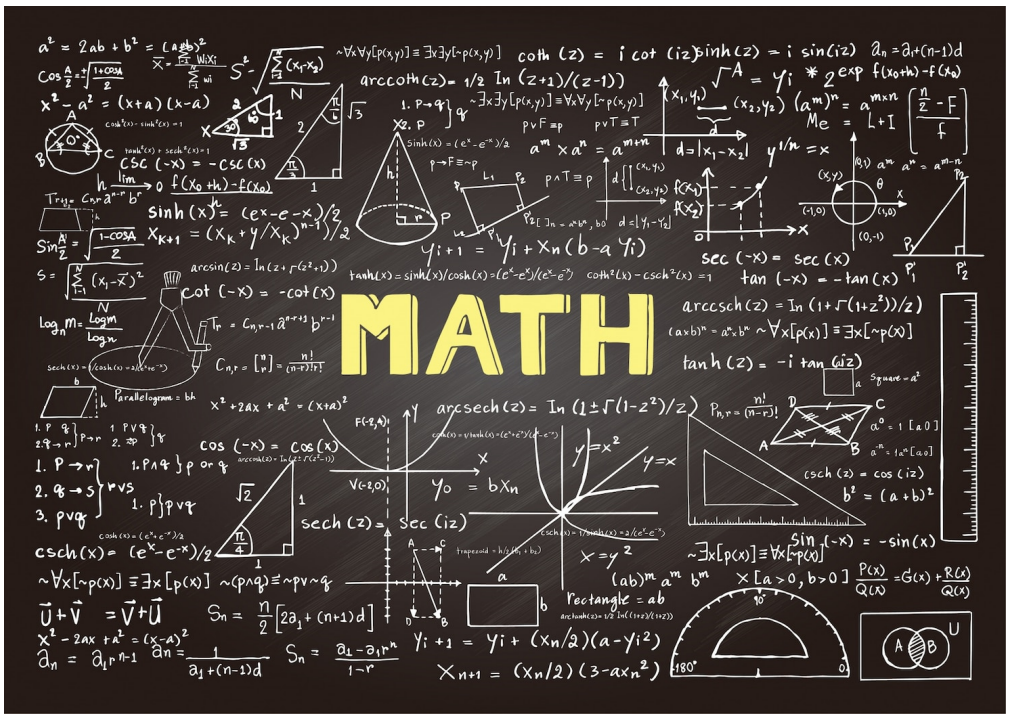
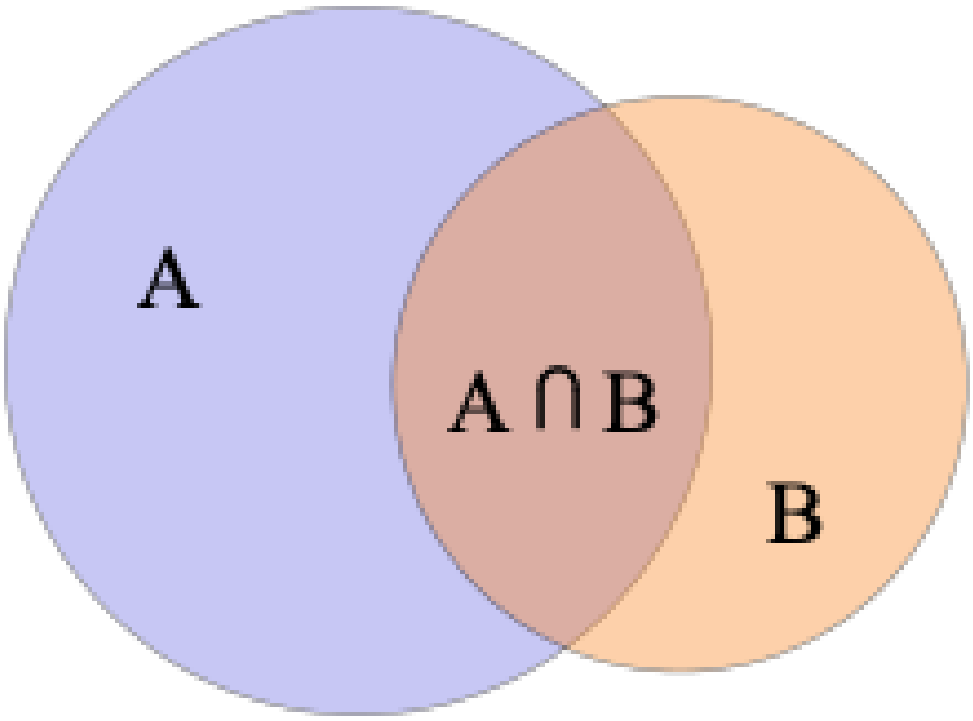


Figure 2.1: https://www.freepik.com



By Svjo - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=20749188

In this unit we will look at functions both from a mathematical perspective and also from a programming perspective. There are some important similarities and some important differences.

We would like to talk about functions. But where should we start the conversation?

You may already have some intuition about functions. Sometimes functions have names, like sin, cos, and log. Sometimes functions lack names but have formulas such as $\frac{x^2-1}{x^2+2x+1}$.

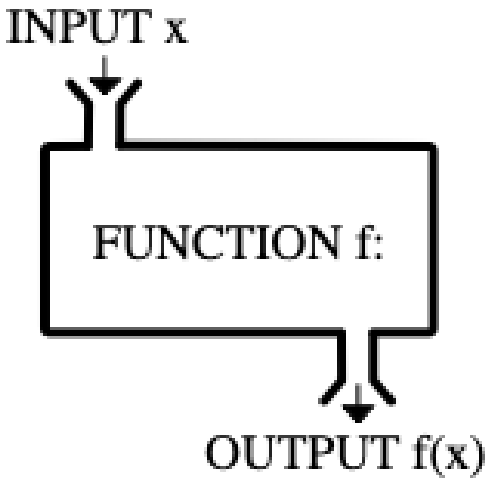
2.1 Set Operations

We avoid the problem of defining exactly what a set is; see Russell's paradox. We will instead concentrate on operations we can perform on sets.

- is-element-of: $x \in A$

- is-subset-of: $A \subset B$
- empty-set: $\emptyset$
- union: $A \cup B$
- intersection: $A \cap B$
- complement: $A \setminus B$ or $\overline{B}$
- De Morgan's theorem: $\overline{A \cup B} = \overline{A} \cap \overline{B}$, or dually $\overline{A \cap B} = \overline{A} \cup \overline{B}$.

2.2 Functions



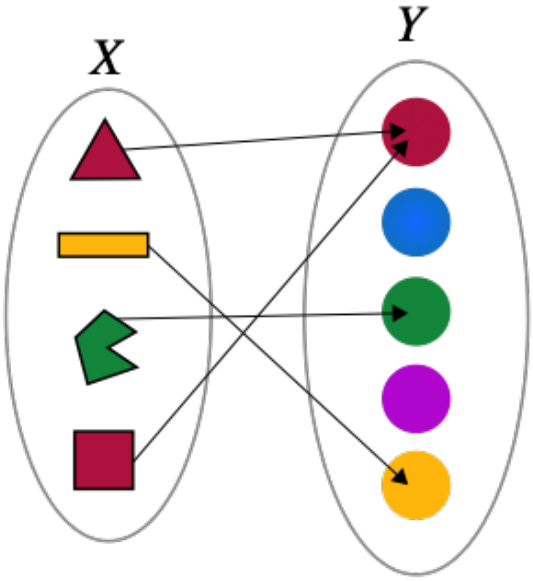
What is a function? A function such as

$$f : X \to Y$$

is a correspondence between two sets. However, it is a special correspondence.

In particular the function, f, associates a unique, well-determined, element of Y with each element of X. More precisely, if $x \in X$, then $f(x)$ designates a unique, well-determined, element of Y.

$$x \in X \implies f(x) \in Y\,.$$



A function is a well-defined correspondence of all the elements of one set (the domain) with some of the elements of another set (the range).

Conceptually, a function has three parts

1. A *domain*, i.e., a place from which input values may be taken.
2. A *range*, i.e., places where output values are found.
3. A *rule*, i.e., a way of determining the output value given only the input value.

2.3 Mathematical notation

$$f(x) = 3x + 1 \tag{2.1}$$

Let's start with a very simple function (2.1). Actually, the only thing specified here is the rule. This function is written in a mathematical notation where the domain and range are not specified. The same notation can be used for different choices of domain and range. For example, this function works for integers, $\mathbb{Z}$, as input, but it will also work for natural numbers, $\mathbb{N}$, rational numbers, $\mathbb{Q}$, real numbers, $\mathbb{R}$, and complex numbers, $\mathbb{C}$. The same function will also work for matrices or for functions themselves.

If we want to be more precise, we can use a more descriptive notation as in (2.2)

$$f : \mathbb{N} \to \mathbb{N} \text{ by } f(x) = 3x + 1 \tag{2.2}$$

The domain and range of a function need not be the same, and need not be a set of simple numbers. For example, we use the symbol $\mathbb{R}^2$ to denote the set of ordered pairs of real numbers; *i.e.*,

$$\mathbb{R}^2 = \{(x, y) \mid x \in \mathbb{R} \wedge y \in \mathbb{R}\}.$$

Equation (2.3) shows an example of such a function.

$$f : \mathbb{R}^2 \to \mathbb{R} \text{ by } f(x, y) = 3x - 2y + 1 \tag{2.3}$$

We may also specify functions using cases, like in Equation (2.4)

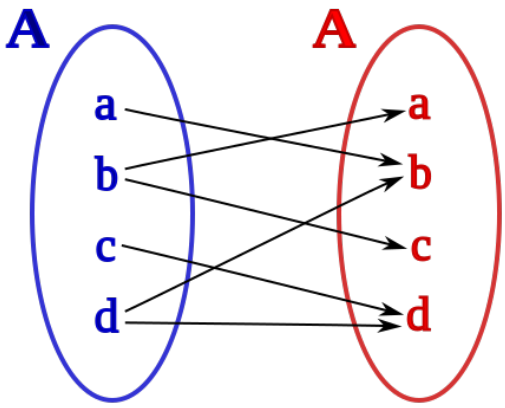
$$|x| = \begin{cases} x & ;\text{if } x > 0 \\ 0 & ;\text{if } x = 0 \\ -x & ;\text{if } x < 0 \end{cases} \tag{2.4}$$

2.4 What a function is not

A correspondence which associates multiple elements of B with an element of A is not a function from A to B. However the range of a function (or the domain for that matter) may be any set, for example we may talk about set-valued functions like a function whose value for each input is some subset of a given set.

Consider the function which takes a finite set as input, and outputs the size of the set, *i.e.*, the number of elements in the set.

Consider the function that takes a whole number, n, as input and returns the set of integers between 0 and n as output.



A function may map several inputs to the same output, but it may not map an input to multiple outputs.

A function must also be well-defined. *I.e.*, the value of the function for a given input must be a fixed value. The value may be difficult to compute, or perhaps even unknown, but it must be some particular value. A function cannot, for example, return a random integer for each input.

2.5 Functions: Programmatic Representation

To implement a function in the Python programming language we:

- specify the function name, such as `square`
- specify zero or more input parameters, such as `(x)`
- compute output value: `return x*x`
- follow the correct syntax

Input values come from the *domain*; output values come from the *range*.

```
1  def square(x):
2    return x * x
```

2.6 Specifying functions by recurrence

Functions can be defined in terms of themselves. We might use an intuitive definition such as Equation (2.5).

$$x^n = \underbrace{x \times x \times \ldots \times x}_{n \text{ times}} . \tag{2.5}$$

We can define the same function more explicitly by using recursion as in Equation (2.6).

$$x^n = \begin{cases} 1 & ; \text{if } n = 0 \\ x \times x^{n-1} & ; \text{if } n > 0 \\ \frac{1}{x^{-n}} & ; \text{if } n < 0 \text{ and } x \neq 0 \end{cases} \tag{2.6}$$

Similarly, we can define the factorial as

$$n! = 1 \times 2 \times \ldots \times n \tag{2.7}$$

or we can define it as

$$n! = \begin{cases} 1 & ; \text{if } n = 0 \\ n \times (n-1)! & ; \text{if } n > 0 \end{cases} \tag{2.8}$$

The first case need not be $n = 0$. We can define the Fibonacci numbers. Let $F(n)$ denote the nth Fibonacci number.

$$F(n) = \begin{cases} 1 & ; \text{if } n = 1 \\ 1 & ; \text{if } n = 2 \\ F(n-1) + F(n-2) & ; \text{if } n > 2 \end{cases} \tag{2.9}$$

Chapter 3

Programming with Functions

Image by Freepik, `https://www.freepik.com/free-vector`

3.1 Programming constructs

In this section we look at several constructions which are essential to your programming experience.

3.1.1 Variables

A variable is an identifier used to hold a value. In the Python language the value of a variable can either be constant or it can change depending on the instructions given by the programmer.

Expressions can contain variables, constants, and function calls (which we'll see in Section 3.1.2).

```
a = 12
b = -7
print(a*b - 3*a)

a = 13
print(a*b - 3*a)
```

An assignment to a variable can also contain the same variable being modified.

```
a = 12
b = -7
a = a + b
print(a*b - 3*a)
```

Basic arithmetic can be performed using `+`, `-`, `*`, and `**` for plus, minus, times, and exponentiation. For example, 3^4 is coded as `3 ** 4`.

```
1 + 2    # addition
3 - 4    # subtraction
5 * 6    # multiplication
3 ** 4   # exponentiation
```

There are two types of division. Floating point division may be performed as `10 / 5` which evaluates to the floating point number `2.0`, while `10 // 5` evaluates to the integer `2`. An important difference arises when the denominator does not evenly divide the numerator. `20 / 8` evaluates to `2.5`, but `20 // 8` evaluates to `2`, because $20 \div 8 = 2$ with remainder 4. To compute the remainder of $20 \div 8$, use `20 % 8` which evaluates to `4`.

```
20 / 8   # division (as floating point)
20 // 8  # division (as integer, discarding the remainder)
20 % 8   # remainder of division (modulus)
```

3.1.2 Functions

Whereas a variable evaluates to the same value every time you evaluate it, a function's value may be different depending on its input, or the variables it references.

A function, defined with `def`, has several parts: name, parameters (zero or more), body (which may be empty), and an optional `return`.

```
def f(a,b):
  return a*b - 3*a

print(f(12,-7))
```

A function without a `return`, returns the special value `None`. It is often the case that if a function has no `return` then the programmer has forgotten something—it is often a bug in the program.

A function may contain other functions within it. The indentation indicates which lines of code are *within* which function.

```
def f(a):
  def g(b):
    return b + 3

  return a + g(-7)
```

A function may reference variables defined outside itself. A function may reference its parameters, and the parameters of functions it is inside of.

```
def f(a):
  def g(b):
    return b + a

  return a + g(-7)
```

However, to reference variables (other than parameters) you must indicate which variable you mean, because there may be several variables of the same name.

In the next example, in the expression `b+a`, the variables reference the parameters of functions `g` and `f`, respectively.

```
a = 100
b = 12

def f(a):
  def g(b):
    return b + a

  return a + g(-7)
```

In the next example, in the expression `a+b+c`, `a` is global, `b` be is the parameter of function `g`, and `c` is -7 as defined within function `h`.

```
a = 100
b = 12

def h(x):
  c = -7
  def g(b):
    global a
    nonlocal c

    return a+b+c

  return g(c)
```

There are many built-in functions and constants, but you often need to `import` a library. The `abs` function is built-in without importing, but the functions `sin`, `cos`, `log`, as well as the constants π and e (`pi` and `e`) are available in a library called `math`.

```
from math import sin, cos, log, pi, e

sin(45 * pi / 180)
cos(0.0)
log(e ** 2)
```

3.1.3 Conditionals

Conditions are important in every programming language, and Python is no exception. A simple conditional is specified as follows using `if` and `else`. Notice which lines end in a colon, `:` and which do not.

```
if x > 10:
  print(x)
```

A two-branch conditional is possible by specifying a *condition*, *consequent*, and *alternative*.

```
  if x > 10:
    print(x)
  else:
    print(y)
```

A multi-branch condition is also supported using `if`, `elif`, and `else`.

```
  if a < 1:
    print(a)
  elif a < 2:
    print(a-1)
  elif a < 3:
    print(a-2)
  else:
    print(None)
```

3.1.4 Collections: Booleans, tuples, lists, sets

Boolean

We've already seen the integer and float data types, but there several more data types which are important to understand. The `bool` (Boolean) data type consists of the special values `True` and `False`. Operators such as `==` (equivalence test), as well as `>`, `>=`, `<`, and `<=` evaluate to a `bool`.

```
print(1 > 2)
print(1 <= 100)
print(1 == 4)
a = 1
print(a == 1)
```

Boolean values are useful in conditions such as the following example.

```
  a = 0
  if a > 2:
    print(a)

  b = 3
  if a > b:
    print(a+b)
```

Tuples

Tuples are like ordered pairs or ordered triples in math: (x, y) or (x, y, z). Triples are not limited to three elements; you may have as many elements as you need. The Python syntax is easy:

```
a = 0
b = 2
c = 4
d = (a, b, c)
x = (3, 4)
y = (c, d, x)
print(y)
print(y[2])
```

A tuple is read-only—you cannot modify the value stored in a tuple once the tuple has been created. You can read any value out of a tuple, for example `a[2]`. Careful, the first value in the tuple is `a[0]` and the second value is `a[1]`; so indexing is so-called zero-based, not one-based as is customary in mathematics.

Lists

A list is like a tuple except that the values can be modified in place if necessary. A list is specified using the square brackets `[1, 2, 3]`, whereas we use round brackets (called parentheses) for tuples.

```
a = 0
b = 2
c = 4
d = [a, b, c]
x = [3, 4]
y = [c, d, x]
print(y)
print(y[2])
y[2] = 4
print(y[2])
print(y)
```

Sets

A set is like a list except that it contains no duplicate elements, and Python does not maintain any particular order of the elements. Use the curly braces `{1, 2, 1, 3}` to specify a set, and note how Python ignores duplicate elements in a set.

```
a = 0
b = 2
c = 4
d = {a, b, c, a, b, c, c}
print(d)
```

3.1.5 Trees

Lists and tuples are not limited to containing numbers. You may have lists of lists of tuples of lists as deep as you like.

```
a = [1, 2, 5, 7]
b = (10, 20, 50, 70, 100)
c = [a, b, a, a]
d = (a, b, c, [(1, [2, 3, [c, a]])])
print(d)
```

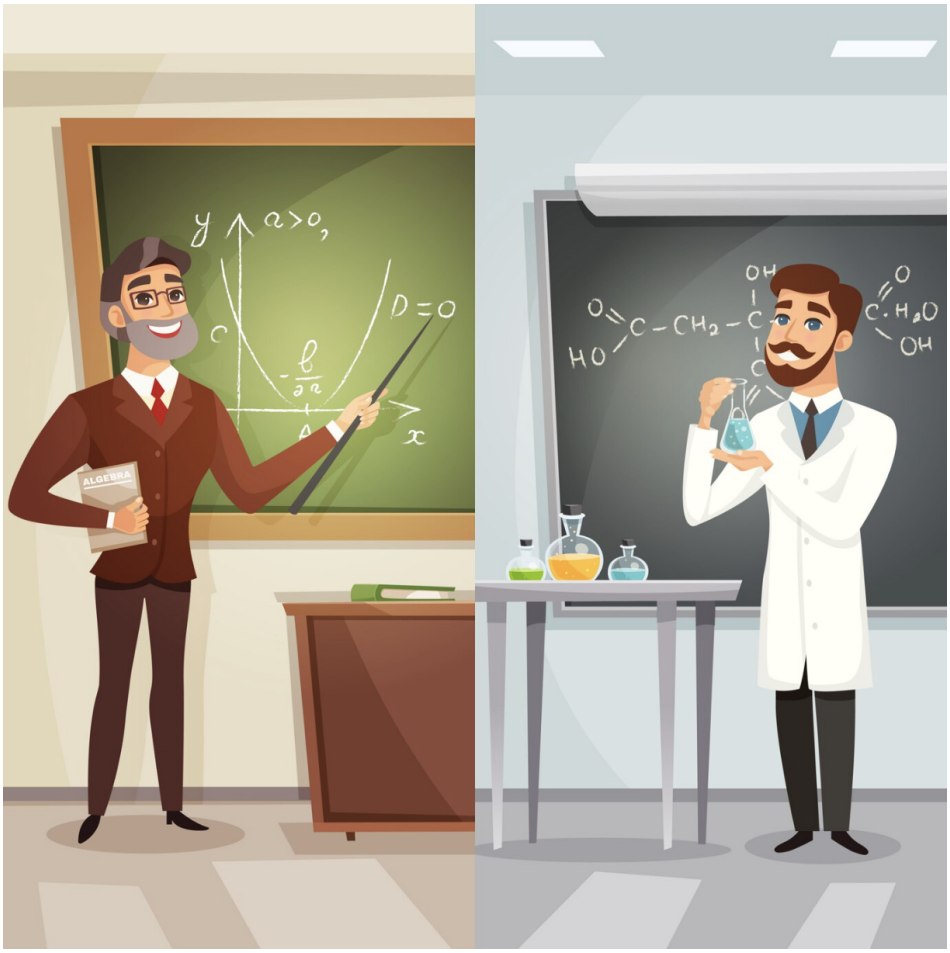
3.1.6 Challenge for the student

If you construct a list containing the same list multiple times, what happens if you modify one of the elements? Evaluate the following code and explain the results.

```
a = [1, 2, 5, 7]
b = [a, a, a, [1, 2, 5, 7]]
b[0] = 100
print(b)
b[1][0] = 200
print(b)
```

We have seen that you can create lists of tuples and tuples of lists. Can you also create sets of lists, sets of tupleslists, tuples of sets, and sets of tuples. Experiment and find out.

3.2 Implement the Quadratic Formula



https://leverageedu.com/blog/mathematics-and-computing/

As a first program in Python you will implement the quadratic formula. recall that if

$$ax^2 + bx + c = 0\,, a \neq 0$$

then

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\,.$$

3.2.1 A Python function for roots of quadratic

Implement the function `quadraticFormula`. The function should implement the mathematical function $x_{1,2}(a, b, c) = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$. The function

computes the roots of the polynomial $ax^2 + bx + c$, assuming $a > 0$, and assuming $b^2 \geq 4ac$.

```
def quadraticFormula(a, b, c):
  from math import sqrt
  # warning, this is a poor implementation,
  # the student should improve this function by refactoring
  return List((-b + sqrt(b*b - 4*a*c))/(2*a),
              (-b - sqrt(b*b - 4*a*c))/(2*a))
```

What are the problems with this implementation? What does it do if there are no real roots? What does it do if the there is only one root.

3.2.2 Challenges for the Student

Implement the following by refactoring the code above.

- Enhance `quadraticFormula`: Notice that the current implementation is inefficient, because it computes almost the same thing twice.
- What happens if $ax^2 + bx + c$ does not have real roots? If you call the function with `a,b,c` what is returned?

  You should refactor the function to define a variable `discriminant` whose value is

  $$\Delta = b^2 - 4ac\,.$$

  Then there are three cases, if $\Delta < 0$, if $\Delta = 0$, and if $\Delta > 0$. Refactor the function so that if there are no real roots it returns an empty list; if there is a unique solution, return a list of one number; and if there are two distinct solutions, return a list of those two solutions **in increasing order**. *I.e.*, if the function returns `[a,b]`, assure that $a < b$.

3.3 Recursive Functions

A recursive function is a function which calls itself.

3.3.1 Recursive implementation of exponentiation

As an example, let's look at a function which computes a^n for some integer or floating point number a, and for some integer $n \geq 0$. We saw a similar function in Equation (2.6) on page 20.

```
def power(a, n):
  if n == 0:
    return 1
  elif n == 1:
    return a
  else:
    return a * power(a, n-1)
```

This function can be extended to handle negative exponents, provided $a \in \mathbb{R}$; *i.e.* if `a` is a floating point number.

```
def power(a, n):
  if n == 0:
    return 1
  elif n < 0:
    return power(1/a, -n)
  elif n == 1:
    return a
  else:
    return a * power(a, n-1)
```

3.3.2 Challenges for the Student

Follow the pattern shown in the `power` function to implement the following functions.

- Implement `String` concatenation using the power model.
- Implement `List` concatenation using the power model.
- Implement `factorial`.

Discussion: In the `factorial` function, do we need a case for $n = 0$ and also $n = 1$? How does this relate to the principle of induction? Look up *Principle of Induction* if you need to review it.

3.4 Fibonacci Sequence

- Implement a recursive function such that given n, the function returns the nth Fibonacci number, F_n.
- Analyze the recursive profile of the Fibonacci generator?
- Suppose that one recursive call requires m bytes, and your computer has a total of M bytes of memory, can you find a bound for the maximum Fibonacci number which is possible to compute?
- Can we use memoization to make the Fibonacci generator faster?
- We may compute any three consecutive Fibonacci numbers as: F_{n-1}, F_n, and F_{n+1},

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

  How can we implement such a multiplication in Python? Hints:

  1. How can we represent a 2×2 matrix programmatically?
  2. How can we multiply two matrices?
  3. How can we compute $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ with approximately $\log_2 n$ matrix multiplications?

- Search on the internet and find the Binet's formula for computing the nth Fibonacci number. Implement this function and compare the results to the recursive implementation.
- **Making code faster** Now you have a way which you suspect is more efficient (faster). Devise an experiment to determine which of the approaches is faster in terms of execution time? Hint, you may need to construct a loop to do the same computation 100 times or 1000 times, and then divide the total time by 100 or 1000 to compute the average time per computation.

3.5 Binary Search

A binary search algorithm is a *divide and conquer* algorithm. Suppose we know that some *event* occurs at a number between x_{left} and x_{right}. For example, suppose $f(x) = 0$ for some $x_{left} < x < x_{right}$. Then if we take $x_{mid} = \frac{x_{left}+x_{right}}{2}$, then at least one of the following is true.

$$f(x) = 0 \qquad \text{for some } x_{left} < x \leq x_{mid} \tag{3.1}$$

$$f(x) = 0 \qquad \text{for some } x_{mid} < x < x_{right} \tag{3.2}$$

This means if we split the interval (x_{left}, x_{right}) in half, the *event* occurs either in the left half or the right half. In such a case, we can write a (recursive) function which divides the interval in half, and *recurs* either on the right half or left half. We end the recursion when we have found the event.

If we don't find the exact value we are searching for, then we must have a termination condition to avoid looping forever. Typically the way this is done is to recur until the interval is sufficiently small: i.e., until $x_{right} - x_{left} < \varepsilon$.

3.6 Roots of a Cubic Polynomial

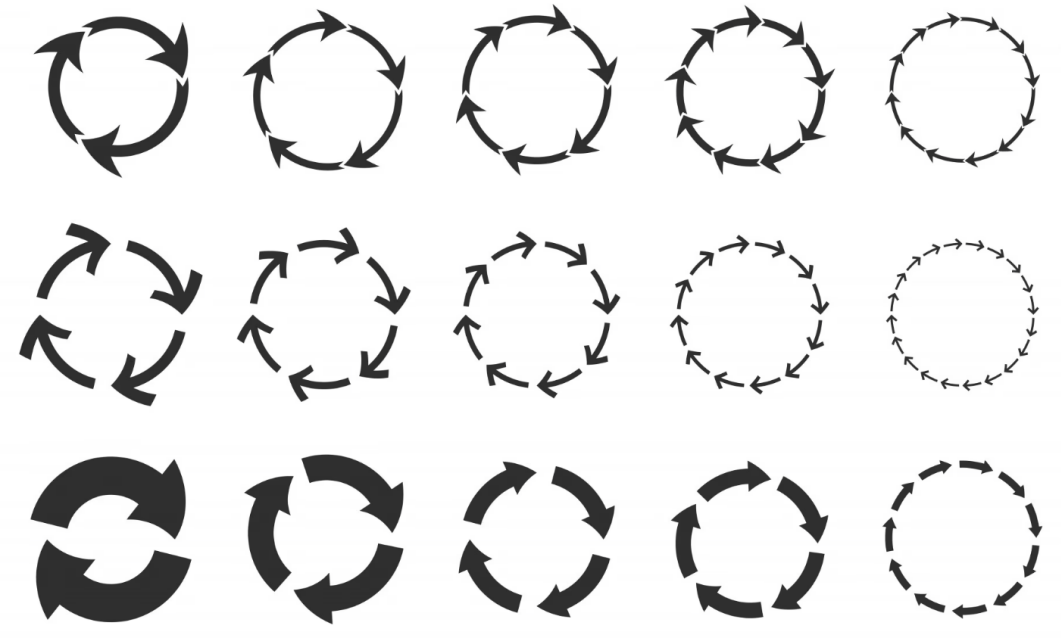
Can we combine the technique of Sections 3.3 and 3.5 to find the roots of a cubic equation?
If you know one root of a cubic, how can you use the quadratic formula to find the remaining two roots, if they exist?
Hint: Factor a cubic into

$$(x - r_1)(ax^2 + bx + c)$$

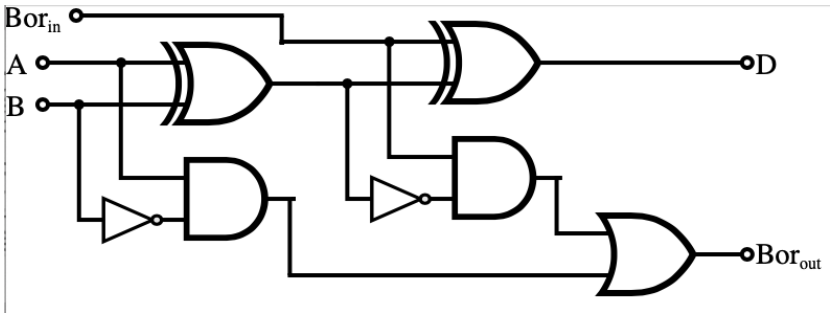
once you know that r_1 is a root. But how can you find a, b, and c?

Chapter 4

Logic and Looping



https://en.wikipedia.org/wiki/File:Polynomialdeg3.svg



4.1 Logic

Definition 1 (proposition)**.** *A* proposition *is a logical statement which has a truth value of true or false.*

For example, let p be the proposition: Light travels at 300,000 km/s in free space; and let q be the proposition: the atomic number of carbon is 33.

We may ask: Is p true? The answer is, yes. Likewise, we may ask: Is q true? The answer is, no.

Definition 2. *A* predicate *is a proposition valued function. We talk about the predicate being true in the case that the proposition which it outputs is true.*

Examples of predicates are *prime(x)*, or $x > 42$. For each of these, given an input value, we derive a proposition, then we can ask whether that proposition is true or false. Given the predicate $prime(x)$ and given the value 13, we construct the proposition: 13 is prime; then we may ask whether that proposition is true. Given the value 15, we may as: Is 15 prime? Given the value -12, we may ask: Is -12 greater than 42?

For example, let $g(n)$ be the predicate *nisprime*. Now we may ask $g(13)$ and $g(15)$. $g(13)$ is true while $g(15)$ is false. Similarly if $h(n)$ is the predicate $n > 42$, then we may ask $h(-12)$ and get false.

Given two propositions, we may combine them by conjunction. The resulting proposition is true if and only if both of the original propositions are true. *E.g.*, Proposition p: Light travels at 300,000 km/s in free space. Proposition q: The atomic number of carbon is 33. Since p true, and q is false, the proposition *conjoining* the two, $p \wedge q$, is consequently false.

Likewise, predicates can be conjoined (or intersected). If we can compute whether given an integer n, it is prime, and we can also compute whether it is greater than 42, we can therefore compute whether it is prime and greater than 42.

$(g \wedge h)(n)$ is true if $g(n) \wedge h(n)$.

Disjunction is the dual of conjunction. The disjunction of two propositions is true if either of the original propositions is true. For example, given p and q as above, the disjunction $p \vee q$ is true because at least one of p and q is true.

Similarly given an integer, n, we can decide whether it is prime or greater than 42, by testing both predicates and combining the output by disjunction.

If a predicate is true, its logical negative is false. If a predicate is false, then its logical negative is true. The if p is a predicate then either p is true or $\neg p$ is true.

We may combine set mechanics and predicates with something called a *set comprehension*.

$$\{m \in \mathbb{N} \mid m > 42\}$$

The idea is that we take the subset of some set ($\mathbb{N}$ in this case) by taking all the elements of that set which satisfy a given predicate.

4.1.1 Implication

If proposition q is true whenever p is true then we can say that p implies q, or $p \implies q$. Note that $p \implies q$ does not say anything about what happens if p is false.

An implication is yet again a proposition, and like any proposition it may be true or false. For example the following is a false implication: $n \in \mathbb{Z} \implies n > 1$. However, the following implication is true: $n \in N \implies n = 0 \vee n \geq 1$.

4.1.2 Quantifiers

$\forall$ specifies a proposition (whose value is either true or false) for which a given predicate is true for every element of a set.

$$\forall x \in N, n > -1$$

$\exists$ specifies a proposition (whose value is either true or false) for which a given predicate is true for at least one element of a set.

$$\exists x \in N, n > 42$$

$\forall$ is always true if the set is empty, we can see this with De Morgan's theorem below. $\exists$ is always false if the set is empty.

$$\forall x \in \emptyset, n > -1$$
$$\exists x \in \emptyset, n > -1$$

4.1.3 De Morgan's theorem

We saw De Morgan's theorem for sets already. But logic and set theory are duals, so De Morgan's theorem can also be stated for propositions and predicates.

$$\neg(p(x) \vee q(x)) = \neg p(x) \wedge \neg q(x)$$
$$\neg(p(x) \wedge q(x)) = \neg p(x) \vee \neg q(x)$$

We may apply De Morgan's theorem to quantifiers

$$(\forall x \in \mathbb{N}, p(n)) = \neg(\exists x \in \mathbb{N}, \neg p(n))$$
$$\neg(\forall x \in \mathbb{N}, p(n)) = (\exists x \in \mathbb{N}, \neg p(n))$$

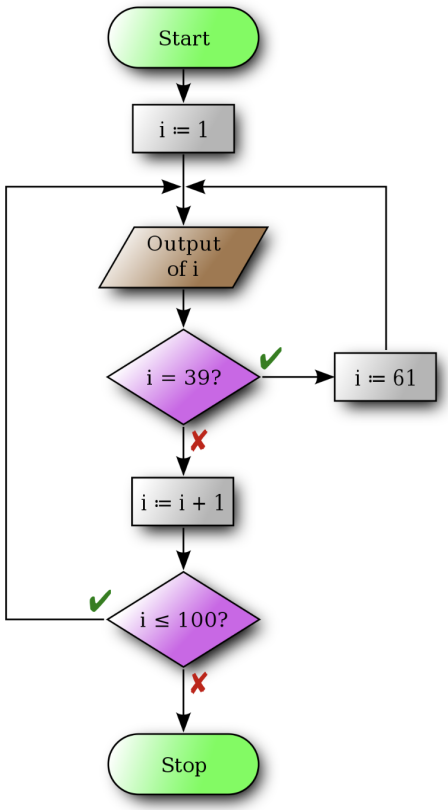
$$(\exists x \in \mathbb{N}, p(n)) = \neg(\forall x \in \mathbb{N}, \neg p(n))$$
$$\neg(\exists x \in \mathbb{N}, p(n)) = (\forall x \in \mathbb{N}, \neg p(n))$$

Question: How can we use De Morgan's theorem to show that

$$\forall x \in \emptyset, p(x)$$

is true independent of p?

4.2 Looping



Three very common styles of loops are

- loop for detection
- loop for collection
- loop for side-effect

In all cases, the loops may be singular, or they may multiple loops running concentrically. Loops may also contain conditions to skip certain iterations.

4.2.1 Loop for detection

Sometimes we need to loop to detect some condition or find an element which meets some condition. For such a loop, use one of:

1. `any` — Is something true at least once? $\exists a, 0 \leq a < 10, a \equiv 1 \mod 3$

```
# (exists?) is there a number 0 <= a < 10 for which a%3 == 1
any(a % 3 == 1 for a in range(10))

# can use multiple lines for readablity
any(a % 3 == 1
    for a in range(10))
```

2. `all` — Is something always true? $a > 2, \forall a \in \{2, 5, 12, -34\}$

```
# (for every?) is something true for all elements in a list
all(a > 2
    for a in [2, 6, 4, 12, -34])
```

3. `next` — Find an element which makes something true.

```
# find the first element which satisfies a condition
next(a
     for a in range(12)
     if a*a > 100)
```

   What does `next` do when it fails to find an element matching the predicate? If `next` fails to find an element which satisfies the logical query, an exception is thrown.

```
# find the first element which satisfies a condition
next(a
     for a in [1, 2, 3]
     if a*a > 100)
```

   The exception looks something like this in the output window:

```
/usr/local/bin/python3.9 mgs-2023/src/looping/examples.py
Traceback (most recent call last):
  File "mgs-2023/src/looping/examples.py", line 12, in <module>
    next(a
StopIteration
```

We see that `any` and `all` correspond to the quantifiers $\exists$ and $\forall$, and we have seen how De Morgan's theorem applies to quantifiers. How does De Morgan's theorem apply to `any` and `all`?

```
# not all not === any
not all(a <= 2
        for a in [2, 6, 4, 12, -34])
any(a > 2
    for a in [2, 6, 4, 12, -34])

# not any not === all
not any(a <= 2
        for a in [2, 6, 4, 12, -34])
all(a > 2
    for a in [2, 6, 4, 12, -34])
```

4.2.2 Loop for collection

Similar to set comprehensions in set theory, we can express comprehensions in Python. However, the programmer must decide whether to generate a set or a list.

```
# generate the list [0, 1, 4, 9, 16, 25]
[a*a for a in range(6)]

# generate the set {0, 1, 4, 9, 16, 25}
{a*a for a in range(6)}
```

We sometimes use comprehension built atop concentric loops.

```
# generate the sums of pairs
[a+b for a in range(6)
     for b in range(3)
]
```

We sometimes use comprehension build atop concentric loops with conditionals.

```
# generate the sums of pairs
[a+b for a in range(6)
     for b in range(3)
     if a+b % 2 == 0
]
```

4.2.3 Loop for side-effect

Sometimes (often) we need to loop for side-effect: printing, modifying an array, communicating with web browser, updating a database, etc. To look for side-effect, use `for`:

```
for a in range(6):
  print(a*a)
```

Of course we may use multiple loops concentrically and with conditionals.

```
for a in range(10):
  if f(a) > 2:
    for b in range(a,15):
      print(a*b)
```

4.3 Programming Projects

1. Choose one challenge:

(a) Collect all prime numbers between n and m.

(b) Collect all Pythagorean triples between 1 and n.

(c) Find (print or collect) all solutions to $a^3+b^3+c^3 = 1$ for a, b, c in range of $-n$ to n.

(d) Taxi cab numbers: For a given n, find numbers between $-n$ and n which are the sum of two cubes in two different ways. E.g., $1729 = 12^3 + 1^3 = 9^3 + 10^3$.

(e) Linear Diophantine Equations: Given integers a, b, c, and n, find all integer solutions ($|x| < n$ and $|y| < n$) to the equation $ax + by = c$. *E.g.*, $2x + 4y = 28$ has $x = 12$, $y = 1$ as solution but also $x = 2$, $y = 6$.

2. Work as team.

3. Create a new file in the `looping` directory.

4. Try to solve the challenge using `any`, `all`, `next`, `for`.

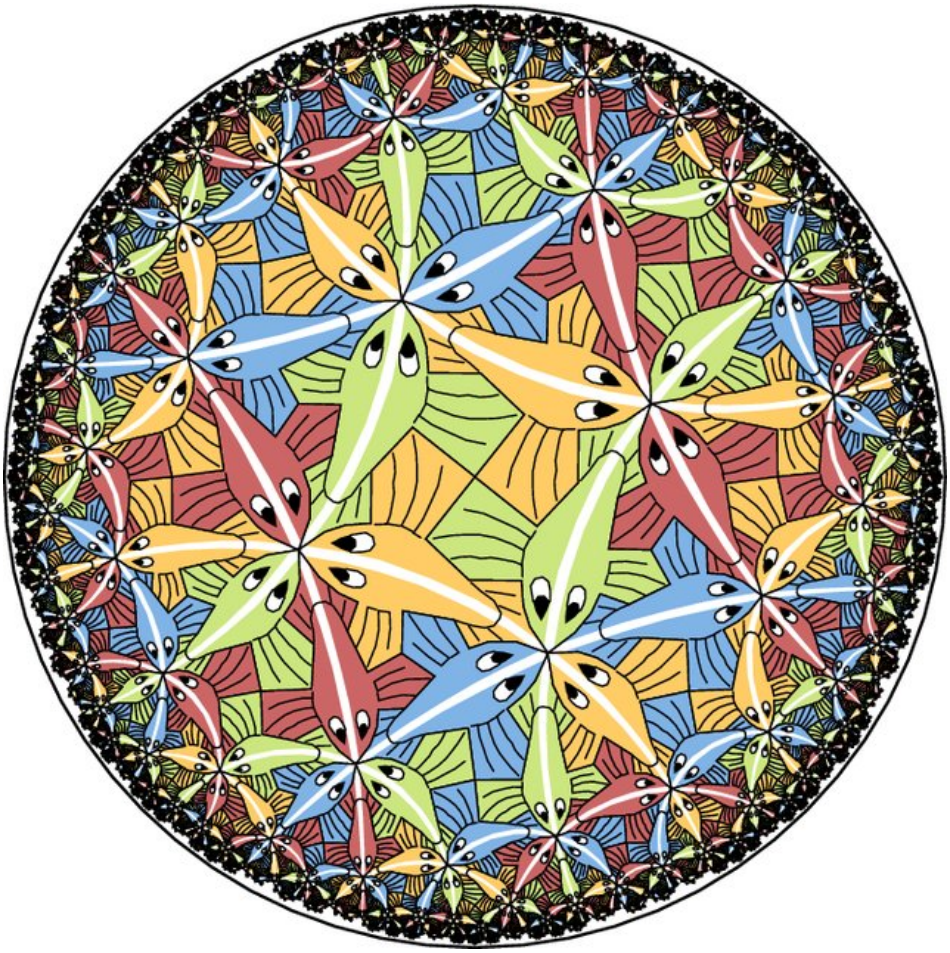
5. (Optional) If possible, can you make it faster? *E.g.*, decrease the search space.

6. When finished and and the code is working, submit a *pull request*.

7. Show, explain, and defend your solution to your fellow scholars.

Chapter 5

Abstract Algebra



https://www.researchgate.net

In Section 3.2, you worked with a function which computes the (real) roots of a second-degree (quadratic) polynomial given only its coefficients. There is an analogous formula to compute the roots of a cubic (degree-3) polynomial, and still another to compute the roots of a degree-4 polynomial. Is there such formula for a polynomial of degree 5 (quintic) or higher?

The question of whether such a formula exists for the quintic had been an open question for 250 years until Niels Henrik Abel proved its non-existence in 1823 at the age of 21.

Around the same time a young, 20 year old, political activist of post-revolutionary France, Evariste Galois, attempted to explain why some higher-order polynomials are *solvable in terms of radicals* and others are not.

Galois, as a staunch republican, would have wanted to participate in the July Revolution of 1830 but was prevented by the director of the École Normale.

I use the word *attempted* not to imply that his explanation was wrong. To the contrary, his ideas were correct but nobody understood his treatment of the subject. One well-respected mathematician of the time, Joseph Fourier, took home a sizable work of Galois to digest it but unfortunately died unexpectedly, and the work was nearly lost.

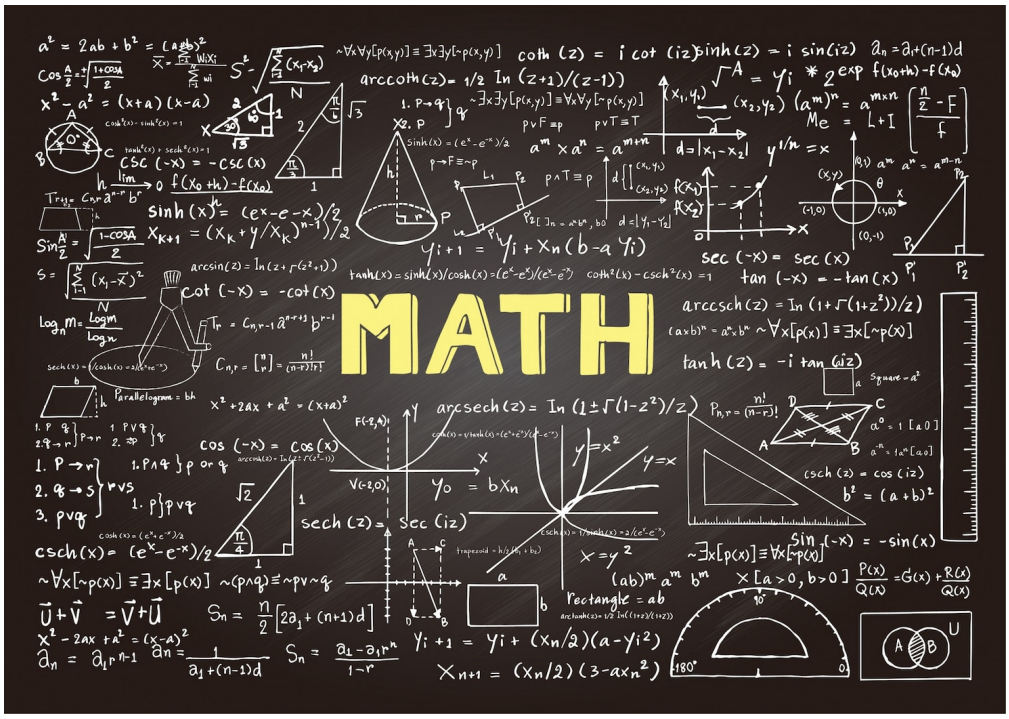
Evariste Galois died at 20 years old as a result of a duel.

The story of Evariste Galois who quickly wrote down the principles of abstract algebra before his death as a result of a duel, is depicted in several YouTube videos: notably *Evariste Galois a documentary* `https://www.youtube.com/watch?v=J6dsanpnpt0` and *Galois Theory Explained Simply* `https://www.youtube.com/watch?v=Ct2fyigNgPY`, both produced by Math Visualized.

It is not surprising that mathematicians did not understand the work of Galois during his short lifetime. Galois's handwriting was confusing at best.

In this unit, we will study sets with mathematical structure. This approach to *abstract algebra* is of course much cleaner than that introduced by Galois. Galois somewhat flippantly introduced the *group* and the *field* just as a tool to talk about the solvability of a polynomial, probably without realizing that they deserve in-depth study themselves.

There are several mathematical structures defined by axioms. We will thereafter write programs as individuals and as teams to manipulate these structures.



https://www.freepik.com

5.1 Monoid

Definition 3 (Monoid)**.** $(S, \circ)$ *is called a* monoid *if*

1. *Closure:* $a, b \in S \implies a \circ b \in S$.
2. *Associative:* $a, b, c \in S \implies (a \circ b) \circ c = a \circ (b \circ c)$
3. *Identity:* $\exists e \in S$ *such that* $a \in S \implies a \circ e = e \circ a = a$

Notice that we assume that a monoid has an identity element, but we don't assume that it is unique. Can a monoid have more than one identity element: e_1 and e_2? The answer is no, but why?

Theorem 1. *The identity of a monoid is unique.*

Proof. Suppose e_1 and e_2 are both identities of monoid, M. Since e_1 is an identity, then $e_1 \circ e_2 = e_2$. But since e_2 is an identity, then $e_1 \circ e_2 = e_1$. Thus

$$e_1 = e_1 \circ e_2 = e_2$$

□

5.2 Examples of monoids

1. $\mathbb{N}$, the set of natural numbers is a monoid. What is the operation? What is the identity element?

2. $(\mathbb{Z}, -)$, the set of integers under subtraction is not a monoid. Why? Which axiom(s) fail?

3. The set of even integers is a monoid. What is the operation? What is the identity element?

4. The set of even integers under multiplication is a not monoid. Why? Which axiom(s) fail?

5. The set of integers under exponentiation is not a monoid. Why? Which axioms fail?

6. $\mathbb{R}$, the set of positive real numbers is a monoid. What is the operation? What is the identity element?

7. The set of 2×3 matrices is a monoid. What is the operation? What is the identity element?

8. The set of 2×3 matrices under multiplication is not a monoid. Which axiom(s) fail? However the set of 3×3 matrices under multiplication is a monoid. Why?

9. The set of subsets of a given set using the operation of union is a monoid. What is the identity element?

10. The set of subsets of a given set using the operation of intersection is a monoid. What is the identity element?

11. The set of logical predicates is a monoid. There are multiple possible operations. Which operations? What is the identity element for each operation?

12. Let Σ be any set of distinguishable elements: $\Sigma = \{a, b, c, d\}$. Now consider the set, $\mathcal{L}(\Sigma)$, of all sequences of finite length $(x_1, x_2, \ldots, x_n)$ for which $x_i \in \Sigma$ for $i = 1, 2, \ldots, n$, $n \geq 0$. I.e.,

$\mathcal{L}(\Sigma)$ includes all sequences of length 12, and all sequences of length 54, and all sequences of length 10,051, etc.

Let + denote sequence concatenation. E.g.,

$$(a, c, a, a) + (d, a, c, a, b) = (a, c, a, a, d, a, c, a, b)\,.$$

$\mathcal{L}(\Sigma)$ is a monoid.

What is its identity element?

Is it a commutative monoid?

13. The integers on the face of the clock form a monoid under addition with the following addition table.

+	1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12	1
2	3	4	5	6	7	8	9	10	11	12	1	2
3	4	5	6	7	8	9	10	11	12	1	2	3
4	5	6	7	8	9	10	11	12	1	2	3	4
5	6	7	8	9	10	11	12	1	2	3	4	5
6	7	8	9	10	11	12	1	2	3	4	5	6
7	8	9	10	11	12	1	2	3	4	5	6	7
8	9	10	11	12	1	2	3	4	5	6	7	8
9	10	11	12	1	2	3	4	5	6	7	8	9
10	11	12	1	2	3	4	5	6	7	8	9	10
11	12	1	2	3	4	5	6	7	8	9	10	11
12	1	2	3	4	5	6	7	8	9	10	11	12

The movement of the hands of the clock is an example of a monoid arithmetic.

We may verify that for every $a \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ we have $12 + a = a + 12 = a$.

14. The integers on the face of the clock also form a monoid under multiplication with the following multiplication table.

*	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	2	4	6	8	10	12
3	3	6	9	12	3	6	9	12	3	6	9	12
4	4	8	12	4	8	12	4	8	12	4	8	12
5	5	10	3	8	1	6	11	4	9	2	7	12
6	6	12	6	12	6	12	6	12	6	12	6	12
7	7	2	9	4	11	6	1	8	3	10	5	12
8	8	4	12	8	4	12	8	4	12	8	4	12
9	9	6	3	12	9	6	3	12	9	6	3	12
10	10	8	6	4	2	12	10	8	6	4	2	12
11	11	10	9	8	7	6	5	4	3	2	1	12
12	12	12	12	12	12	12	12	12	12	12	12	12

In the clock multiplication monoid, 1 has the property that $1 \times a = a \times 1 = a$ for every $a \in \{1, 2, \ldots, 12\}$.

5.3 Fast Exponentiation in a Monoid

Recall the definition of exponentiation in Equation (2.6). In any monoid with operation $\circ$ and identity element e, we can compute an integer power of an element more efficiently according to the following formula.

$$x^n = \begin{cases} e & ;\text{if } n = 0 \\ x & ;\text{if } n = 1 \\ x \circ x^{n-1} & ;\text{if } n \text{ is odd} \\ (x^{\frac{n}{2}}) \circ (x^{\frac{n}{2}}) & ;\text{if } n \text{ is even} \end{cases} \tag{5.1}$$

If the monoid is also group, then inverse elements exist, so we can talk about *negative* exponents in the sense that $x^{-n} = (x^{-1})^n = (x^n)^{-1}$, in which case we can efficiently compute both positive and negative powers. Either we compute $x^{|n|}$ and then compute its inverse, or we can compute x^{-1} and then raise that to the power of $|n|$.

$$x^n = \begin{cases} e & ;\text{if } n = 0 \\ x & ;\text{if } n = 1 \\ x \circ x^{n-1} & ;\text{if } n > 0 \text{ is odd} \\ (x^{\frac{n}{2}}) \circ (x^{\frac{n}{2}}) & ;\text{if } n > 0 \text{ is even} \\ (x^{-1})^{|n|} & ;\text{if } n < 0 \end{cases} \tag{5.2}$$

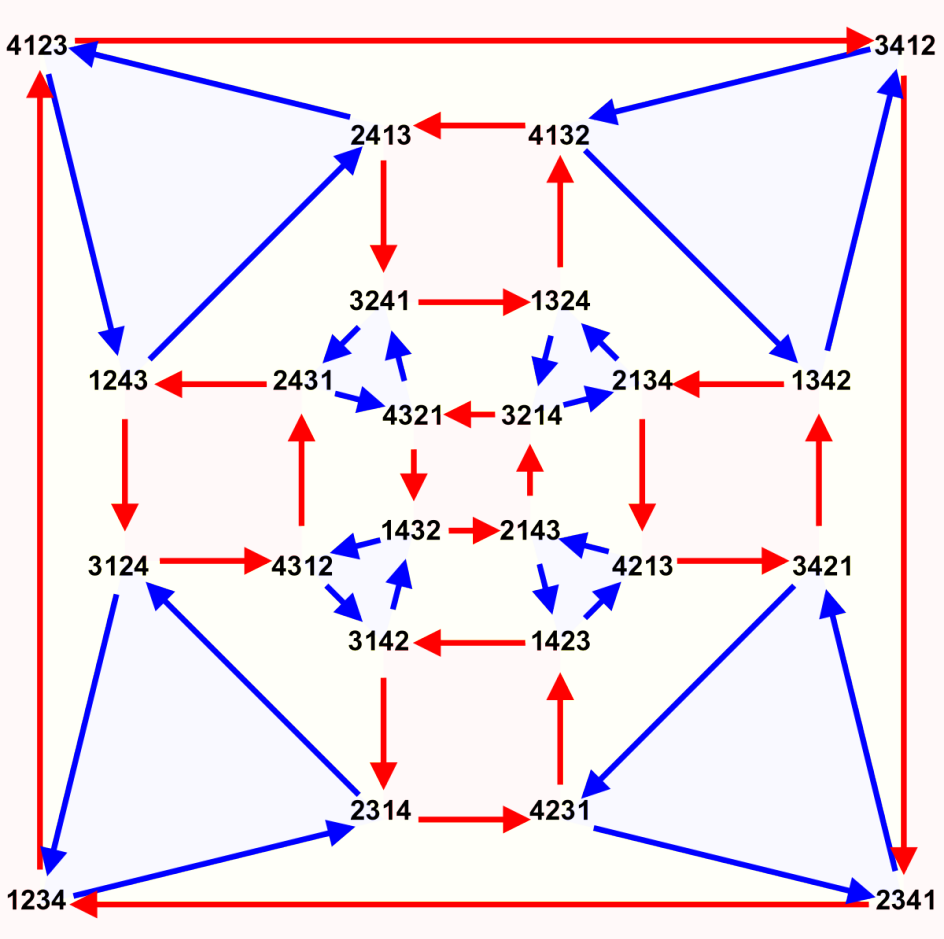
Discussion: What must you do to allow negative exponents? Is it possible for a monoid? Which algebraic structure is necessary to support negative exponents?

5.4 Proofs with Monoids

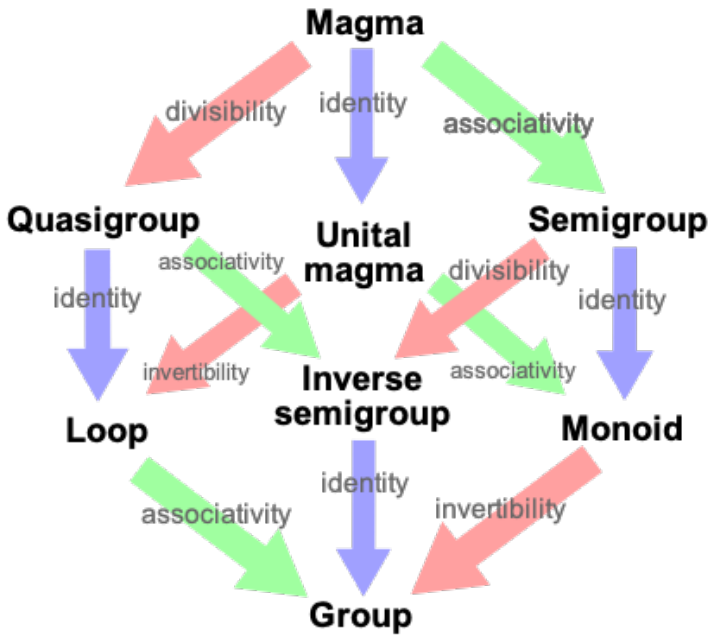
- Give an example of a 2 element monoid.
- Given an example of a 3 element monoid.

Chapter 6

Groups, Rings, and Fields



6.1 Group



There are many algebraic structures which are linked by subset relations.

A notable property missing from a monoid is the ability to solve equations. As an example, take the 12-clock above. Which of these equations can be solved in the integers?

1. Solve for a in $a \times 3 = 7$.
2. Solve for b in $4 \times b = 8$.
3. Solve for c in $7 \times c = 11$.

We observe that some such equations have unique solutions, such as $4 \times b = 8 \implies b = 2$. However, to systematically solve such an equation we need to be able to find an inverse with respect to the operation of the monoid.

$$a \times 3 = 7$$
$$a \times 3 \times 3^{-1} = 7 \times 3^{-1}$$

But there is no integer, $3^{-1} \in \mathbb{Z}$, such that $1 = 3 \times 3^{-1}$. However, some monoids do have the property that every element is invertible. A

monoid for which every element has a unique inverse is called a group. Essential to the idea of inverse is that an inverse be unique.

Definition 4 (Group)**.** $(G, \circ)$ *is called a* group *if*

1. $(G, \circ)$ *is a monoid.*
2. *Inverse:* $\forall a \in G \ \exists a^{-1} \in G$ *such that* $a \circ a^{-1} = a^{-1} \circ a = e$

Definition 5 (Abelian Group)**.** *If* G *is a group such that* $a \circ b = b \circ a$ *for all* $a, b \in G$*, then we call* G *an* Abelian *group.*

A group has an identity which is also the identity of the monoid, which we already know to be unique. See Theorem 1. But we can ask a similar question about uniquenes of the inverse. Can an element $a \in S$ have two different inverses? No, but why?

Theorem 2. *Each element of a group has exactly one inverse.*

Proof. Let G be a group with identity e. Let $a \in G$ have inverses x and y.

$$\begin{aligned} x &= x \circ e \\ &= x \circ (a \circ y) \\ &= (x \circ a) \circ y \\ &= e \circ y \\ &= y \end{aligned}$$

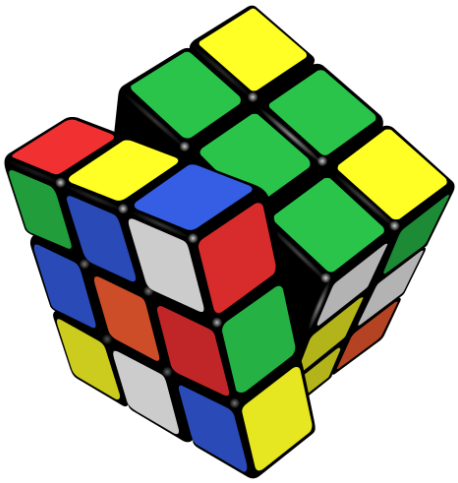
□

6.2 Examples of groups

1. The set of integers, $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$, is a group under integer addition. Why?
   - $(\mathbb{Z}, +)$ is a monoid with 0 being the identity.
   - If $a \in \mathbb{Z}$ there exists $b \in \mathbb{Z}$ such that $a + b = 0$. E.g. $12 + (-12) = 0$

2. The set of integers under multiplication is not a group. Why?

3. The set of rotations of the $n \times n$ Rubik's cube is a group.



The set of rotations of the Rubik's cube is a group.

Every rotation has an inverse rotation, and the null-rotation is the identity.

4. The set of 3×3 matrices of real numbers is not a group. Why? What about the set of *invertible* 3×3 real matrices?

5. The set of subsets of a given set using the operation of union is not a group. Why?

6. The set of subsets of a given set using the operation of intersection is not a group. Why?

7. The set of subsets of a given set, G, using

$$A \circ B = (A \cap \overline{B}) \cup (\overline{A} \cap B)$$

as the operation, is a group. Every element is its own inverse. The identity element is the empty set, $\emptyset$.

8. Imagine a clock going from 1 to 11, rather than 1 to 12. Similar to the clock monoid seen above, here is the multiplication table of the strange 11-clock.

*	1	2	3	4	5	6	7	8	9	10	11
1	1	2	3	4	5	6	7	8	9	10	11
2	2	4	6	8	10	1	3	5	7	9	11
3	3	6	9	1	4	7	10	2	5	8	11
4	4	8	1	5	9	2	6	10	3	7	11
5	5	10	4	9	3	8	2	7	1	6	11
6	6	1	7	2	8	3	9	4	10	5	11
7	7	3	10	6	2	9	5	1	8	4	11
8	8	5	2	10	7	4	1	9	6	3	11
9	9	7	5	3	1	10	8	6	4	2	11
10	10	9	8	7	6	5	4	3	2	1	11
11	11	11	11	11	11	11	11	11	11	11	11

In this example, we see that the set is NOT a multiplicative group, because the element 11 has no inverse.

9. Every non-11 element in the table above has an inverse. We can tediously verify the existence of a unique inverse for every non-11 element.

   The set of non-11 elements of the 11-clock is a multiplicative group with the following multiplication table.

*	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	1	3	5	7	9
3	3	6	9	1	4	7	10	2	5	8
4	4	8	1	5	9	2	6	10	3	7
5	5	10	4	9	3	8	2	7	1	6
6	6	1	7	2	8	3	9	4	10	5
7	7	3	10	6	2	9	5	1	8	4
8	8	5	2	10	7	4	1	9	6	3
9	9	7	5	3	1	10	8	6	4	2
10	10	9	8	7	6	5	4	3	2	1

10. In Section 5.1, we defined the set $\mathcal{L}(\Sigma)$ of finite-length sequences using list concatenation as the monoid operation. This set is not a group. Why?

6.2.1 Challenge

1. Can you find a 3-element non-Abelian group (or non-commutative monoid)?

2. How many possible multiplication tables are there if a group (or monoid) has exactly 3 elements?

3. Not every group is Abelian. However, every group has at least one element which commutes with every other element of the group. What is that element?

4. Experimentally determine whether the set of all elements of a group which commute with all other elements is a group. Stated more clearly. Let G be a group, and let

$$H = \{x \in G \mid x \circ y = y \circ x, \quad \forall y \in G\}.$$

   H is called the *center* of G. Is the center of a group also a group?

6.3 Exponentiation in a Group

Create an exponentiation function for a group which also handles negative exponents.

Why can we compute negative exponents in a group but not in a monoid? Recall Section 5.3.

6.4 Ring

A monoid and a group are both sets with a single operation for which the elements of the set interact to obey a set of axioms (the monoid axioms or the group axioms). Now we will look at sets with two operations, which you can naïvely think of as addition and multiplication.

Definition 6 (Ring)**.** $(S, +, \times)$ *is called a* ring *if*

1. $(S, +)$ *is an Abelian group.*
2. $(S, \times)$ *is a monoid.*
3. *Left-to-right distribution:*
   $a, b, c \in S \implies a \times (b + c) = (a \times b) + (a \times c)$.
4. *Right-to-left distribution:*
   $a, b, c \in S \implies (b + c) \times a = (b \times a) + (c \times a)$.

In the special case that $(S, \times)$ *is commutative,* $(S, +, \times)$ *is called an Abelian ring.*

In the set of whole numbers we can think of multiplication as repeated addition. However, in many cases this does not hold. For example, matrix multiplication cannot be expressed simply as repeated matrix addition. In fact, in a ring, the only thing that connects the addition and the multiplication is the distributive axiom. Multiplication or addition may be defined in strange ways, but if the ring axioms are satisfied, the ring is valid.

In the integers $0 \cdot a = a \cdot 0 = 0$. Actually this principle, called *annihilation* is true for rings in general as shown in Theorem 3. Several other familiar connections between multiplication and addition also hold for rings, in general. In particular $-1 \cdot -1 = 1$ (Theorem 4), and $-1 \cdot a = -a$ (Theorem 5).

Theorem 3. *For ring,* S *with additive identity,* z*, we have* $za = az = z$ *for all* $a \in S$*.*

Let's denote the additive inverse of any element x by $\overline{x}$

Proof.

$$\begin{aligned} za &= za + z \\ &= za + (za + \overline{za}) \\ &= (za + za) + \overline{za} \\ &= (z + z)a + \overline{za} \\ &= za + \overline{za} \\ &= z \end{aligned}$$

Likewise,

$$\begin{aligned} az &= az + z \\ &= az + (az + \overline{az}) \\ &= (az + az) + \overline{az} \\ &= a(z + z) + \overline{az} \\ &= az + \overline{az} \\ &= z \end{aligned}$$

□

Theorem 4. *For ring, S, with multiplicative identity e, we have $\overline{e} \cdot \overline{e} = e$.*

Proof. Let z be the additive identity of the ring.

$$\begin{aligned} \overline{e} \cdot \overline{e} &= \overline{e} \cdot \overline{e} + z \\ &= \overline{e} \cdot \overline{e} + (\overline{e} + e) \\ &= \overline{e} \cdot \overline{e} + (\overline{e} \cdot e + e) \\ &= (\overline{e} \cdot \overline{e} + \overline{e} \cdot e) + e \\ &= \overline{e} \cdot (\overline{e} + e) + e \\ &= \overline{e}z + e \\ &= z + e && \text{By Theorem 3} \\ &= e \end{aligned}$$

□

Theorem 5. *For ring, S, with multiplicative identity, e, we have $\overline{e} \cdot a = \overline{a}$ for all $a \in S$.*

Proof. Let z be the additive identity.

$$\begin{aligned}
\overline{e} \cdot a &= \overline{e} \cdot a + z \\
&= \overline{e} \cdot a + (a + \overline{a}) \\
&= (\overline{e} \cdot a + a) + \overline{a} \\
&= (\overline{e} \cdot a + e \cdot a) + \overline{a} \\
&= (\overline{e} + e) \cdot a + \overline{a} \\
&= z \cdot a + \overline{a} \\
&= z + \overline{a} && \text{By Theorem 3} \\
&= \overline{a}
\end{aligned}$$

□

6.5 Examples of rings

1. The integers $(\mathbb{Z}, +, \times)$ is a ring.
2. The integers $(\mathbb{N}, +, \times)$ is not a ring. Why?
3. The set of $n \times n$ matrices, for a fixed value of $n > 0$ is a ring.
4. The set of 2×3 matrices is not a ring. Why?
5. $\mathbb{Z}[x]$, the set of polynomials with integer coefficients such as $3x^4 + 2x^2 - 5x + 1$ is a ring.
6. $\mathbb{Q}[x]$, $\mathbb{R}[x]$, $\mathbb{C}[x]$, $(\mathbb{Z}/p)[x]$.
7. The set of polynomials with rational coefficients with leading coefficient equal to 1 is not a ring. $(x^3 + \frac{1}{2}x^2 + 5x - \frac{2}{3})$ Why?
8. The set of subsets of a given set using intersection as multiplication and exclusive or as addition forms a ring.
9. The set of 8-bit sequences (bytes) consisting of 0 and 1, such as 00101101, with the two operations AND and XOR, defined as bitwise operations, using the following tables is a ring.

XOR	0	1
0	0	1
1	1	0

AND	0	1
0	0	0
1	0	1

For example

- 00101101 XOR 00110011 = 00011110
- 00101101 AND 00110011 = 00100001

6.6 Field

The most complicated structure we will examine is the field. You can think of a field as a set which is a ring but more than that. In a ring you can add, subtract, and multiply. But in a field you can also divide with the exception that you cannot divide by zero.

Definition 7 (Field)**.** $(F, +, \times)$ *is called a* field *if*

1. $(F, +, \times)$ *is an Abelian Ring.*
2. $(F \setminus \{0\}, \times)$ *is a (Abelian) group, where* 0 *is the identity for* $+$*.*

6.7 Examples of fields

1. The rational numbers, $\mathbb{Q}$.
2. The set of $n \times n$ matrices is not a field. Why?
3. The set of 2×2 matrices of the form $\begin{bmatrix} a & -b \\ b & a \end{bmatrix}$ where $a, b \in \mathbb{Q}$.
4. The integers modulo 12.
5. The integers modulo any prime such as 7.

$+$	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

$\times$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

With these addition and multiplication tables we can tediously verify that

- 0 is the additive identity.
- 1 is the multiplicative identity.
- Every element has an additive inverse.
- Every non-zero element has a multiplicative inverse.
- Addition and multiplication are associative.
- Addition and multiplication are commutative.
- The distributive property holds.

6. The set $\mathbb{Q}\{\sqrt{2}\} = \{a + b\sqrt{2} \mid a, b \in \mathbb{Q}\}$ is a field. Moreover, each of the following can be expressed in the same form $a + b\sqrt{2}$.

   - $(a_1 + b_1\sqrt{2}) + (a_2 + b_2\sqrt{2})$
   - $-(a + b\sqrt{2})$
   - $(a_1 + b_1\sqrt{2}) \times (a_2 + b_2\sqrt{2})$
   - $\frac{1}{a+b\sqrt{2}}$, provided a, b are not both 0.

   In particular

$$(a_1 + b_1\sqrt{2}) + (a_2 + b_2\sqrt{2}) = \underbrace{(a_1 + a_2)}_{\in\mathbb{Q}} + \underbrace{(b_1 + b_2)}_{\in\mathbb{Q}}\sqrt{2}$$

$$-(a + b\sqrt{2}) = \underbrace{(-a)}_{\in\mathbb{Q}} + \underbrace{(-b)}_{\in\mathbb{Q}}\sqrt{2}$$

$$(a_1 + b_1\sqrt{2}) \times (a_2 + b_2\sqrt{2}) = \underbrace{(a_1a_2 + 2b_1b_2)}_{\in\mathbb{Q}} + \underbrace{(a_1b_2 + a_2b_1)}_{\in\mathbb{Q}}\sqrt{2}$$

$$\frac{1}{a + b\sqrt{2}} = \underbrace{\frac{a}{a^2 - 2b^2}}_{\in\mathbb{Q}} + \underbrace{\frac{-b}{a^2 - 2b^2}}_{\in\mathbb{Q}}\sqrt{2}$$

   Why do we know $a^2 - 2b^2 \neq 0$ and $a + b\sqrt{2} \neq 0$?

6.8 Summary of Algebraic Structures

1. A *monoid* is a set where we can add.

2. A *group* is a set where we can add and subtract.

3. A *ring* is a set where we can add, subtract, and multiply.

4. A *field* is a set where we can add, subtract, multiply, and divide.

For simplicity we omit the formal definition of *semi-ring* which is a set in which we can add and multiply, but not necessarily subtract and divide.

These summaries are vague. For example when we say "add" in a monoid, the actual operation may be addition, multiplication, concatenation, or many other operations depending on the actual monoid. The distinction between adding and multiplying requires that both exist, thus they can only be distinguished in a semi-ring.

If a set has both a 1 and a 0, then dividing by 0 does not make sense. For example, in a field, we require all non-zero elements to have a multiplicative inverse even though all elements have an additive inverse.

Chapter 7

Coding Challenges

https://beaccessible.com/post/html-css-accessibility/

7.1 Compute number of combinations

Recall the definition

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}\,.$$

Write function to compute this quantity several different ways.

7.1.1 Direct computation

Compute $\binom{n}{k}$ directly by computing the three factorials and performing the integer division.

Examine the values of the numerator and denominators. How many bits are necessary for these computations to occur successfully?

In Python, integers have arbitrary precision. What would happen in another language whose integers are limited to 64 or 32 bits?

7.1.2 Compute the list of prime factors

Rather than computing $n!$ directly, express it as a list of prime factors. *I.e.*,

- Implement a function which, given n, uses recursion to compute and return the list of prime factors of n.
- Use that function to implement another function which, given n, returns the prime factors of $n!$.
- Effectively perform the division by removing all the prime factors in the denominator, $k!(n-k)!$, from those of the numerator, $n!$.
- Finally, multiply the remaining factors to compute $\binom{n}{k}$.

E.g.,

$$\begin{aligned}
\binom{7}{3} &= \frac{7}{3!(7-3)!} \\
&= \frac{7}{3! \cdot 4!} \\
&= \frac{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2}{(3 \cdot 2)(4 \cdot 3 \cdot 2)} \\
&= \frac{7 \cdot \cancel{3} \cdot \cancel{2} \cdot 5 \cdot \cancel{2 \cdot 2 \cdot 3 \cdot 2}}{(\cancel{3} \cdot \cancel{2})(\cancel{2 \cdot 2 \cdot 3 \cdot 2})} \\
&= 7 \cdot 5 \\
&= 35
\end{aligned}$$

7.1.3 Recursive computation

How is $\binom{n}{k}$ related to $\binom{n-1}{k-1}$?

Notice that if $\binom{n}{k} = \frac{n!}{k!\ (n-k)!}$, and if $n > 1$ and $k > 1$, then we have

$$\begin{aligned}
\binom{n-1}{k-1} &= \frac{(n-1)!}{(k-1)!\ ((n-1)-(k-1))!} \\
&= \frac{(n-1)!}{(k-1)!\ (n-k)!} \\
&= \frac{\frac{n!}{n}}{\frac{k!}{k}\ (n-k)!} \qquad \text{; because } n! = n(n-1)! \text{ and } k! = k(k-1)! \\
&= \frac{k}{n} \times \frac{n!}{k!\ (n-k)!} \\
&= \frac{k}{n}\binom{n}{k}
\end{aligned}$$

So we can solve for $\binom{n}{k}$ to obtain:

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}. \tag{7.1}$$

Notice that that

$$\binom{n}{1} = \frac{n!}{1!\ (n-1)!} = \frac{n\ \cancel{(n-1)!}}{1!\ \cancel{(n-1)!}} = n \tag{7.2}$$

and that

$$\binom{n}{n-k} = \frac{n!}{(n-k)!\ (n-(n-k))!} = \frac{n!}{(n-k)!\ k!} = \binom{n}{k}. \tag{7.3}$$

Finally by (7.3), we have

$$\binom{n}{k} = \binom{n}{\min\{k, n-k\}} \tag{7.4}$$

As an example of (7.4), suppose we need to compute $\binom{17}{12}$. We chose the minimum of 12 and $17-12$ which is 5. We know by (7.3) that $\binom{17}{12} = \binom{17}{5}$, and since $\binom{17}{5}$ is easier to compute than $\binom{17}{12}$, we compute $\binom{17}{5}$.

How can we use the equalities (7.1), (7.2), and (7.4) to write yet another implementation of `choose` in Python?

Example 1. *Let's look at an example: Compute* $\binom{7}{3}$.

$$\binom{7}{3} = \frac{7}{3}\binom{6}{2} = \frac{7}{3}\frac{6}{2}\binom{5}{1} = \frac{7}{3}\frac{6}{2}5 = \frac{7}{\cancel{3}}\frac{\cancel{6}}{\cancel{2}}5 = 7 \cdot 5 = 35$$

However, if we compute $\binom{7}{4}$*, we should get the same thing. Do we?*

$$\binom{7}{4} = \frac{7}{4}\binom{6}{3} = \frac{7}{4}\frac{6}{3}\binom{5}{2} = \frac{7}{4}\frac{6}{3}\frac{5}{2}\binom{4}{1} = \frac{7}{4}\frac{6}{3}\frac{5}{2}4 = \frac{7}{\cancel{4}}\frac{\cancel{6}}{\cancel{3}}\frac{5}{\cancel{2}}\cancel{4} = 7 \cdot 5 = 35$$

7.1.4 Review of different approaches

Compare the functions both in terms of computation time, and also find values of $\binom{n}{k}$ which are computable by one function but not the other because of integer overflow.

7.2 Modulo Arithmetic

1. For a given n, implement addition, subtraction, and multiplication modulo n.

2. If n is prime, implement division.

3. A number can have multiple square roots modulo n, find such a case.

4. Implement exponentiation modulo (prime) p. How many multiplications are necessary to compute $a^b \mod p$?

5. If p is prime, and $a \notin \{0, 1\}$, what are a^0, a^1, a^2, $\ldots$, a^{p-1}, $a^p \mod p$?

6. If n is composite, and $a^b = a^c$ for $a \notin \{0, 1\}$, what do we know about b and c?

7. If p is prime, find Pythagorean triples modulo p.

7.3 Team Project: Structure Recognition

- Implement functions to detect monoid, group, ring, and field given a finite set, and the $+$ and $\times$ operations.
- When finished with your function, commit, push, and send a pull request.
- To test your code, you will need to pull the changes in the repository.
- Define $\mathbb{Z}/n$, the integers modulo n. Is it a monoid? Is it a group (Abelian)? etc. ...
- For which values of n is $\mathbb{Z}/n$ a field?