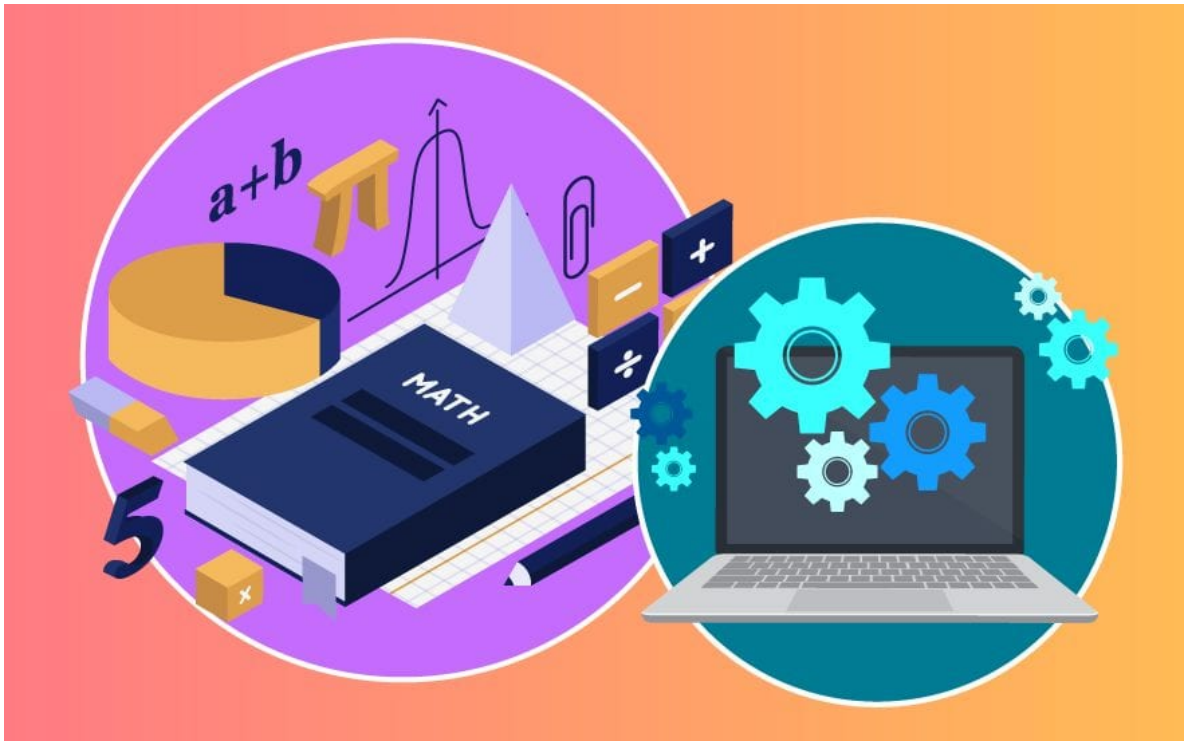
Practical Classical Algebra



Dr. Jim Newton

September 22, 2026

Download the most recent version of this handout:

Contents

Chapter 0

Forward

0.1 About this Course

In this course you will learn or review basic classical algebra and reinforce your learning by writing short programs to perform computations relating to the mathematical principles.

It is assumed that you are attending a Python programming course in parallel. So some programming concepts may not be introduced in the same order in both courses. For this reason the beginning of the course might be seen as difficult, and might require extra time and dedication. Hopefully by the end of the course, lots of patterns and techniques will be much clearer because you have studied them from different perspectives, and you have used the same tools to solve seemingly unrelated problems.

A recurring theme in this course is the close relationship between mathematical thinking and programming. Recursive definitions, algebraic structures, and inductive reasoning all have direct counterparts in code. Students who are comfortable thinking mathematically often find this style surprisingly natural in Python: a recursive function mirrors an inductive definition; an algebraic structure suggests the shape of its implementation. This course makes that connection explicit, treating programming not as a skill bolted onto the mathematics but as a medium in which mathematical ideas come alive. Writing a test case is itself an act of mathematical precision — it forces you to state exactly what a function must do — and working through a failing test is one of the most effective ways to sharpen both your programming and your

mathematical understanding simultaneously.

0.2 Course Objectives

1. Review of Classical Algebra for Programmers
2. Develop Programming Skills
3. Practice Testing and Debugging
4. Understand mathematical structures and recursive thinking as foundations for computation

0.3 How to Read this Document

Throughout this document, margin annotations and colored boxes serve as navigation aids.

Margin annotations. Icons appearing in the margin signal the nature of the adjacent content.

- **Essential** — a core concept you must understand and remember.
- **Recall** — a pattern or technique introduced elsewhere; look it up if needed.
- **Warning** — a common mistake or a subtle point requiring care.
- **Enrichment** — optional deeper material for the curious reader.

Colored boxes. Different types of mathematical content are distinguished by background color.

Theorem / Claim — A mathematical statement that has been proven true.

Activity — a hands-on task to run or observe, followed by a discussion question for the class.

Definition — A precise meaning assigned to a mathematical term.

Example — A worked illustration of a concept or technique.

Pattern — A reusable strategy or recipe to recognize and apply.

Proof — A logical argument establishing a theorem or claim.

Listing — A code extract for reference.

Chapter Objectives — Learning goals listed at the start of each chapter; use them to check your understanding before moving on.

0.4 Syllabus

Session	Chapter	Topics
1,2	A, B, C, D	Forge Intranet, git
3	1	Python, Testing
4	2	Looping, Logic, Sets
5	3	Relations
6		TD, Review
7	4	Introduction to Recursion
8	5	Divide and Conquer
9	6	Combinations, Memoization
10	7	Polynomials: Addition, Multiplication
11	8	Polynomials: Quotient, Remainder
12		TD, Review
13	9	Limits: Derivative, Binary Search
14	10	Roots: Quadratic, Cubic
15	10.7	Roots: Quartic, Quintic, Arbitrary Degree
16	11	Abstract Algebra: Monoids, Groups
17	12	Abstract Algebra: Rings, Fields

لقد آنجزت واجبي المنزلي ، لكن كلبي آكله

Figure 1: Clear to anyone who reads Arabic

0.5 Evaluation

Your grade in this course will depend on the homework problems you submit. Each chapter ends with a group of *Challenges for the student*. You will be required to complete a template file which contains partial implementations of functions. You must complete the functions to pass a given set of test cases. To obtain a perfect score, you must complete all the exercises and pass all the tests. Partial credit will be given depending on the number of tests passed.

During the several weeks of this course you may need to refactor old code in order to reuse it in later exercises. In doing so there is a risk of breaking old tests. For this reason, the final challenge is that all tests should still pass at the end of the course.

0.6 Philosophy of Programming

Figure 1 contains a piece of text in Arabic. The meaning of the text is completely clear to anyone who understands the Arabic language. If you do not understand the text, it is not because the text is unclear; rather it is because you don't understand the language.

We always strive to write clear, understandable, maintainable, testable code. But from time to time it is difficult to do so, so we have to balance trade-offs. There is an art to writing clear programs. The programmer should strive to be clear, but different people will disagree about what constitutes clarity. The program need not be clear to someone who does not know the language, or who does not know the domain. In general less code is better, but do not take that adage to an extreme. Some programming languages (*e.g.*, Java) force the programmer to type redundant code, called *boiler-plating*. Other programming languages (*e.g.* Common Lisp) allow programmers to avoid huge amounts of boilerplat-

ing by enabling the programmer to conform the language to the needs of the application.

As programmers, we don't always understand what we are doing. In the life of a programmer, you will work on projects in teams, and you will be forced to trust the expertise of colleagues. This principle may also apply to this course, from time to time. There may be times when you don't completely understand why a certain piece of code works. However, this lack of understanding should not hinder you from reusing the code elsewhere where it is needed.

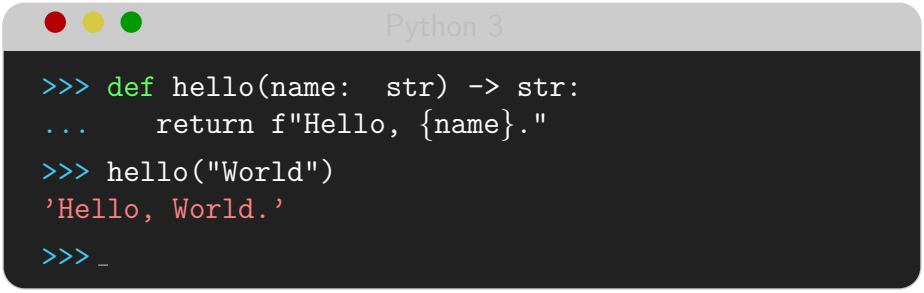
Rule of Thumb: Avoid writing the same code twice. Avoid copy-pasting large blocks of code and changing just a few things. Instead, refactor the code and parameterize it so it can be reused.

Chapter 1

Introduction to Python

Chapter Objectives

- Distinguish between pure functions (no side effects, deterministic) and functions with side effects such as printing.
- Write Python functions using `def`, including nested functions and higher-order functions that accept or return other functions.
- Annotate function signatures with Python type hints (`int`, `str`, `bool`, `List[...]`, `Tuple[...]`, etc.).
- Use Python's built-in collection types: strings, tuples, lists, ranges, and sets, and understand when each is appropriate.
- Implement the `hello` programming assignment.



We *could* use any programming language for this course, but we must choose one. We will use Python. Python is a good programming language for learning. The language is easy to learn, and in many cases easy problems have easy solutions. A programming language is a tool, and no tool is perfect. We will work within the limitations of the language. In fact, in your career as a programmer or software developer, you will always work within the limitations and confines of the implementation language.

In the following sections we look at several constructions which are essential to your programming experience.

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 1.2).

Listing 1.1.1

```
1  a = 12
2  b = -7
3  print(a*b - 3*a)
4
5  a = 13
6  print(a*b - 3*a)
```

An assignment to a variable can also reference the very variable being modified.

Listing 1.1.2

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

Arithmetic is further discussed in Section 1.4.1.

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`.

Listing 1.2.1

```
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.

Listing 1.2.2

```
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.

Listing 1.2.3

```
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.

Listing 1.2.4

```
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`.

Listing 1.2.5

```
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`.

Listing 1.2.6

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

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

In the above example `from math import sin, cos, log, pi, e` was used to make the function names `sin`, `cos`, `log` and the variable names `pi` and `e` usable for the programmer. But in some cases you might already have a variable `e` used for some other purpose in the program. In such a case you may `import math` a library without importing the actual identifiers. In such a case to reference the functions and variables, you must prefix them such as `math.cos` and `math.pi`.

Listing 1.2.7

```
import math

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

1.2.1 Pure Functions

There are two genre of functions, pure functions, and functions with side-effects. The Python language does not support any syntax to distinguish between these two genre. You cannot look at a function declaration and know whether there are side effects.

A *pure function* is a function with no side effects. These functions model mathematical functions. Pure functions take wall-time to execute and compute their return values, but they have no lasting effect on the world. A pure function may be called a second, third, forth time on the same input and will always return the same value. Thus, subsequent calls may sometimes be avoided by storing the return value in a local variable and referencing the variable rather than calling the function subsequent times. In Python, such *optimization* is entirely left up to the choice of the programmer. The language does not optimize this for you.

A *side-effecting function* is a function which changes the environment in some way. For example it may print something to standard output, or may write to a file. Thus calling the function a second time may write to the file a second time. Such a side effect might be desired or might be a bug. The most common side effect is probably printing something to standard output. In fact, this technique is a useful way of debugging.

Another common side effect is to modify a global variable.

A third type of side effect, one which we will *usually* avoid, is to destructively modify a data structure. For example, given a list, a Python program is allowed to change the content of the list. The effect is that every part of your program which has access to the list, will see the new value. The problem is that such a global effect might be accidental. For this reason, we will avoid this type of programming technique. However, there are times when it is unavoidable because of limitations of the Python language.

1.2.2 Higher Order Functions

Functions which return functions are common in mathematics. You've worked with such functions before.. For example the derivative $\frac{d}{dx}$ is a function which takes a function as argument and returns another function. *E.g.*,

$$\frac{d}{dx}\sin = \cos\ . \tag{1.1}$$

Another example is the definite integral.

$$\int_0^x \cos(t)dt = \sin(x)\,. \tag{1.2}$$

Equation (1.2) is perhaps less obviously a function which returns a function than is Equation (1.1). This difficulty is merely an artifact of the notation. Consider the following:

$$I(f) = x \mapsto \int_0^x f(t)dt \tag{1.3}$$

In the case of Equation (1.3) it should be clear that $I(\cos) = \sin$.

In Python a function is allowed to return a function. A function which takes a function as an input value or returns a function in its output is called a *higher order function*.

A function which takes exactly one argument is called a *unary* function. In contrast to a function which takes two arguments is called a *binary* function. We call a function which takes no arguments either a *0-ary* function or sometimes a *thunk*.

The following is a simple but commonly seen example. The function `f` is a unary function takes a parameter `n`, and returns another unary function which adds its input value to `n`

Listing 1.2.8

```
def f(n):
  def g(m):
    return n + m

  return g
```

The function `f` can be called with a single argument, say `3`, in which case it returns a value which is not a number but rather a unary function. One way to use the return value is to store it in a variable.

Listing 1.2.9

```
>>> h = f(3)
>>> h(1)
4
>>> h(0)
3
>>> h(100)
103
```

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.

Listing 1.3.1

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

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

Listing 1.3.2

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

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

Listing 1.3.3

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

1.4 Atomic Types

1.4.1 Numbers

In this section we look at the two most common numerical types, integers and floating-point. The Python types are called `int` and `float`.

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

Listing 1.4.1

```
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`.

Listing 1.4.2

```
>>> 20 / 8  # division (as floating point)
2.5
>>> 20 // 8 # division (as integer, discarding remainder)
2
>>> 20 % 8  # remainder of division (modulus)
4
>>> -20 % 8 # remainder of division (modulus)
4
```

Warning! In some programming languages `-20 % 8` will evaluate to `-4` rather than positive `4`. However, in Python the modulus operator always evaluates to 0 or a positive integer.

We use the modulus operator to test divisibility. It is very common to use an idiom such as `if n % 2 == 0:` to test whether n is even, and `if n % 2 != 0:` to test whether n is odd.

We can use the `type` function to detect which type of value an expression returns.

Listing 1.4.3

```
>>>  20 / 8
2.5
>>>  type(20 / 8)
<class 'float'>
>>>  20 // 8
2
>>>  type(20 // 8)
<class 'int'>
```

1.4.2 Working with Floating Point

Be careful when working with floating point numbers. Numbers such as `0.1`, `0.2`, and `0.3` have exact representations in base 10, but Python stores floating point numbers internally in base 2. Numbers such as `0.5`, `0.25`, and `0.125` have exact representations in base 2, but many numbers do not. Programs are normally written using base 10; *i.e.*, we use textual representation of numbers such as `0.1`, `0.2`, and `0.3` in

our programs. Sometimes, unintuitive things happen.

Listing 1.4.4

```
>>> 0.1 + 0.2 == 0.3
False

>>> 0.2 * 5 == 1.0
True

>>> 0.1 * 2 * 5 == 1.0
True

>>> (0.1 * 2) * 5 == 1.0
True

>>> 0.1 * 2 == 0.2
True

>>> 0.0 == 0.1 + 0.2 - 0.3
False
```

1.4.3 Booleans

We've already seen the integer and float data types, but there is another atomic data type 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`.

Listing 1.4.5

```
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.

Listing 1.4.6

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

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

In Python we may use any value in a *Boolean context*. For example:

Listing 1.4.7

```
if "hello":    # considered True
  print(10)
if "":         # considered False
  print(11)
if [10, 20]:   # considered True
  print(12)
if []:         # considered False
  print(13)
if 42:         # considered True
  print(14)
if 0:          # considered False
  print(15)
```

In general *empty* values are considered `False`, and every other value is considered `True`.

There is a special value `None` which is also considered `False`.

1.5 Collections

A collection is a single object which *contains* other objects which can be manipulated independently.

1.5.1 Strings

A `string` is a string of characters. In many programming languages the character type and the string type are distinct. In Python there is no character type. Extracting the nth element of a string results in a single-character string. When Python prints a string, it omits the surrounding

quotation marks.

A string can be designated by double quotes, `"hello"` or by single quotes `'world'`.

Listing 1.5.1

```
>>> a = "hello"
>>> print(a)
hello

>>> print(a[0])
h

>>> print(a[1])
e

>>> print(a[2])
l

>>> b = 'world'
>>> print(b)
world
```

Notice in the preceding example `print(a[1])` prints the string `e` not `h`. In fact, in Python strings are zero-based; *i.e.*, strings are indexed starting with zero, not with one.

1.5.2 Tuples

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

Listing 1.5.2

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

>>> print(y[2])
(3, 4)

>>> y[0] = 1
```

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.

1.5.3 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.

Listing 1.5.3

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

>>> print(y[2])
[3, 4]

>>> y[2] = 4
>>> print(y[2])
4

>>> print(y)
[4, [0, 2, 4], 4]
```

1.5.4 Ranges

A `range` is like a list, but internally Python stores only the start, stop, and step. Whereas a list of 1000 element requires 1000 objects allocated in memory, a range of n element requires the same amount of memory allocation, regardless of the value of n.

Listing 1.5.4

```
>>> type(range(2,100,3))
<class 'range'>

>>> range(10)
range(0, 10)

>>> range(10,30)
range(10, 30)

>>> range(10,30,2)
range(10, 30, 2)

>>> range(30,20,-2)
range(30, 20, -2)

>>> list(range(30, 20, -2))
[30, 28, 26, 24, 22]
```

1.5.5 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`.

Listing 1.5.5

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

Cetain types are compatible with sets, and certain types are not. Observe the `TypeError: unhashable type` error if you try to create a list with objects which are incompatible.

Listing 1.5.6

```
>>> b = {10,20,30}
>>> print(b)
{10, 20, 30}

>>> a = {1,2,3}
>>> print(a)
{1, 2, 3}

>>> b = set((10,20,30))
>>> print(b)
{10, 20, 30}

>>> c = [a,b]
>>> print(c)
[{1, 2, 3}, {10, 20, 30}]

>>> print(set(c))
Traceback (most recent call last):
  File "...file.py", line 17, in <module>
    print(set(c))
TypeError: unhashable type: 'set'
```

1.5.6 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.

When such a value is printed, it normally is displayed as a single line of output. In the example below, we've displayed it on multiple lines for readability.

Listing 1.5.7

```
>>> from pprint import pprint
>>> 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]])])
>>> pprint(d)
([1, 2, 5, 7],
 (10, 20, 50, 70, 100),
 [[1, 2, 5, 7], (10, 20, 50, 70, 100), [1, 2, 5, 7], [1, 2, 5,
    7]],
 [(1, [2, 3, [[[1, 2, 5, 7],
                (10, 20, 50, 70, 100),
                [1, 2, 5, 7],
                [1, 2, 5, 7]],
       [1, 2, 5, 7]]])])
```

In the code above, the list `d` is actually printed on a very long single line. If you'd like a list to be printed with smarter indentation, you may use the `pprint` function which is available from `from pprint import pprint`.

1.5.7 Operator Overloading

The Python language uses the same operators for different types. Sometimes the same operator can have very different meanings when applied to different types. For numeric types the operators `+` and `*` perform familiar numerical addition and multiplication. However, when applied to strings and lists , the same operators perform concatenation and repetition.

Listing 1.5.8

```
>>> type(1)
<class 'int'>

>>> type("hello ")
<class 'str'>

>>> type([10, 20, 30])
<class 'list'>

>>> "hello" + "world"
'helloworld'

>>> "hello " + "world"
'hello world'

>>> "hello " * 3
'hello hello hello '

>>> [10, 20, 30] * 3
[10, 20, 30, 10, 20, 30, 10, 20, 30]

>>> [10, 20, 30] + [10, 20 ,30]
[10, 20, 30, 10, 20, 30]
```

1.5.8 Collections and Subscripts

Some collections support subscripting. An expression such as `a[1]` retrieves the element at index `1` in the collection, where `a[0]` retrieves the left-most element. If the index is negative, Python interprets it as indexing from the right, but 1-based, not 0-based. So `a[-1]` retrieves the last (right-most) element of the list.

Listing 1.5.9

```
>>> a = range(2,100,3)
>>> b = "hello"
>>> c = [10, 20, 30]
>>> d = (11, 12, 13)
>>> e = {1, 3, 7}
>>> a[1]
5
>>> a[-1]
98
>>> b[1]
'e'
>>> b[-1]
'o'
>>> c[1]
30
>>> c[-1]
10
>>> d[1]
12
>>> d[-1]
13
>>> e[1]
Traceback (most recent call last):
  File ".../pydevconsole.py", line 364, in runcode
    coro = func()
  File "<input>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

1.5.9 Mutability of Collections

Some composed types are mutable; some are not. Of the ones we have seen, `list` is mutable, while `tuple` and `string` are not.

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.

Listing 1.5.10

```
>>> a = [1, 2, 5, 7]
>>> print(a)
[1, 2, 5, 7]

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

>>> a[3] = b
>>> print(a)
[[100, 200], 2, 5, [100, 200]]

>>> a[0][1] = 333
>>> print(a)
[[100, 333], 2, 5, [100, 333]]
```

As is seen this the previous example, the Python language allows a program to modify data. If data is modified, then all places that data is referenced (rather than copied) can be accidentally or intentionally effected. This is both a dangerous and powerful feature.

In this course we will avoid modifying data; however sometimes it is unavoidable or not worth the effort to write clean code without hidden side-effects.

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

1.5.10 Building Strings

There are several ways to build strings. Use the one which best fits your use case.

1. String concatenation

   The first we look at is simple concatenation. Use the `+` operator between strings, either literal strings or variables, to concatenate them into a single string.

Listing 1.5.11

```
>>> s1 = "hello"
>>> s2 = "world"
>>> s1 + s2
'helloworld'

>>> s1 + ' ' + s2
'hello world'
```

2. Stream repetition

   Use the `*` character to repeat a string.

   Listing 1.5.12

```
>>> "hello " * 4
'hello hello hello hello '

>>> "hello " * 0
''
```

3. String interpolation

   Begin a string with the `f` character to indicate that expressions between braces {} should be replaced with the value of the expression.

   Listing 1.5.13

```
>>> s1 = "hello"
>>> s2 = "world"
>>> f"{hello} {world}"
'hello world'
```

1.6 Type Annotations

The `typing` package allows programmers to annotate functions with *type decorations*.

Listing 1.6.1

```
def f(n:int) -> List[int]:
  return [i for i in range(n)]
```

These type annotations are completely optional. Their purpose is to help humans and analyzers to understand your code. In the example above the function `f` accepts an argument `n`. The type annotation indicates that the programmer intends the caller to pass an integer (object of type `int`) as argument. The annotation `-> List[int]` indicates that the programmer promises that the function will return an object of type `List[int]`.

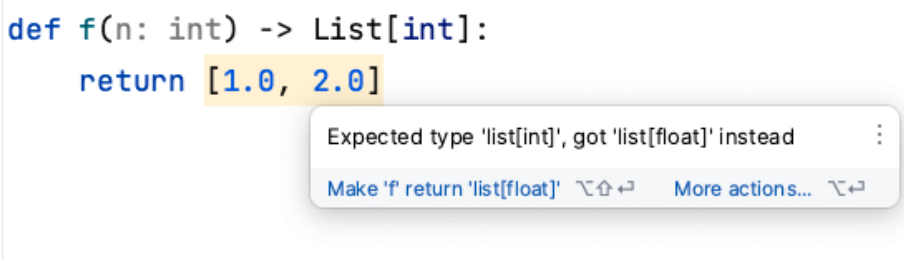
Code provided for the student in the class often comes decorated with type annotations. The annotations are intended to be helpful so that the student understand what kind of values a function will be passed, and to understand which kind of return value to generate.

The student is not required to annotate new functions.

It is important to remember that the Python language does not inforce these promises. You may still call the function with some object of type other than `int`, in which case the program *might* still work, might return some value other than `List[int]` or might even encounter a run-time error.

Often the static analyzier built into PyCharm can warn you about suspicious code if it recognizes inconsistency in the type annotations vs. the actual code.

In the example here, PyCharm is warning that the function is returning a list of floats, `List[float]` rather than a list of integers, `List[int]` as promised.



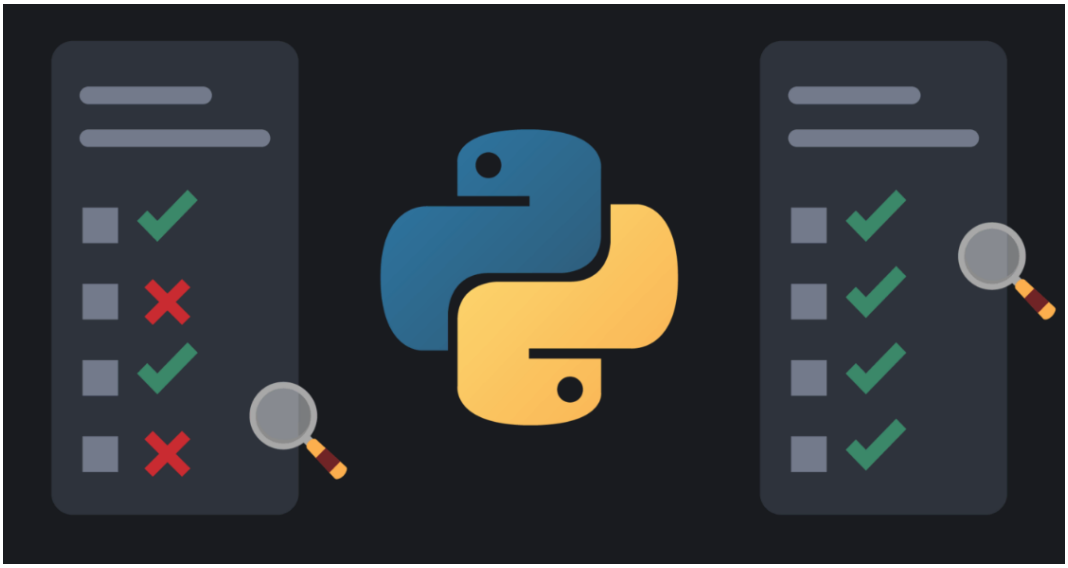
The typing system is very good but imperfect. Sometimes it is not possible to correctly describe the type of a function, because Python supports dynamic behavior. Nevertheless, the typing system, although imperfect, is extremely useful.

Some important type annotation syntax

- `List` — specifies a list of objects, all of some type: `List[int]`, `List[List[int]]`.

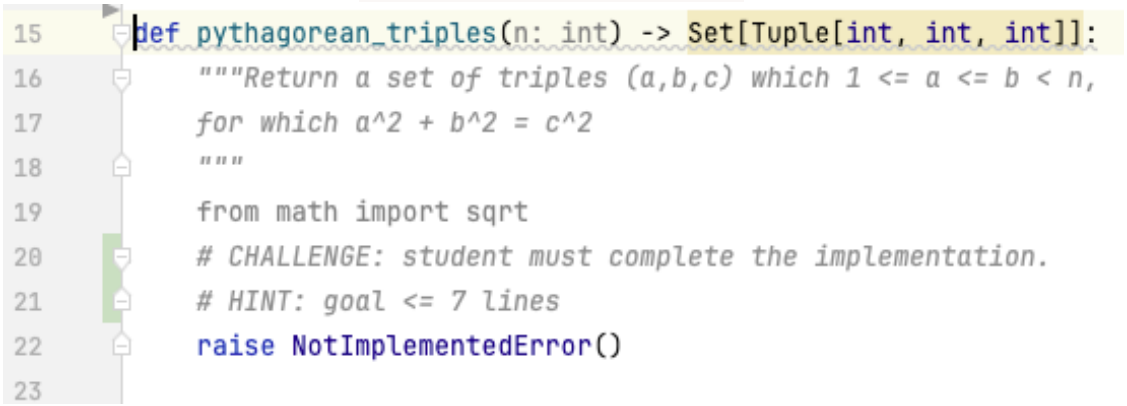
- `Tuple` — specifies a tuple of objects of specified types: `Tuple[int,str]`.
- `Set` — specifies a set of objecs of a specified type: `Set[int]`
- `Optional` — Specifies that a value may be either a specified type or the value `None`. `Optional[int]` specifies a value is either an `int` or it is the value `None`.
- `Union` — Specifies that a value may be any one of a specified set of types. `Union[int,float,string]`.
- `Callable` — Specifies the value of a function. You must specify the types of each input parameter, and the type of output value: `Callable[[int,string],string]`, `Callable[[int],Optional[int]]`
- `TypeVar` — Allows a type designator to reference the same unknown type multiple times. *E.g.*, `Tuple[A,A,int]` specifies a tuple of three elements, the 3rd of which is an `int`, the first and second may be any type, but the same type.

1.7 Testing



1.7.1 Code Templates

Most of the programming exercises the student will encounter in this course are as follows: The user must complete the implementation of a *stub* function such as `pythagorean_triples`.



Listing 1.7.1

```
def pythagorean_triples(n: int) -> Set[Tuple[int, int, int]]:
    """Return a set of triples (a,b,c) which 1 <= a <= b < n,
    for which a^2 + b^2 = c^2
    """
    from math import sqrt
    # CHALLENGE: student must complete the implementation.
    # HINT: goal <= 7 lines
    raise NotImplementedError()
```

The student must replace `raise NotImplementedError()` with valid Python code which implements the description in the documentation string. Use any editor or IDE you prefer, but during the lectures we will use PyCharm. Some students prefer Microsoft Visual Studio, also called VS Code.

Each template to be completed has several parts.

- A function definition with arguments already provided: `def pythagorean_triples(n)`
- Type annotations such as `(n: int) -> Set[Tuple[int, int, int]]`. The annotations are provided to help the student understand which types the function will receive and which type it needs to return to pass the

associated tests. Students are not required to include type annotations on any new function written to complete the exercises. The annotations are only provided to be helpful.

- A so-called *docstring* which describes what the completed function is expected to compute.
- In some cases some imports or asserts may be already given `from math import sqrt`. The imports (or assertions) serve as hints.
- A `Hint` indicating how many lines of code is expected to solve the assignment. You will be graded on number of lines of code. If your solution has 100 lines while the comment indicates `HINT goal <= 7 lines`, then there is something fundamental you probably do not understand. You should solve the assignment with fewer lines. There is a test case in place which will fail if you take more than twice the recommended number of lines to solve the assignment.
- `raise NotImplementedError()` If you simply run the tests without modifying the code, then this command will cause the test to fail. This failure will help you find which functions you need to complete to finish the assignment.

To know whether the code is correct, run the tests which are provided in the `tests/` directory. There is one testing file per assignment file. The code you have to write in the file `looping.py` can be tested by running the tests in the file `looping_test.py`.

1.7.2 Anatomy of Tests

The student has full access to all the test cases. If a test fails, the student is responsible for reading the testing code to understand what is failing, and using the debug environment of the IDE to help repair the code in question. This student is allowed (encouraged) to write additional tests, following the pattern shown, to help debug this implementation.

In the following code extract there are three testing functions (methods), `test_code_check`, `test_primes_empty`, and `test_primes_0`.

The first test is explained in Section D.4. The tests and test the function `find_primes` (which you must implement in `looping.py`) test your implementation against several fixed return values. Other tests in the test suite test your function against machine generated values.

Listing 1.7.2

```
1  class LoopingTestCase(AlgTestCase):
2      def test_code_check(self):
3          self.code_check(["looping.py"], import_exception)
4
5      def test_primes_empty(self):
6          self.assertEqual(find_primes(1, 1), [])
7          self.assertEqual(find_primes(1, 2), [])
8          self.assertEqual(find_primes(2, 2), [])
9
10     def test_primes_0(self):
11         self.assertEqual(find_primes(2, 3), [2])
12         self.assertEqual(find_primes(2, 4), [2, 3])
13         self.assertEqual(find_primes(2, 11), [2, 3, 5, 7])
```

1.7.3 Testing in PyCharm

Figure 1.1 shows an excerpt from the file `looping_test.py` as seen in PyCharm. Notice the small green go-triangles in the left margin. You may run a single test, such as `test_code_check` by pressing the green go-triangle (or circle) beside that function. You may run the entire suite of tests associated with `looping.py` by pressing the go-rectangle beside `LoopingTestCase` on line 20 in Figure 1.1.

Alternatively you may run the tests using the pop-up menu over the file name, as shown in Figure 1.2.

1.7.4 Testing in VS Code

Figure 1.3 shows an excerpt from the file `looping_test.py` as seen in VS Code. Notice the small green go-triangles (or sometimes green circles as in the figure) in the left margin in both IDEs. You may run a single test, such as `test_code_check` by pressing the green go-triangle (or cir-

```
20  class LoopingTestCase(AlgTestCase):
        Jim Newton
21      def test_code_check(self):
22          self.code_check( pyfiles: ["looping.py"], import_exception)
23
        new *
24      def test_primes_empty(self):
25          self.assertEqual(find_primes( n: 1,  m: 1),  second: [])
26          self.assertEqual(find_primes( n: 1,  m: 2),  second: [])
27          self.assertEqual(find_primes( n: 2,  m: 2),  second: [])
28
        Jim Newton
29      def test_primes_0(self):
30          self.assertEqual(find_primes( n: 2,  m: 3),  second: [2])
31          self.assertEqual(find_primes( n: 2,  m: 4),  second: [2, 3])
32          self.assertEqual(find_primes( n: 2,  m: 11),  second: [2, 3, 5, 7])
33
```

Figure 1.1: Test cases displayed in PyCharm

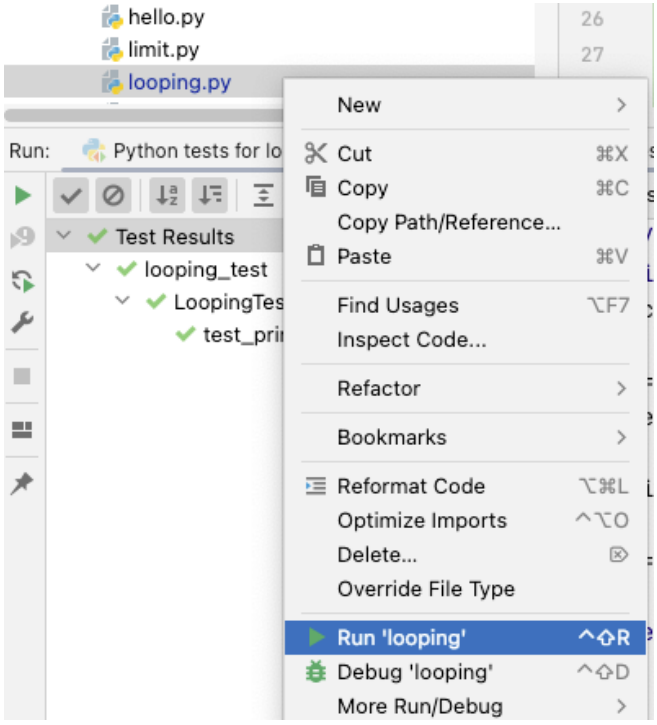


Figure 1.2: Run all tests in a file with pop-up menu on the file name.

```
19
20  class LoopingTestCase(AlgTestCase):
21      def test_code_check(self):
22          self.code_check(["looping.py"], import_exception)
23
24      def test_primes_empty(self):
25          self.assertEqual(find_primes(1, 1), [])
26          self.assertEqual(find_primes(1, 2), [])
27          self.assertEqual(find_primes(2, 2), [])
```

Figure 1.3: Test cases displayed in VS Code

cle) beside that function. You may run the entire suite of tests associated with `looping.py` by pressing the go-rectangle beside `LoopingTestCase` on line 20 in Figure 1.3.

Figure 1.4: VS Code Test Explorer

VS Code provides Test Explorer, accessible with the icon  . Pressing that Icon, should show an interface as shown in Figure 1.4. Press the **Run Test** to run the tests in one of the testing files.

1.7.5 Testing in Command Line

Still another way to run the tests is using `python` or `python3` from the command line. Make sure the current directory is the `algebra` directory which contains the subdirectory `tests`. Run the tests independent of an IDE as shown here.

Listing 1.7.3

```
shell> ls
bin algebra tests

shell> python3 -m pytest tests/looping_test.py
==================== test session starts ====================
platform darwin -- Python 3.9.6, pytest-7.2.1, pluggy-1.0.0
rootdir: .../bcs-algebra-1-student/algebra
collected 18 items

tests/looping_test.py ..................                [100%]

==================== 18 passed in 0.17s =====================

shell>
```

1.8 Programming Challenges

The main purpose of this exercise is to understand how to make homework submissions.

Complete the exercises in the file `algebra/hello.py` and test with the tests in the file `tests/hello_test.py`.

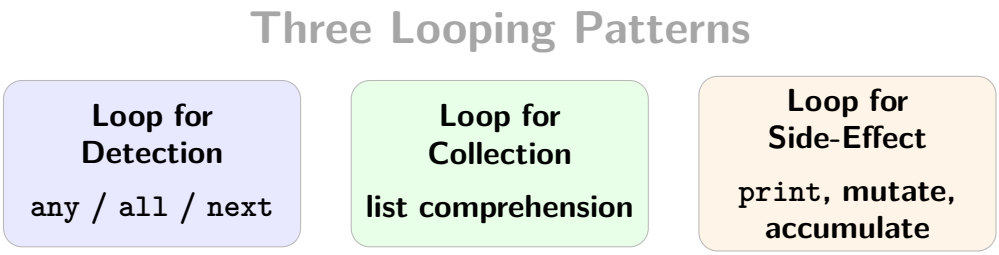
If your submitted code fails the test named `test_code_check`, that means you code does not meet the *coding standard*. You'll probably need to reformat your code. This topic is fully explained in Section D.4 on page 298.

Chapter 2

Logic, Sets, and Looping

Chapter Objectives

- Evaluate propositions and predicates using conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\implies$), including their truth tables.
- State and apply De Morgan's theorem for both logic and sets.
- Translate the universal quantifier $\forall$ into `all`, the existential quantifier $\exists$ into `any`, and "find a witness" into `next`.
- Classify a looping problem as loop-for-detection, loop-for-collection, or loop-for-side-effect, and choose the appropriate Python construct.
- Implement set operations (filter, subset) using lsets and avoid Python's built-in `set()` for collections that may contain unhashable elements.
- Implement the `predicates` programming assignment.


Three Looping Patterns
Loop for Detection
any / all / next
Loop for Collection
list comprehension
Loop for Side-Effect
print, mutate, accumulate

Logical Connectives

p	q	$p \wedge q$	$p \vee q$	$\neg p$	$p \implies q$
T	T	T	T	F	T
T	F	F	T	F	F
F	T	F	T	T	T
F	F	F	F	T	T

2.1 Logic

Definition 2.1.1 (*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 2.1.2 (*predicate*)

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 *unary* predicates are *prime(x)*, or $x > 42$. A *unary* predicate is a predicate with exactly one argument. For each of these, given one 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?

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.

A *binary* predicate has two arguments. A binary predicate is also called a *relation* which we will cover in more detail in Chapter 3. And example of a binary predicate is $x < y$. Given a value for x (say 3) and a value for y (say 7), we can ask: is $3 < 7$ and get a truth value.

2.1.1 Conjunction

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 $g(n) \wedge h(n)$.

2.1.2 Disjunction

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.

2.1.3 Negation or Complement

Given a proposition, p, we say $\neg p$ is a proposition which is true if p is false, and is false if p is true. We say p is the (logical) complement of $\neg p$, and we say $\neg p$ is the (logical) complement of p.

Likewise we may *complement* a predicate. If $prime(x)$ is a predicate which evaluates to true if its argument is a prime number, then

$\neg prime(x)$ is true whenever $prime(x)$ is false, and false whenever $prime(x)$ is true.

2.1.4 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: $n \in N \implies n = 0 \vee n \geq 1$.

By $p \iff q$ we mean (by definition) that $(p \implies q) \wedge (q \implies p)$.

If an implication $p \implies q$ is true, then its contrapositive $(\neg q \implies \neg p)$ is necessarily true. Likewise, if the contrapositive is true, then the original proposition is also true. We we can state the following implication about implications.

$$(p \implies q) \iff (\neg q \implies \neg p)$$

However, it is not the case that a propsition being trues implies its converse is true.

$$\neg((p \implies q) \implies (q \implies p))$$

or equivalently

$$(p \implies q) \not\iff (q \implies p)$$

The implication operator can be expressed in terms of complement and disjunction.

$$(p \implies q) \iff (\neg p \vee q)$$

2.1.5 Truth Tables

We may represent Boolean functions such as conjunction and disjunction using a *truth table*

p	q	$p \vee q$ $q \vee p$	$p \wedge q$ $q \wedge p$	$\neg p$	$\neg q$	$p \implies q$ $\neg p \vee q$	$\neg q \implies \neg p$ $q \vee \neg p$	$q \implies p$ $p \vee \neg q$	$p \iff q$ $q \iff p$ $q = p$
T	T	T	T	F	F	T	T	T	T
T	F	T	F	F	T	F	F	T	F
F	T	T	F	T	F	T	T	F	F
F	F	F	F	T	T	T	T	T	T

2.1.6 Commutative and Associative

It is straightforward to verifies from the truth table that intersection, disjunction, and double implication are commutative; however implication is not commutative.

$$\begin{aligned}(p \wedge q) &\iff (q \wedge p)\\ (p \vee q) &\iff (q \vee p)\\ (p \iff q) &\iff (q \iff p)\end{aligned}$$

With a bit of work we can show that these three operators are associative.

$$\begin{aligned}((p \wedge q) \wedge r) &= (p \wedge (q \wedge r))\\ ((p \vee q) \vee r) &= (p \vee (q \vee r))\\ ((p \iff q) \iff r) &= (p \iff (q \iff r))\end{aligned}$$

Is implication associatve? Does $(p \implies q) \implies r$ equal to $p \implies (p \implies r)$? A simple counter example suffices to show that implication is NOT associative.

$$\begin{aligned}(F \implies F) \implies F \quad &= T \implies F \quad = F\\ F \implies (F \implies F) \quad &= F \implies T \quad = T\end{aligned}$$

2.1.7 Quantifiers

First we look at the *universal quantifier*. $\forall$ specifies a proposition (whose value is either true or false) for which a given predicate is true for every element of a set.

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

$$(\forall x \in A, P(x)) \iff (x \in A \implies P(x)) \tag{2.1}$$

Equation (2.1) states an equivalence between implication and universal quantification. Whenever, you see $\forall$ you may rewrite it as $\implies$, if that helps to understand or prove a proposition.

Next we look at the *existential quantifier*. $\exists$ 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.

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

$$(\exists x \in A, P(x)) \iff \neg(x \in A \implies \neg P(x)) \tag{2.2}$$

Equation (2.2) states an equivalence between implication and existential quantification. Whenever, you see $\exists$ you may rewrite it as $\implies$, if that helps to understand or prove a proposition.

$\forall$ 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.

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

2.1.8 De Morgan's Theorem for Logic

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

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

Activity 1

Do: Can we use De Morgan's theorem to show that

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

is true independent of p?

Discuss: Explain

2.2 Sets

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?

2.2.1 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.

If x is some *object* (whatever that means) and A and B are sets, then we can ask whether x is an element of A and whether A is a subset of B.

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

There is a special set denoted $\emptyset$, called the *empty set*. The proposition $x \in \emptyset$ is true regardless of x, and $\emptyset \subset A$ is true for any set A.

New sets may be defined in terms of existing sets. Given two sets, A and B, the following are also sets.

- Union: $A \cup B$, the superset of A and of B which contains all the elements of A and all the elements of B and nothing else.
- Intersection: $A \cap B$, the subset of A and of B which contains all the elements common to A and B and nothing else.
- Complement: $A \backslash B$ or $\overline{B}$, the subset of A which contains no element of B, but all other elements of A and nothing else.
- Cartesian product: $A \times B$, the set of ordered pairs (x, y) where $x \in A$ and $y \in B$.

Given a predicate p, we can define a new set of all the elements of a given set for which the predicate is true. This is called a *set compre-*

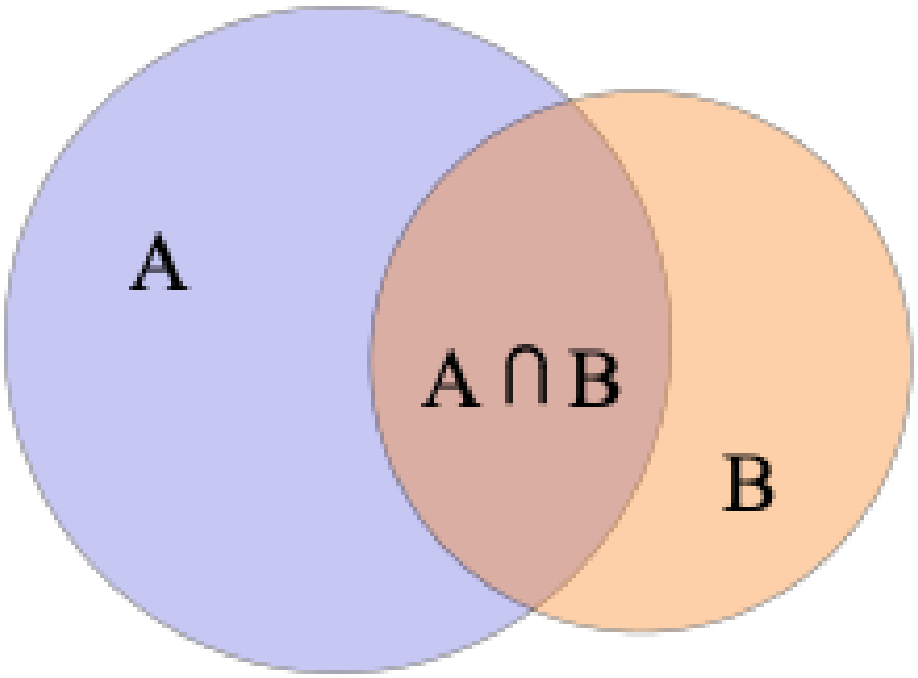


Figure 2.1: By Svjo - Own work, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=20749188`

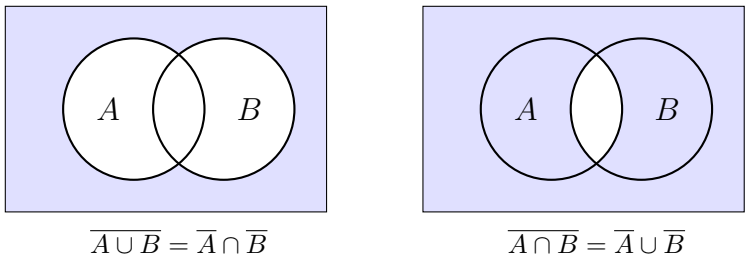


Figure 2.2: Venn Diagrams of De Morgan's Theorem

hension and is denoted: $\{x \in A \mid p(x)\}$. As a simple example suppose $A = \{1, 2, 3, 4, 5, 6, 7\}$, then

$$\{x \in A \mid even(x)\} = \{2, 4, 6\}\,.$$

Finally, every set has at lest one subset, $\emptyset$ and all non-empty sets have another subset, the set itself. The set of all subsets of a given set is called its *power set*: $\{s \mid s \subset A\}$.

2.2.2 Set Comprehension

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

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

or more concisely

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

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. *I.e.*, we filter a set by a predicate.

2.2.3 De Morgan's Theorem for Sets

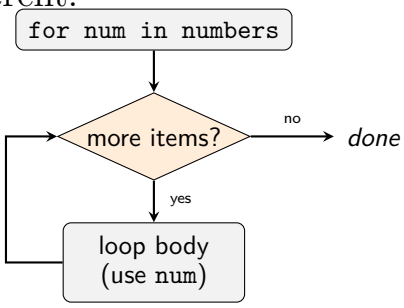
De Morgan's Theorem can be succinctly as $\overline{A \cup B} = \overline{A} \cap \overline{B}$, and dually $\overline{A \cap B} = \overline{A} \cup \overline{B}$.

De Morgan's Theorem basically says that the complement of the union is the intersection of the complements and the complement of the

intersection is the union of the complements. These two equalities can be illustrated as in Figure 2.2.

2.3 Looping

In this section we look at the connection between looping in the Python language and mathematical operations such as comprehensions and quantifiers. These two representations are very similar even if the syntax is different.



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 there may be multiple loops running concentrically. Loops may also contain conditions to skip certain iterations.

2.3.1 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? $\exists a, 0 \leq a < 10, a \equiv 1 \mod 3$

Listing 2.3.1

```
# (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 readability
any(a % 3 == 1
    for a in range(10))
```

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

Listing 2.3.2

```
# (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.

Listing 2.3.3

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

In Python, `any` implements the existential quantifier ∃ and `all` implements the universal quantifier ∀ from the Quantifiers subsection above. The `next` function goes further: instead of just answering "does a witness exist?", it returns the witness itself—the concrete value that made `any` return `True`.

Be careful of the two following expressions which may seem equivalent but are subtly different.

Listing 2.3.4

```
# (for every?) is something true for all elements in a list
all(a % 2 == 0
    for a in [2, 6, 4, 12])
```

The `all` quantifier in 2.3.4 evaluates to `True` if all the integers in the given list `[2, 6, 4, 12, -34]` are even, which they are.

Listing 2.3.5

```
all(a
    for a in [2, 6, 4, 12]
    if a % 2 == 0)
```

However, `all` quantifier in 2.3.5 also evaluates to `True` for a different reason. The code in 2.3.5 examines all the even elements of the given list, and asserts that they all have a Boolean *true* value. In Python some values are Boolean *false* other than the explicit `False` object.

For example, `None` is considered *false* as well as empty-string `""`, they empty list `[]`, and in particular the integer `0` is considered false.

If we change the list in 2.3.5 from `[2, 6, 4, 12]` to `[0, 2, 6, 4, 12]`, we will find that the `all` quantifier returns `False`, because no longer are all the even values *false*.

Listing 2.3.6

```
# returns False because 0 is false
all(a
    for a in [0, 2, 6, 4, 12]
    if a % 2 == 0)
```

2.3.2 Using `next`

Recall that `any` and `all` return a Boolean value. Thus if `any` return `True`, you cannot know which *satisfying* value made the predicate `True`. Likewise, if `all` returns `False`, you cannot know which *counter-example* value made the predicate `False`. If you need to know which value satisfies, or which value is a counter-example, you may use `next`.

The expression `any(a > 2 for a in range(10))` returns `True`, while `next(a > 2 for a in range(10))` return `3`.

Likewise, `all(a > 2 for a in range(10))` returns `False`, while `next(not (a > 2) for a in range(10))`, returns `0` as 0 is the first element of the range which fails to satisfy $a > 2$. Note, that we must negate the predicate to translate `all` to `next`.

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.

Listing 2.3.7

```
# 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:

Listing 2.3.8

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

If you want to avoid an exception being raised when `next` fails to find an element, you may provide a default value, as a second argument to `next`. To provide a second argument, it us usally necessary (for syntax reasons) to enclose the first arument in parentheses.

In the following example, `next` returns 0 (the default value) because it fails to find an a between 0 and 99 such that $a^2 < 0$.

Listing 2.3.9

```
  # find the first element which satisfies a condition
  next((a
        for a in range(100)
        if a*a < 0),
       0  # the default value
      )
```

The default value will be returned in the case `next` fails to find an element. It is the responsibility of the programmer to provide a distinguishing default value. For example, in the following code, if `next` returns `12`, you cannot know if it is because `f(12) > 0` or, because `f(a)` failed to return `0` over the given range.

Listing 2.3.10

```
  # find the first element which satisfies a condition
  next((a
        for a in range(100)
        if f(a) > 0),
       12)
```

2.3.3 Revisiting De Morgan's Theorem

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`?

Listing 2.3.11

```
  # 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])
```

2.3.4 Composing Quantifiers

We want to write a function which takes to arguments, `a` and `b`. The first function `contains1` returns `True` if all the strings in list `a` are contained in some string in list `b`. The second function `contains2` return `True` if some string in list `b` is contained in all strings in list `a`.

Let's suppose we have a predicate `p(x,y)` which decides whehter the string `x` is contained in the string `b`. Mathematically `contains1` can be represented as

$$\forall x \in a \; \exists y \in b, \; p(x, y)\,,$$

while `contains2` can be represented as

$$\exists y \in b \; \forall x \in a, \; p(x, y)\,.$$

Let's implement these in Python.

Listing 2.3.12

```
def contains1(a, b) -> bool:
  return all(any(x in y
                 for y in b)
             for x in a)

def contains2(a, b) -> bool:
  return any(all(x in y
                 for x in a)
             for y in b)
```

Now let's implement a function which returns the Boolean complement of `contains2`.

$$\begin{aligned}\neg(\exists y \in b\ \forall x \in a,\ p(x,y)) &= \forall y \in b\ \neg(\forall x \in a,\ p(x,y)) \\ &= \forall y \in b\ \exists x \in a,\ \neg p(x,y)\end{aligned}$$

So we could implement the function `not_contains2` as follows, by changing `all` to `any`, `any` to `all`, and negating the predicate `x in y` to `x not in y`.

Listing 2.3.13

```
def not_contains2(a, b) -> bool:
  return all(any(x not in y
                 for x in a)
             for y in b)
```

2.3.5 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.

Listing 2.3.14

```
  # 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.

Listing 2.3.15

```
  # 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.

Listing 2.3.16

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

Section 2.4 presents several programming assignments. Several of these problems are similar in that they as you to collect the set of tuples in a given range which satisfy a given predicate. Let's look at some similar problems here to introduce the intended technique for implementating the functions.

First, write a function to generate the list of order pairs (a, b) where $0 \leq a \leq b < n$, where n is given. The template of the function is the following.

Listing 2.3.17

```
def gen_pairs(n):
  return ...
```

Given n we wish to compute and return something. We need to loop for collection.

Listing 2.3.18

```
def gen_pairs(n):
  return [ ... ]
```

We'd like to generate a list of ordered pairs as a tuple.

Listing 2.3.19

```
def gen_pairs(n):
  return [(a, b) ... ]
```

We'd like all (a, b) for a and b in certain intervals.

Listing 2.3.20

```
def gen_pairs(n):
  return [(a, b)
          for a in range(...)
          for b in range(...)
         ]
```

We are asked to assure that $0 \leq a \leq b < n$. Thus a needs to vary between 0 and n, including 0 but excluding n; so we can use `range(0,n)` which includes its lower bound and excludes its upper bound.

Listing 2.3.21

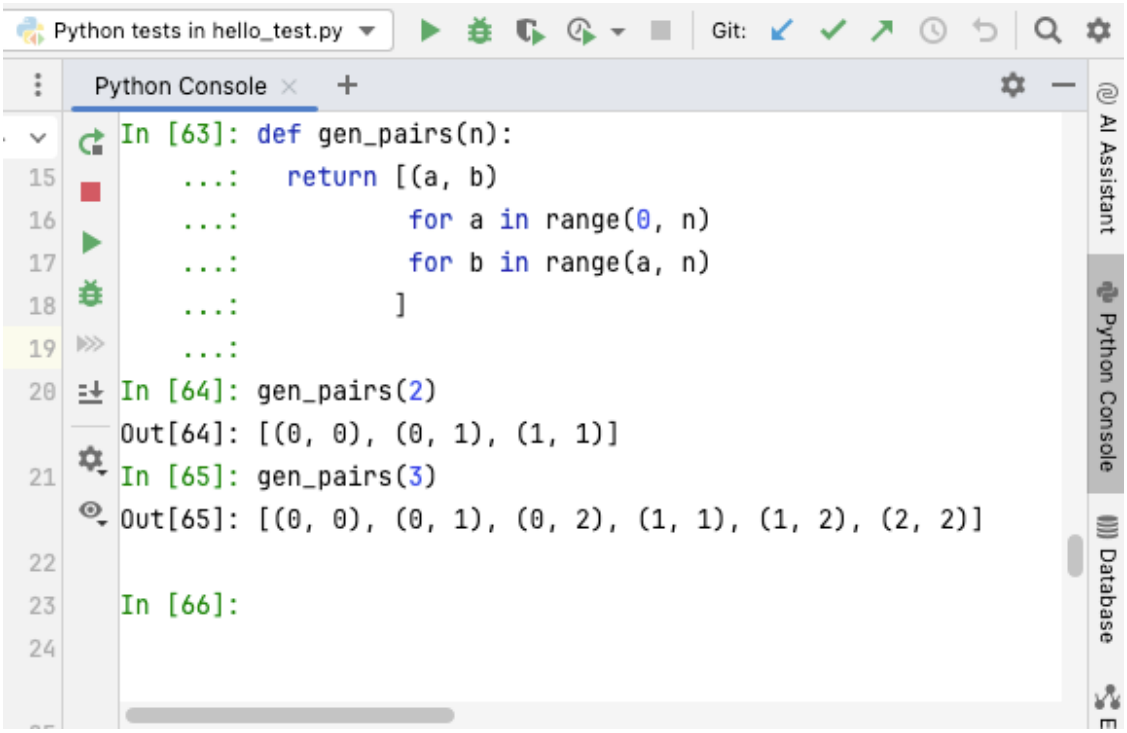
```
def gen_pairs(n):
  return [(a, b)
          for a in range(0, n)
          for b in range(...)
         ]
```

Next, b should vary between a and n, including a and excluding n; so we can use `range(a, n)`.

Listing 2.3.22

```
def gen_pairs(n):
  return [(a, b)
          for a in range(0, n)
          for b in range(a, n)
         ]
```

We can test `gen_pairs` in the IntelliJ Python Console.



The assignments in Section 2.4 have an additional requirement. You need to collect only *some* of the tuples in the range, *i.e.* the tuples which satisfy a predicate. How can we modify the functionp `gen_pairs` to only collect (a, b) if a is a factor of b. In python `b % a == 0` is true if a divides b with a 0 remainder. So we can modify the function as follows. Note, that we also wish to avoid division by 0, so we let's just avoid 0 altogether, and start with `range(1,n)`

Listing 2.3.23

```
def gen_pairs(n):
  return [(a, b)
          for a in range(1, n)
          for b in range(a, n)
          if b % a == 0
         ]
```

If we test this by calling `gen_pairs(6)` we get the following correct return value.

Listing 2.3.24

```
gen_pairs(6)
=> [(1, 1),
    (1, 2),
    (1, 3),
    (1, 4),
    (1, 5),
    (2, 2),
    (2, 4),
    (3, 3),
    (4, 4),
    (5, 5)]
```

In the homework assignments you'll need to find the correct ranges, and find the correct conditionals.

2.3.6 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`:

Listing 2.3.25

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

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

Listing 2.3.26

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

2.3.7 Examples

In Listing 2.3.27 we compute the set of integers which appear a target number of times in a given list.

Listing 2.3.27 (*Count Occurances*)

```
from typing import List, Set

def count_occurrences(data:List[int],
                      item:int
                     ) -> int:

    return len([x for x in data
                if x == item])

def filter_by_frequency(data:List[int],
                        freq:List[int]
                       ) -> Set[int]:

    return {x for x in data
            if count_occurences(data,x) == freq}
```

In listing 2.3.28 we compute whether a given Boolean operator such as conjunction or implication is commutative or associative.

Listing 2.3.28 (*Testing Associativity*)

```
from typing import Optional, Tuple

def implies(p:bool,q:bool) -> bool:
    return (not p) or q

def commutative(op) -> bool:
    return all(op(p,q) == op(q,p)
               for p in [True, False]
               for q in [True, False])

def counter_example_commutative(op):
    return next(((p,q)
                 for p in [True, False]
                 for q in [True, False]
                 if op(p,q) != op(q,p)),
                 None)

def associative(op) -> bool:
    return all(op(p,op(q,r)) == op(op(p,q),r)
               for p in [True, False]
               for q in [True, False]
               for r in [True, False]
    )
```

The `commutative` and `associative` functions above are Loop-for-

Detection (Section 2.3.1): `all` checks a predicate across every combination of Boolean inputs, and a single `False` is enough to return `False`. The nested `for p ... for q ...` expresses $\forall p, q \in \{\texttt{True}, \texttt{False}\}$, exactly as in the quantifier notation from the Logic section above.

2.3.8 Representing Sets in Python

The Python language has a limited, built-in, implementation of `set`.

Listing 2.3.29

```
x = {1, 2, 3, 2, 3, -1}
1 in x
# => returns True
4 in x
# => returns False
```

Only certain types of elements are compatible with Python sets.

Listing 2.3.30

```
x = {[1, 2], [3, 2], [3, -1]}

Traceback (most recent call last):
  File ".../python/site-packages/IPython/core/interactiveshell
    .py", line 3526, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-10-8385e23981d1>", line 1, in <module>
    x = {[1, 2], [3, 2], [3, -1]}
TypeError: unhashable type: 'list'
```

Because Python sets are not compatible with some data types, we will implement the semantics of sets using Python lists. The homework exercise `lset.py` will contain the complete implementation.

The first challenges are to implement the functions `find_duplicates` and `is_lset` according to the docstrings provided. The `find_duplicates` is probably considered the most difficult, and make take 5 to 10 lines of code. All the rest of the functions in `lset.py` are very small, *i.e.* very few number of lines each.

As an example of how to treat lists as sets, let's implement the function `lset_filter` using a list comprehension and `lset_subset` using the `all` quantifier. The `lset_filter` function will take an `lset` a

Boolean predicate, and will return the list representing the `lset` of elements in the given set which satisfy the predicate. The `lset_subset` will take two `lset` arguments and return an `lset`.

Listing 2.3.31

```
def lset_filter(s, p):
  return [x in s
          if p(x)]

def lset_subset(s, t):
    return all(x in t
               for x in s)
```

`lset_subset` is a Loop-for-Detection (Section 2.3.1): `all(x in t for x in s)` asks $\forall x \in s, x \in t$, which is precisely the set-inclusion definition $s \subseteq t$. `lset_filter` is a Loop-for-Collection: it scans every element of `s` and retains those satisfying predicate `p`, matching the set-comprehension notation $\{x \in s \mid p(x)\}$ from Section 2.2.

2.4 Programming Challenges

Complete the exercises in the file `algebra/predicates.py` and test with the tests in the file `tests/predicates_test.py`.

Complete the exercises in the file `algebra/looping.py` and test with the tests in the file `tests/looping_test.py`. Try to solve the challenges using `any`, `all`, `next`, `for`.

1. `find_primes` Collect all prime numbers between n and m.

2. `pythagorean_triples`: Collect all Pythagorean triples (a, b, c) of integers between 1 and n, such that $1 \leq a \leq b < n$, $1 \leq c < n$, and $a^2 + b^2 = c^2$. This exercise is discussed further in Section 1.7.1.

3. `sum_cubes`: Collect all solutions to $a^3 + b^3 + c^3 = 1$ for a, b, c in range of $-n \leq a \leq b \leq c < n$.

4. `linear_diaphantine`: Linear Diophantine Equations: Given integers a, b, c, and n, find all integer solutions ($0 \leq x < n$ and $0 \leq y < n$) to the equation $ax + by = c$. *E.g.*, $2x + 4y = 28$ has $x = 12$, $y = 1$ as solution but also $x = 2$, $y = 6$.

Complete the exercises in the file `algebra/lset.py` and test with the tests in the file `tests/lset_test.py`.

Chapter 3

Relations

Chapter Objectives

- Define a relation on a set A and state the nine properties: reflexive, irreflexive, symmetric, asymmetric, antisymmetric, transitive, connected, strongly connected, and well-founded.
- Implement each relation-property test in Python using `all`, `any`, and the `implies` helper.
- Implement `uniquely` as the Python realization of the unique-existence quantifier $\exists!$, and use it to implement `is_function_relation`.
- Define equivalence relation and partial order, and give examples of each.
- Distinguish injective, surjective, and bijective function relations and implement `is_injection`, `is_surjection`, and `is_bijection`.
- Implement the `relation` programming assignment.

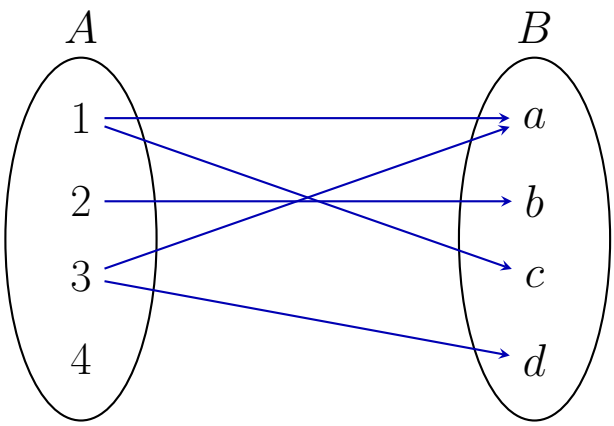


Figure 3.1: A relation $R \subseteq A \times B$ represented as arrows

Definition 3.0.1 (*relation*)

A *relation*, R, between sets A and B (from A to B) is a possibly empty subset of $A \times B$. If (a, b) is in the relation, $(a, b) \in R$ we may write aRb. In the case that $A = B$ we say that R is a *relation on* A.

We use symbols such as $<$, $=$, $\simeq$, $\subset$ to denote more familiar relations.

Let's look at some examples. Suppose $A = \{1, 2, 3\}$ and $B = \{7, 8, 9\}$. The set $A \times B$ is the set of all ordered pairs (x, y) where $x \in A$ and $y \in B$:

$$A \times B = \{(1,7), (2,7), (3,7), (1,8), (2,8), (3,8), (1,9), (2,9), (3,9)\} \tag{3.1}$$

There are 9 elements of $A \times B$ so there are 2^9 subsets including $\emptyset$ and $A \times B$ itself. Every subset of $A \times B$ is a relation from A to B; *e.g.*,

$$\begin{aligned} \{(1,7), (2,7), (3,7)\} &\subset A \times B \\ \{(1,7), (2,7), (3,7), (1,8), (2,8), (3,8)\} &\subset A \times B \\ \{(2,7), (3,7), (2,8), (3,8), (1,9)\} &\subset A \times B \\ \{(3,7), (2,8), (1,9), (3,9)\} &\subset A \times B \end{aligned}$$

The Python function `is_relation` computes whether the given set of tuples is a relation.

Listing 3.0.2 (*Detecting a Relation*)

```
def is_relation(a: Set, b: Set, r: Set[Tuple]) -> bool:
    ...
```

3.1 Implementing Relations with `all` / `any`

Not all relations are *useful* or *meaningful*. There are many kinds of relations which are important and are named.

Definition 3.1.1 (*reflexive*)

A relation R on A is called *reflexive* if

$$a \in A \implies (a, a) \in R\,.$$

As an example, consider $A = \{1,2,3\}$; any subset of $A \times A$ which is a superset of $\{(1,1),(2,2),(3,3)\}$ is a reflexive relation. So $\{(1,1),(2,2),(3,3),(1,2),(2,3)\}$ is reflexive, and $\{(3,3),(1,2),(2,3)\}$ is not.

Recall, Eq (2.1) defined a correspondence between $\forall$ and $\implies$. Using Eq (2.1), another way to state the implication in Definition 3.1.1, is as

$$\forall a \in A, (a,a) \in R \tag{3.2}$$

Restating $a \in A \implies (a,a) \in R$ as 3.2 is important, because there is a special syntax, `all`, in Python to represent $\forall a \in A, (a,a) \in R$.

Listing 3.1.2 (*Python reflexive relation*)

```
def is_reflexive(A, R):
  return all((a,a) in R
             for a in A)
```

`is_reflexive` is a Loop-for-Detection (Section 2.3.1): `all((a,a) in R for a in A)` asks $\forall a \in A, (a,a) \in R$, the exact mathematical definition of reflexivity. The same `all` + generator idiom is used throughout this chapter for every relation property expressed as a universal quantification.

Some examples:

Listing 3.1.3 (*Python example of* $\exists$)

```
is_reflexive({1,2,3}, {})
=> False

is_reflexive({1,2,3}, {(1,1),(2,2)})
=>  False

is_reflexive({1,2,3}, {(1,1),(2,2), (3,3)})
=>  True

is_reflexive({1,2,3}, {(1,1),(2,2), (3,3), (1,2)})
=>  True
```

Similar to `all` let's look at `any`. A logical expression such as $\exists a \in A, (a,a) \in \mathbb{R}$ can be expressed in python using the `any` keyword.

Listing 3.1.4 (*Python example of* $\exists$)

```
any((a,a) in R
    for a in A)
```

The student may want to review Section 2.3.1 for more explanation of `all` and `any`. Understanding the Python syntax is important for working the homework problems for this chapter, Section 3.9.

For example. The relation, $=$ is reflexive on $\mathbb{N}$. In particular for every $n \in \mathbb{N}$ we know that $n = n$. However, the relation $<$ is not reflexive on $\mathbb{N}$; it is not true that $n < n$.

Consider the power set of $\mathbb{N}$, (*i.e.*, the set of all subsets of $\mathbb{N}$), and consider the relation $\subseteq$. Every set is a subset of itself, so we have $A \subseteq A$ for all $A \subset \mathbb{N}$. So the subset relation is a reflexive relation.

3.2 Other Relations

Definition 3.2.1 (*irreflexive*)

A relation R on A is called *irreflexive* if

$$a \in A \implies (a, a) \notin R\,.$$

The relation $<$ on $\mathbb{N}$ is irreflexive because there is no natural number, m such that $m < m$. The not-equal $\neq$ relation is also irreflexive; $n \neq n$ is never true.

However the disjoint set relation $(A, B) \in R$ if $A \cap B = \emptyset$ is not irreflexive. For most sets, A we know that we know A is not disjoint with A, however there is one exception: the empty set: $\emptyset \cap \emptyset = \emptyset$.

To convert the definition of irreflexive to a Python expression, we must express it in terms of existential or universal quantifiers $(\exists, \forall)$.

$$\forall a \in A, (a, a) \notin R$$

The equivalent Python expression is

Listing 3.2.2 (*Python irreflexive relation*)

```
all((a,a) not in R
    for a in A)
```

Definition 3.2.3 (*symmetric*)

A relation R on A is called *symmetric* if

$$(a,b) \in R \implies (b,a) \in R\,.$$

The $=$ on $\mathbb{N}$ is symmetric because if $a = b$ then $b = a$. Likewise the $\neq$ relation is symmetric because $a \neq b \implies b \neq a$.

Another example of symmetric is the set-disjoint relation. If A is disjoint with B, (has no element in common), then B is disjoint with A.

We expressing the definition of symmetric as a universal quantifier:

$$\forall (a,b) \in R, (b,a) \in R\,,$$

which can be converted to Python as

Listing 3.2.4 (*Python symmetric relation*)

```
all((b,a) in R
    for a,b in R)
```

Definition 3.2.5 (*asymmetric*)

A relation R is called *asymmetric* if

$$(a,b) \in R \implies (b,a) \notin R\,.$$

An example of an asymmetric relation is $<$ on $\mathbb{N}$. Whenever $a < b$ then we know that $b < a$ is false.

The proposition in terms of universal quantifier is

$$\forall (a,b) \in R, (b,a) \notin R\,.$$

The Python expression is

Listing 3.2.6 (*Python asymmetric relation*)

```
all((b,a) not in R
    for a,b in R)
```

If we apply De Morgan's theorem to the proposition

$$\forall (a,b) \in R, (b,a) \notin R\,,$$

we obtain

$$\neg(\exists (a,b) \in R, (b,a) \in R)\,.$$

This dual expression and equivalent expression means that a relation R fails to by asymmetric, if we can find a counterexample, *i.e.*, a case when $(a,b) \in R$ and simultaneously $(b,a) \in R$. We can also express this dual expression in Python.

Listing 3.2.7 (*Python asymmetric relation (De Morgan)*)

```
not any((b,a) in R
        for a,b in R)
```

Definition 3.2.8 (*antisymmetric*)

A relation R on A is called *antisymmetric* if

$$((a,b) \in R \wedge (b,a) \in R) \implies a = b\,.$$

An example of an antisymmetric relation is $\leq$ on $\mathbb{N}$. If $a \leq b$ and $b \leq a$, then we know that $a = b$.

Set equivalence is often defined as $A = B$ provided that $A \subseteq B$ and $B \subseteq A$. Given that definition we see that $\subseteq$ is an antisymmetric relation.

Writing the definition of antisymmetric as a universal quantifier is slightly more difficult because of the conjunction ($\wedge$) on the left hand side of the implies ($\implies$). There are several equivalent expressions which can be used.

The first, $\forall a \in A \forall b \in A (a,b) \in R \wedge (b,a) \in R \implies a = b$, can be converted to Python two different ways, both of which are inefficient.

Python does not a logical implication operator `=>`, but if it did, we would be able to write the following.

Listing 3.2.9 (*Broken Python antisymmetric relation*)

```
all(((a,b) in R and (b,a) in R) => (a == b)  # does not work
    for a in A
    for b in A)
```

Lacking the `=>` operator, we can recognize that $X \implies Y$ is logically equivalent to $\neg X \vee Y$, and use `not X or Y` rather than `X => Y`.

Listing 3.2.10 (*Python antisymmetric relation*)

```
  all(not ((a,b) in R and (b,a) in R) or (a == b)
      for a in A
      for b in A)
```

This could perhaps be made more readable with a helper function, `implies`, and less error prone.

Listing 3.2.11 (*Python antisymmetric relation*)

```
  def implies(x,y):
    return (not x) or y

  all(implies((a,b) in R and (b,a) in R,
              (a == b))
      for a in A
      for b in A)
```

The helper `implies(x,y)` translates the logical connective $x \implies y$ — which Python has no built-in operator for — into the equivalent `(not x) or y`. This same helper is reused in `is_transitive` below.

However, it is more idiomatic to employ the `if (a,b) in R` syntax directly into the `all(...)` expression as:

Listing 3.2.12 (*Python antisymmetric relation*)

```
  all(a == b
      for a in A
      for b in A
      if (a,b) in R
      if (b,a) in R)
```

A disadvantage of the code shown above is that it loops a over A and concentrically loops b or A. Thus, if A contains n elements, the loop

has n^2 iterations. Since we know we really need only check values of (a, b) for which $(a, b) \in R$, we can instead loop over R directly.

Listing 3.2.13 (*Python antisymmetric relation*)

```
all(a == b
    for a,b in R
    if (b,a) in R)
```

Every asymmetric relation is also trivially antisymmetric. Because in the case of an asymmetric relation, (a, b) and (b, a) are never simultaneously in the relation. Therefore

$$(a, b) \in R \land (b, a) \in R \implies a = b\,.$$

Definition 3.2.14 (*transitive*)

A relation R on A is called *transitive* if

$$(a, b) \in R \land (b, c) \in R \implies (a, c) \in R\,.$$

The equivalence relation on $\mathbb{N}$ is an example of a transitive relation. If $a = b$ and $b = c$, then we can conclude that $a = c$. Another example is the *divides* relation on $\mathbb{N}$. If b has a as a factor we say $a \mid b$. For example $2 \mid 6$ or $13 \mid 51$. This is a transitive relation, because if $b = m_1 a$ and $c = m_2 b$ then $c = (m_1 m_2)a$ so if $a \mid b$ and $b \mid c$ then $a \mid c$.

The $\neq$ relation on $\mathbb{N}$ is not transitive. If we know $a \neq b$ and $b \neq c$ we cannot conclude that $a \neq c$.

In terms of universal quantifiers the relation is transitive if

$$\forall a, b, c \in A, ((a, b) \in R \land (b, c) \in R \implies (a, c) \in R)\,.$$

In Python this is explicitly:

Listing 3.2.15 (*Python transitive relation*)

```
all(implies(((a,b) in R and (b,c) in R,
            (a,c) in R)
    for a in A
    for b in A
    for c in A)
```

However, since this expression will loop n^3 times (for $|A| = n$), it is better to express the `all(...)` loop smarter.

Listing 3.2.16 (*Python transitive relation*)

```
all((a, c) in R
    for a, b in R
    for c in A
    if (b,c) in R)
```

Definition 3.2.17 (*equivalence relation*)

A relation is called an *equivalence relation* if it is reflexive, symmetric, and transitive.

The equal relation $=$ on $\mathbb{N}$ is an equivalence relation, but there are many other equivalence relations. For example congruence between polygons and similarity between triangles.

An equivalence relation we see often is congruence modulo $n \in \mathbb{N} \setminus \{0, 1\}$. We say $a = b \mod n$ if $(a - b) \mid n$. For example, $13 = 7 \mod 2$ because $13 - 7 = 6$ and $2 \mid 6$ (2 divides 6).

Definition 3.2.18 (*partial order*)

A relation R is called a *partial order* if R is reflexive, antisymmetric, and transitive.

The subset relation $\subseteq$ is an example of a partial order. Given a set of subsets of a given set, we can show that $\subseteq$ is reflexive, antisymmetric, and transitive.

Intuitively, we can deconstruct a set of subsets into setsof chains of subsets: $A_1 \subseteq A_2 \subseteq \cdots A_n$, even though such a chain may not encompass the entire set of subsets. However, this intuition breaks down when we consider subsets of uncountable sets. Consider the set of open intervals of real numbers, $\mathbb{R}$. Whenever we have $A_1 \subset A_2$ we can always find another A_3 such that $A_1 \subseteq A_3 \subseteq A_2$.

Definition 3.2.19 (*strict partial order*)

A relation R is called a *strict partial order* if R is irreflexive, asymmetric, and transitive.

Clearly every strict partial order is also a partial order because every

asymmetric relation is also antisymmetric. But a not ever partial order is a strict partial order.

An example of a strict partial order is the strict subset relation $\subsetneq$. $A \subsetneq B$ is like $A \subseteq B$ except that by definition $A \neq B$.

Definition 3.2.20 (*(strongly) connected relation*)

A relation R on A is called *connected* if $a, b \in A$ with $a \neq b$, then $(a, b) \in R$ or $(b, a) \in R$. Furthermore, R is called *strongly connected* if $a, b \in A$, then $(a, b) \in R$ or $(b, a) \in R$.

The $<$ relation on $\mathbb{N}$ is a connected relation, because given any distinct $a, b \in \mathbb{N}$, either $a < b$ or $b < a$. Intuitively, the all $\mathbb{N}$ can be ordered into a single connected (infinite) sequence subject to the $<$ so that at any point, all the numbers on the left are *less than* all the numbers on the right.

Likewise, $\leq$ on $\mathbb{N}$ is a strongly connected relation.

The connected relation can be expressed in Python as

Listing 3.2.21 (*Python strongly connected relation*)

```
all((a,b) in R or (b,a) in R
    for a in A
    for b in A
    if a != b)
```

Likewise, the connected relation can be expressed as

Listing 3.2.22 (*Python connected relation*)

```
all((a,b) in R or (b,a) in R
    for a in A
    for b in A)
```

Definition 3.2.23 (*well-founded order relation*)

A relation R on A is called a *well-founded order* relation if the following holds: if $S \subseteq A$ then either $S = \emptyset$ or $\exists m \in S$ (called a *minimal element*) such that $\forall x \in S \setminus \{m\}$ we have $(x, m) \notin R$.

To implement this relation in Python using universal and existential quantifiers we need a way to iterate across all subsets of a given set.

There are many ways to do this, but one is to use the `powerset` function from the `more_itertools` Python package. `powerset(A)` returns an iterator over all subsets of `A` — for example, `powerset([1,2])` yields `[]`, `[1]`, `[2]`, and `[1,2]`.

Listing 3.2.24 (*Python well-founded order relation*)

```
from more_itertools import powerset

all(not S or any(all((x,m) not in R
                     for x in S
                     if x != m)
                 for m in S)
    for S in powerset(A))
```

This Python code corresponds to the logical expression:

$$\forall S \subseteq A \; ((S = \emptyset) \vee (\exists m \in S \; \forall x \in S \setminus \{m\}, (x, m) \notin R))$$

Warning, it is usually a bad idea to iterate over the power set, because the size of the power set of A is 2^n when the size of A is n. This iteration of an *exponentially* large set can be exceedingly slow.

Something to note about the careful formation in Definition 3.2.23 is that it does not claim an element exists which is *less than* every other element, but rather that an element m exists for which nothing is less than m. The student should reflect on this until convinced that those are two different claims.

A minimal element need not be unique. Consider the set

$$A = \{\{1\}, \{2\}, \{1, 2\}\},$$

and the relation $\subseteq$, we see that both $\{1\}$ and $\{2\}$ are both minimal elements, because there is no other subset of $\{1\}$ or $\{2\}$ in A. However, $\{1\}$ is not a subset of every other set in A; in particular $\{1\} \not\subset \{2\}$.

3.3 Functions as Relation

Definition 3.3.1 (*function relation*)

A relation R from A to B is called a *function* if $\forall a \in A \; \exists! \; b \in B$ for which $(a, b) \in R$.

Another way of stating Definition 3.3.1 is that the following hold

1. $a \in A \implies \exists b \in B$ such that $(a, b) \in R$, and
2. $(a_1, b_1) \in R \wedge (a_2, b_2) \in R \implies (a_1 = a_2 \implies b_1 = b_2)$

The following Python expression implements the first condition:

Listing 3.3.2 (*Python first condition of function relation*)

```
all(any((a,b) in R
        for b in B)
    for a in A)
```

The second expression can be implemented as in Listing 3.3.3 which takes advantage of the fact that

$$(x \in R \implies p(x)) \iff (\forall x \in R, p(x)) \tag{3.3}$$

Equation 3.3 allows us to rewrite condition 2 (above) as

$$\forall (a_1, b_1) \in R \; \forall (a_2, b_2) \in R, a_1 = a_2 \implies b_1 = b_2,$$

which can easily be translated to Pthon thanks to the `implies` function given in Listing 3.2.11

Listing 3.3.3 (*Python second condition of function relation*)

```
all(implies(a1 == a2, b1 == b2)
    for a1, b1 in R
    for a2, b2 in R)
```

We can implement $\exists!$ in Python as the following function which accepts an iterable, `g`, of values to be interprested as Booleans, and determines whether exactly one of the values is `true`. Keep in mind that

in Python there are many `false` values. `False` is perhaps the most common such value; however, `[]` (empty list), {} (empty set), 0, and many other values are considered `false`.

Listing 3.3.4 (*Python* `uniquely` *function*)

```
def uniquely(g):
  i = 0
  for e in g:
    if e:
      i = i + 1
      if i == 2:
        return False
  return i == 1
```

To better understand the function `uniquely` let's look at some examples.

Listing 3.3.5 (*Python* `uniquely` *function*)

```
uniquely([])
=> False

uniquely([False, True, False, False])
=> True

uniquely([False, True, False, True])
=> False

uniquely([False, False, False, False])
=> False
```

`uniquely` completes the quantifier trio from Chapter 2: just as `all` implements ∀ and `any` implements ∃ (Section 2.3.1), `uniquely` implements the *unique-existence* quantifier ∃!. Whenever you see ∃! in a mathematical definition, reach for `uniquely`.

Just as $\forall x \in S, P(x)$ maps to `all(P(x) for x in S)` and $\exists x \in S, P(x)$ maps to `any(P(x) for x in S)`, the *unique existence* quantifier maps to:

$$\exists!\ x \in S, P(x) \quad \longleftrightarrow \quad \texttt{uniquely(P(x) for x in S)}$$

Using `uniquely` we can implement the function relation as the fol-

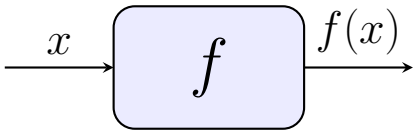
lowing function. For each $a \in A$ it tests whether there is exactly one $b \in B$ for which (a, b) is in the relation R.

Listing 3.3.6 (*Python function relation*)

```
all(uniquely((a,b) in R
                for b in B)
     for a in A)
```

`is_function_relation` composes two quantifier patterns: an outer `all` (Loop-for-Detection, $\forall a \in A$) wrapping an inner `uniquely` ($\exists! b \in B$). Nesting quantifier implementations mirrors nesting quantifiers in mathematics: $\forall a \in A,\ \exists! b \in B,\ (a, b) \in R$.

3.4 Functions as Generators



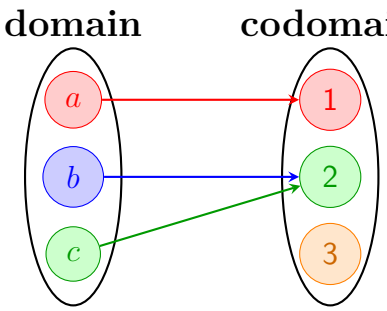
What is a function? A function such as

$$f : X \to Y$$

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.

3.5 Functional Notation

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:

$$f(x) = 3x + 1 \tag{3.4}$$

Let's start with a very simple function (3.4). 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 (3.5)

$$f : \mathbb{N} \to \mathbb{N} \text{ by } f(x) = 3x + 1 \tag{3.5}$$

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 (3.6) shows an example of such a function.

$$f : \mathbb{R}^2 \to \mathbb{R} \text{ by } f(x, y) = 3x - 2y + 1 \tag{3.6}$$

We may also specify functions using cases, like in Equation (3.7)

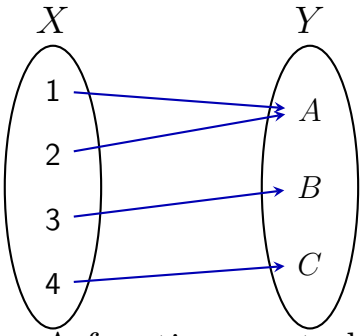
$$|x| = \begin{cases} x & ; \text{if } x > 0 \\ 0 & ; \text{if } x = 0 \\ -x & ; \text{if } x < 0 \end{cases} \tag{3.7}$$

3.6 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.

3.7 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*.

Listing 3.7.1

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

3.8 Functions as Mappings

3.8.1 Injections

A function is injective if no two values from the domain are mapped to the same value in the range. These functions are also called 1:1 or *one-to-one*.

Definition 3.8.1 (*injective*)

A function $f : A \to B$ is called *injective* if

$$f(a) = f(b) \implies a = b\,,$$

or by the contrapositive

$$a \neq b \implies f(a) \neq f(b)\,.$$

Listing 3.8.2 (*Python* `is_injection` *function*)

```
def is_function(f, xs, ys):
  return all(???
             for x in xs)

def is_injection(f, xs, ys):
  return is_function(f, xs, ys) and \
         all(???
             for x1 in xs
             for x2 in xs
             if ???)
```

3.8.2 Surjections

A function is *surjective* if every element in the range is the image of some element in the domain. These functions are also sometimes called *onto* and we say f maps A *onto* B.

Definition 3.8.3 (*surjective*)

A function $f : A \to B$ is called *surjective* if

$$\forall y \in B \; \exists x \in A, \; f(x) = y \, .$$

Listing 3.8.4 (*Python* `is_surjection` *function*)

```
def is_surjection(f, xs, ys):
  return is_function(f, xs, ys) and \
         all(any(???
                 for x in xs)
             for y in ys)
```

3.8.3 Bijections

If a function is both injective and surjective, then it is called *bijective*. Such functions are also invertible.

Definition 3.8.5 (*bijective*)

A function f is called *bijective* if it is injective and surjective.

3.9 Programming Challenges

Complete the exercises in the file `algebra/relation.py` and test with the tests in the file `tests/relation_test.py`.

Complete the exercises in the file `algebra/function.py` and test with the tests in the file `tests/function_test.py`.

Chapter 4

Recursion

Chapter Objectives

- State the four components of the Recursion Recipe: termination case, reduction, recursive call, and bookkeeping.
- Apply the Recursion Recipe to implement and trace recursive functions including `factorial`, `gcd` (Euclid's algorithm), `slow_power`, and `sum_list`.
- Implement `prime_factors` to return the sorted list of prime factors of an integer.
- Apply the Legendre formula to compute the exact power of a prime p dividing $n!$ without computing $n!$ itself.
- Identify tail-recursive functions (where the recursive call's result is returned directly, with no bookkeeping).
- Implement the `recursion` programming assignment.

4.1 Specifying Functions by Recurrence

Functions can be defined in terms of themselves. We might use an intuitive definition such as the definition of factorial in Equation (4.1).

We can define the factorial as

$$n! = 1 \times 2 \times \ldots \times n \tag{4.1}$$

or we can define it as

$$n! = \begin{cases} 1 & ;\text{if } n = 0 \\ n \times (n-1)! & ;\text{if } n > 0 \end{cases} \tag{4.2}$$

Notice that $n!$ is defined in terms of itself.
Similarly, we can define exponentiation.

$$x^n = \underbrace{x \times x \times \ldots \times x}_{n \text{ times}} . \tag{4.3}$$

We can define the same function more explicitly by using recursion as in Equation (4.4).

$$x^n = \begin{cases} 1 & ;\text{if } n = 0 \\ x \times x^{n-1} & ;\text{if } n > 0 \\ \frac{1}{x^{-n}} & ;\text{if } n < 0 \text{ and } x \neq 0 \end{cases} \tag{4.4}$$

4.2 Principles of a Recursive Function

A recursive function is a function which calls itself, or a sequence of functions, f_1 calls f_2, f_2 calls f_3, ..., f_n calls f_1. Many mathematical relationships can be elegantly represented by recursive functions.

In Python there is no keyword to mark a function as recursive. As a professional habit, annotate recursive functions with a comment such as `# RECURSIVE` so that readers immediately know to look for the four-element structure described in Pattern 4.2.1.

Some programming languages require an explicit *declaration* for recursive functions. This matters most for *tail-recursive* functions: a recursive call is in *tail position* when it is the very last operation performed before returning. Compilers for languages such as Scheme, Scala, and OCaml can transform tail calls into simple loops, eliminating call-stack

growth — this is called *tail call optimization* (TCO). Annotating a function `@tailrec` (Scala) or `[@@tail_mod_cons]` (OCaml) lets the compiler verify the property and raise an error if the code fails to satisfy it. Python intentionally does not perform TCO; see Section 4.2.3 for details.

Pattern 4.2.1 (*The Recursion Recipe*)

A *simple recursive function* calls itself by name. It has four structural elements:

1. **Termination case** — a condition under which the function returns immediately without a recursive call.
2. **Reduction** — a transformation of the input that makes the problem strictly smaller, guaranteeing progress toward the termination case.
3. **Recursive call** — a call to the same function with the reduced input.
4. **Bookkeeping** — combining or transforming the result of the recursive call before returning.

```
# RECURSIVE
def f(n, ...):
  if some_condition(n):            # termination case
    return some_value
  else:
    r = some_reduction(n)          # reduction
    partial = f(r, ...)            # recursive call
    return some_combination(partial)  # bookkeeping
```

4.2.1 Termination Case

To avoid looping forever, a recursive function generally needs a termination case.

In the example above, `return some_value` terminates the recursion. The condition of termination, indicated by `if some_condition(n)`, is different for different applications, but is normally a decision based on one of the input parameters to the function. Typical termination conditions

are `if n==0:` , `if n < 0:` , or `if n < epsilon:` .

It is common in terms of code layout, that the termination conditions comes first, and the inductive cases follow later. However, when writing the code, we may write the termination condition last. It might not always be clear at first what the correct condition of termination is. So your might first write something like the following, and come back later to replace the placeholders `...` and `pass` with the correct code.

Listing 4.2.2

```
def f(n, ...):
  if ...
    pass
  else:
    return g( f(n-1))
```

There are exotic cases, of recursive functions which do not terminate. These functions either implement intentionally infinite loops such as event handlers, or they implement lazy infinite sequences. A lazy sequence is a sequence which *generates* the successive element only when needed. The Python language avoids the use of recursion to implement such function by providing the `yield` keyword, as shown in the following example.

Listing 4.2.3

```
# define a generator which produces a lazy infinite
# sequence of integers
def positive_integers(n, ...):
  while True:
    yield n
    n = n+1
```

4.2.2 Inductive Case

The Inductive case deals with three things: reduction, the recursive call, and usually some bookkeeping. Usually one or more recursive calls are made, each with some *reduced* input. Each recursive call returns a value, and the function often needs to do something with that return value such as add it to an accumulator, or conjoin it to a list, add it to a set, etc.

For a function to be *simply* recursive it must contain a call to itself. By contrast, see Section 4.2.4, for a discussion of *mutually* recursive functions.

The simply recursive function may call itself multiple times in a single expression, such as `f(left) + f(right)`. Or the function may make a different call in branches of a conditional such as

Listing 4.2.4

```
def f(n, ...):
  if finished(n):
    return something
  elif x % 2 == 0:
    return f(n-1, x+1)
  else:
    return f(n-1, x+2)
```

In order for a recursive function to terminate, each recursive call needs to *somehow* reduce the input so that the input *tends toward* the termination condition. Of course this *reduction* may look very different for different applications. If the termination condition is `n == 0`, then the reduction might look like `recursive_call(n - 1)`. If the termination conditions is `abs(x) < epsilon`, then the reduction might look like `recursive_call(x/2.0)`. If the recursive call is always made with the same arguments that it was given, then termination will never occur.

4.2.3 Tail Call Optimization

A recursive call is in *tail position* when it is the last thing a function does before returning — no further computation is applied to its result. For example:

Listing 4.2.5

```
# tail-recursive: the recursive call IS the return value
def count_down(n):
  if n == 0:
    return 0
  else:
    return count_down(n - 1)   # tail position

# NOT tail-recursive: multiplication happens AFTER the call
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)   # n * (...) is applied
   after
```

In a tail-recursive function, the current stack frame is no longer needed once the recursive call is made. A compiler that performs *tail call optimization* (TCO) reuses the same frame rather than pushing a new one, turning the recursion into a loop with $O(1)$ stack space. Languages that guarantee TCO include Scheme, Haskell, and Scala (`@tailrec`). In these languages, TCO is essential — recursive loops over large data would otherwise overflow the stack.

Python's designer, Guido van Rossum, deliberately chose *not* to implement TCO. His reasoning: stack traces are important for debugging, and TCO would silently remove frames from them. As a consequence, Python recursive functions are limited by the call stack (default depth ≈ 1000). For large inputs, Python programmers convert tail-recursive functions to explicit loops or use `sys.setrecursionlimit`. See Section 5.1 for a concrete example of Python hitting this limit, and a divide-and-conquer strategy that sidesteps it.

4.2.4 Mutual Recursion

It is also possible that two or more different functions contain mutual calls; *e.g.*, if function `f` calls function `g`, and function `g` calls function `f`. These types of recursions are more rare, and we will not consider them in detail in this chapter.

Listing 4.2.6

```
# f and g are MUTUALLY recursive
def f(n):
  ...
  g(n)
  ...

def g(n):
  ...
  f(n)
  ...
```

4.3 Simple Recursive Functions

4.3.1 Factorial

First let's look at some simple examples of recursive functions.

$$n! = \begin{cases} 1 & ;\text{if } n = 0 \\ n \times (n-1)! & ;\text{if } n > 0 \end{cases} \tag{4.5}$$

We can implement this in Python.

Apply the Recursion Recipe (Pattern 4.2.1): the termination case is $n = 0$; the reduction is $n \to n-1$; the recursive call is `factorial(n-1)`; the bookkeeping multiplies the result by n.

Listing 4.3.1 (*Factorial Function*)

```
def factorial(n):
  ...
  return ???
```

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.

4.3.2 Greatest Common Divisor (GCD)

We can use Euclid's algorithm to recursively compute the GCD of a given a and b.

$$\gcd(a,b) = \begin{cases} a & ;\text{if } b = 0 \\ \gcd(b, a \mod b) & ;\text{otherwise} \end{cases}$$

The recursive function, `gcd`, computes the greatest common divisor using Euclid's algorithm.

Apply the Recursion Recipe: the termination case is $b = 0$; the reduction transforms (a, b) into $(b, a \bmod b)$; the recursive call is `gcd(b, a % b)`; there is no bookkeeping — the result of the recursive call is returned directly, making `gcd` *tail-recursive* (Section 4.2.3).

Listing 4.3.2 (*Greatest Common Divisor*)

```
def gcd(a, b):
  ...
  return ???
```

4.3.3 Continued Fraction

We would like to compute the value of the following infinite continued fraction.

$$\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cdots}}}}$$

We will do this by developing a function, `continued_fraction`, which computes the nth term of the following sequence.

$$
\begin{aligned}
a_0 &= x \\
a_1 &= \frac{1}{1+x} &&= \frac{1}{1+a_0} \\
a_2 &= \frac{1}{1+\frac{1}{1+x}} &&= \frac{1}{1+a_1} \\
a_3 &= \frac{1}{1+\frac{1}{1+\frac{1}{1+x}}} &&= \frac{1}{1+a_2} \\
&\vdots && \vdots \\
a_n &= \cdots &&= \frac{1}{1+a_{n-1}}
\end{aligned}
$$

This sequence of equalities can be combined into a single equation.

$$
a_n = \begin{cases} x & ;\text{if } n = 0 \\ \frac{1}{1+a_{n-1}} & ;\text{otherwise} \end{cases} \tag{4.6}
$$

Listing 4.3.3

```
def continued_fraction(x, n):
  ...
  return ???
```

Before implementing, identify the Recursion Recipe elements from Equation (4.6): what is the termination case? What input is passed to the recursive call? What bookkeeping transforms the recursive result into a_n?

Implement this sequence as a function of x and n.

For which values of x does the sequence converge? For which values does it not converge? What is the value the sequence converges to?

4.3.4 Slow Exponentiation

We will look at three different functions for computing a^n. The first is called *slow exponentiation* and in Sections 5.2 and 5.3 we will look at two faster algorithms.

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 (4.4) on page 88.

Listing 4.3.4

```
1  def slow_power(a, n):
2    if n == 0:
3      return 1
4    elif n == 1:
5      return a
6    else:
7      return a * slow_power(a, n-1)
```

Trace the Recursion Recipe here: termination at $n = 0$ (and $n = 1$ as an optimization); reduction $n \to n - 1$; recursive call `slow_power(a, n-1)`; bookkeeping multiplies the result by a. The $n = 1$ case avoids one unnecessary recursive call — small, but a good habit.

4.3.5 Summing a List

Suppose you with to add up a given list of numbers. Mathematically, the sum of a list is 0 if the list is empty; otherwise the sum is the first element plus the sum of the remaining elements.

Before looking at the code, note the Python *list slicing* syntax used below. `numbers[i:j]` returns a new list containing the elements of `numbers` from index `i` up to (but not including) index `j`. Either bound may be omitted: `numbers[1:]` means "everything from index 1 to the end", *i.e.*, all elements except the first. Similarly, `numbers[:3]` means the first three elements.

Listing 4.3.5

```
1 def sum_list(numbers):
2   if not numbers:
3     return 0
4   else:
5     return numbers[0] + sum_list(numbers[1:])
```

Trace the Recursion Recipe: termination when the list is empty; reduction passes `numbers[1:]` (every element except the first); the recursive call is `sum_list(numbers[1:])`; the bookkeeping adds `numbers[0]` to the result.

4.4 Prime Factors

By *factor* of $n \in \mathbb{N}$, we mean an integer $p > 1$ such that $pk = n$ for some $k \in \mathbb{N}$. One might equivalently consider that 1 is a factor, but if so all the statements *let p be a factor* in the following section would have to annoyingly be replaced with *let p be a factor different than 1*.

Given a number n, how can we compute its prime factors? We would like to write the Python function, `prime_factors` (Listing 4.4.9) to compute the list of prime factors. To make this function correct and efficient we will make use of several theorems. These theorems will allow us to write `prime_factors` using another function called `smallest_factor`.

4.4.1 The Smallest Prime Factor

We would like to implement the function in Listing 4.4.1.

Listing 4.4.1

```
def smallest_factor(n, start, stop):
    ...
    return ???
```

The first thing we notice is that if we find the smallest factor, it is necessarily prime. Theorem 4.4.2 formalizes this claim. For example the smallest factor of 100 is 2, and 2 is prime. The smallest factor of 51 is 3, and 3 is prime. If we claim the smallest factor if 16 is 4, then this is clearly false because 4 is not prime—4 has a factor, namely 2, and 2 is also a factor of 16; *i.e.*, 4 is not the smallest factor of 16.

Theorem 4.4.2 (*Smallest Factor is Prime*)

If n is a composite number, then its smallest factor is prime.

Proof by contradiction. Suppose p is the smallest factor of n with $n = pk$ and that p is composite with $p = ab$. Then $a < p$. But $n = pk = abk$ so a is also a factor of n less than p, which contradicts our assumption that p is the smallest factor of n.

Why is Theorem 4.4.3 useful when writing the Python function `prime_factors`? Because, it implies that to find the smallest prime factor, we simply need to find the smallest factor; we don't need to test that it is prime.

The next Theorem (4.4.3) formalizes the observation that if n has a factor, then it has factor between 1 and $\sqrt{n}$. For example, 100 is composite, thus it has at least one factor less than $\sqrt{100} = 10$, namely 2. On the contrary, 1009 is prime because it has no factor less than $\sqrt{1009} <= 31.7 < 32$. To determine that 1009 is prime, we need only check potential factors, 2, 3, ... 32.

Theorem 4.4.3 (*Smallest Factor is Bounded*)

If n is composite, then the smallest factor n is not larger than $\sqrt{n}$.

Proof by contradiction. Suppose the contrary, that n is composite, and the smallest factor of n is p and that $p > \sqrt{n}$. Since n is composite, then there is a $q \in \mathbb{N}$ with $n = pq$. Since p is the smallest factor, then $p \leq q$. *I.e.*, $\sqrt{n} < p \leq q$. Let $\alpha, \beta \in \mathbb{R}$ with $\alpha, \beta > 0$ such that $p = \sqrt{n} + \alpha$ and $q = \sqrt{n} + \beta$. so

$$\begin{aligned} n &= pq \\ &= \big(\sqrt{n} + \alpha\big)\big(\sqrt{n} + \beta\big) \\ &= n + \sqrt{n}(\alpha + \beta) + \alpha\beta \\ &> n \end{aligned}$$

But $n > n$ is a contradiction. So no such p exists. And thus if p is the smallest factor of n, then $p \leq \sqrt{n}$.

According to Theorem 4.4.3 we know that unless n is prime, then it has a prime factor p such that $2 \leq p \leq \sqrt{n} \leq \lceil\sqrt{n}\rceil$, where $\lceil\sqrt{n}\rceil$

represents the smallest integer p' such that $\sqrt{n} \leq p'$. If p is a perfect square then $\sqrt{n} = \lceil \sqrt{n} \rceil$; otherwise $\sqrt{n} < \lceil \sqrt{n} \rceil$.

Since $\sqrt{n} \leq \lceil \sqrt{n} \rceil$, Theorem 4.4.3 also claims the smallest factor is not larger than $\lceil \sqrt{n} \rceil$.

Theorem 4.4.4 (*Prime factors of a perfect square*)

If n is a perfect square with $n = q^2$ then the factors of n are simply the factors of m but repeated.

Proof. We will show that if p is prime and $p^m | q$ then $p^{2m} | q^2$.

Suppose that q has been factored into ℓ-many distinct primes, $p_1 \ldots \_\ell$ with ℓ-many (not necessarily distinct) multiplicites, $m_1 \ldots m_\ell$ as follows

$$q = p_1^{m_1} \cdot p_2^{m_2} \cdots p_\ell^{m_\ell}.$$

Then

$$\begin{aligned} q^2 &= (p_1^{m_1} \cdot p_2^{m_2} \cdots p_\ell^{m_\ell})^2 \\ &= p_1^{2m_1} \cdot p_2^{2m_2} \cdots p_\ell^{2m_\ell}. \end{aligned}$$

Thus if $p_i \mid n$, then $p_i^2 \mid n^2$.

Theorem 4.4.4 tells us that if we have computed a list of the prime factors of n, in Python, then to compute the prime factors of n^2, we simply duplicate the list. Likewise, if computing the prime factors of n, if we discover that n is a perfect square, then we need only compute the prime factors of $\sqrt{n}$, which presumably is faster, and then duplicate the list. In Python, we can know whether `n` is a perfect square, by testing `round(sqrt(n))**2 == n`. If n is a perfect square, and if the variable `fs` is a list of prime factors of $\sqrt{n}$, then `fs + fs` (Python list concatenation) is the list of prime factors of n.

WARNING, in the function `prime_factors` you are asked to compute the prime factors, explicitly in increasing order. If `fs` is a list in increasing order, then in general `fs + fs` is not. Thus `fs + fs` will need to be either sorted, or `fs` must be duplicated element by element

instead.

Theorem 4.4.5

If s is the smallest factor of n, then $\frac{n}{s}$ has no factor less than s.

Proof by contradiction. Suppose n is composite and that s is its smallest factor. Choose t such that $st = n$. Suppose that $\frac{n}{s}$ is composite and that q is its smallest factor. In order to show a contradiction, suppose $q < s$. By Theorem 4.4.2 we know s and q are both prime, thus q does not divide s. Since $q|\frac{n}{s}$, choose r such that $qr = \frac{n}{s}$, so $qrs = n$. This means that q is a factor of n with $q < s$ which contradicts the claim that s is the smallest factor of n.

How does Theorem 4.4.5 help in writing the Python function, `prime_factors`? Once we've determined the smallest factor of 1001 is 7, then we know that $\frac{1001}{7} = 143$ has no factor less than 7. Furthermore, since $\lceil\sqrt{143}\rceil = 12$, we only need to search $7, 8, \ldots, 12$.

Theorem 4.4.6 (*Odd Factors*)

If n is odd, then all its factors are odd.

Proof by contrapositive. We will prove, that if n has an even factor, then n is even.

Suppose n has an even factor $2p$, then there is a q such that $n = (2p)(q) = 2(pq)$. Thus n is even.

Using Theorem 4.4.6 we can optimize the example above of 1009. We already said to determine that 1009 is prime we only need to search $2, 3, \ldots, 32$. However, by Theorem 4.4.6, we can skip even numbers, and check only 3, 5, 7, ..., 31; thus cutting search time in half.

Furthermore, we argued above that to find a factor of 143, is suffices to check $7, 8, 9, 10, 11, 12$. But by Theorem 4.4.6, we now know it suffices to check $7, 9, 11$, skipping the even candidates.

4.4.2 Is a Given Integer Prime?

We can easily use the function 4.4.1 to determine whether a given integer is prime by testing whether it is equal to its smallest factor.

Listing 4.4.7

```
def is_prime(n):
  assert n > 1
  assert isinstance(n, int)
  return n == smallest_factor(n, 2, n+1)
```

4.4.3 Finding All Prime Factors

According to Theorem 4.4.2, if we want to find a prime factor of n, it suffices to find the smallest factor. Theorem 4.4.3 assures us that we need only search the integers in the range $2 \leq p \leq \lceil \sqrt{n} \rceil$.

To find the prime factors of a given n, the following algorithm suffices.

1. Initialize $n_0 = 2$.
2. If n is a perfect square, then, by Theorem 4.4.4, recursively compute the prime factors of $\sqrt{n}$, duplicate them, and quit.
3. Try, in turn, the values $n_0 \leq p \leq \lceil \sqrt{n} \rceil$, by Theorem 4.4.3. If n_0 is odd, then we need not search odd candidates for p, by Theorem 4.4.6.
4. If we discover p such that `n % p == 0`, then p is a factor.
   - The first such p discovered is the smallest factor, and is therefore prime.
   - Recursively compute the prime factors of $\frac{n}{p}$, and prepend p to the result.
   - To find prime factors of $\frac{n}{p}$, start at $n_0 = p$, by Theorem 4.4.5.
   - Any such prime factor will be less than $\left\lceil \sqrt{\frac{n}{p}} \right\rceil$.
5. If no p in the range $n_0 \leq p \leq \lceil \sqrt{n} \rceil$ is a factor of n,

- Then n is prime and its only factor is n itself.

The algorithm explained above makes one recursive call per prime factor. In each recursive call we search a range with is smaller than the previous time. The upper bound decreases from $\lceil\sqrt{n}\rceil$ to $\left\lceil\sqrt{\frac{n}{p}}\right\rceil$, and the lower bound either remains the same (to accommodate repeated factors) or increases to the next prime factor.

Example 4.4.8 (*Computing Prime Factors*)

Let's compute the prime factors of

$$29,215,368 = 2 \times 2 \times 2 \times 3 \times 3 \times 7 \times 7 \times 7 \times 7 \times 13 \times 13\,.$$

1. Factors of $29,215,368$:
   According to Theorem 4.4.3, if $29,215,368$ is composite, then its smallest prime factor, p is in the range, $2 \leq p \leq \lceil\sqrt{29,215,368}\rceil = 5406$; *i.e.* `range(2, ceil(sqrt(29215368))+1)`. We find that $\frac{29,215,368}{2} = 14,607,684$. So we compute the prime factors of $14,607,684$, and then prepend 2.

2. Factors of $14,607,684$:
   By Theorem 4.4.4, since $14,607,684$ is a perfect square $14,607,684 = 3822^2$, then we compute the prime factors of 3822, and then append that list to itself.

3. Factors of 3822:
   We search for a factor, p, in the range $2 \leq p \leq \lceil\sqrt{3822}\rceil = 62$. We find that $\frac{3822}{2} = 1911$. So we prepend 2 to the prime factors of 1911.

4. Factors of 1911:
   We now search for a factor, p, of 1911 in the range $2 \leq p \leq \lceil\sqrt{1911}\rceil = 44$. We find that $\frac{1911}{3} = 637$. So we prepend 3 to the prime factors of 637.

5. Factors of 637:
   Since we know there are no remaining factors less than

3, we now search for a factor, p, of 637 in the range $3 \leq p \leq \lceil\sqrt{637}\rceil = 26$. We find that $\frac{637}{7} = 91$. So we prepend 7 to the prime factors of 91.

Note that when searching the range $3 \leq p \leq \lceil\sqrt{637}\rceil$ we need not search any even values of p. Why? Because of Theorem 4.4.6, we know that since 637 is odd, then all its factors are odd. This means in Python we may use a loop which starts at an odd number and steps by 2 , (thus halving the each time) such as:

```
next(p for p in range(3,27,2) if 367 % p == 0)
```

6. Factors of 91:
   Since we know there are no remaining factors less than 7, we now search for a factor, p, of 91 in the range $7 \leq p \leq \lceil\sqrt{91}\rceil = 10$. We find that $\frac{91}{7} = 13$. So we prepend 7 to the prime factors of 13.

7. Factors of 13:
   Finally, we search in the range of $7 \leq p \leq \lceil\sqrt{13}\rceil = 4$ for a p which divides 13, and we fail to find one. By Theorem 4.4.3, this means 13 is prime; so its list of prime factors is `[13]` .

Notice in Example 4.4.8 that when search in range $[r_{lower}, r_{upper}]$, for a factor of n, if we find a prime factor, p, then we continue the search (recursively) on a smaller range $\left[p, \left\lceil\sqrt{\frac{n}{p}}\right\rceil\right]$

The function `prime_factors` computes a sorted list of prime factors of the given n.

The algorithm above follows the Recursion Recipe: termination occurs when n is prime (no factor found in the search range); the reduction divides n by its smallest prime factor p; the recursive call finds the prime factors of $\frac{n}{p}$; the bookkeeping prepends p to the result.

Listing 4.4.9

```
def prime_factors(n):
    ...
    return ???
```

4.4.4 Prime Factors of $n!$

The function `factorial_factors` computes the sorted list of prime factors of $n!$, but without computing $n!$. *I.e.*, `factorial_factors` appends the prime factors of n with the prime factors of $n-1$, etc. down to the prime factors of 2. And finally sorts them into one list in order.

Implement a function `factorial_factors` which, given n, uses a list comprehension to compute and return the list of prime factors of n. Use the function, `prime_factors` to get the prime factors of each integer between n and 2.

WARNING: do not compute $n!$ and then find its prime factors, rather append the prime factors of n to the prime factors of $(n-1)!$, recursively.

Hint: The prime factors of $n!$ are the prime factors of n concatenated to the list of prime factors of $(n-1)!$, except for the termination condition where the prime factors of 2! is the list `[2]` and the prime factors of 1! and 0! are the empty list `[]`.

Listing 4.4.10

```
def factorial_factors(n):
  ...
  return sorted(...)
```

4.4.5 Legendre's Formula for Factors of $n!$

Another way to find the prime factorization of $n!$ is to use Legendre's formula. We know that all the primes between 2 and n are all prime factors and the only prime factors of $n!$. Legendre's formula helps us find how many times each occurs in the prime factorization.

Theorem 4.4.11 (*Legendre*)

If p is prime and $2 \leq p \leq n$, then the number of occurrences of p in the prime factorization of $n!$ is given by $v_p(n!)$ in the following expression.

$$v_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor .$$

Finally

$$n! = \prod_{p \in \text{primes}(n)} p^{v_p(n!)}$$

The proof is from Wikipedia `https://en.wikipedia.org/wiki/Legendre%27s_formula`

Proof. Since $n!$ is the product of the integers 1 through n, we obtain at least one factor of p in $n!$ for each multiple of p in in $[1, 2, \ldots, n]$, of which there are $\left\lfloor \frac{n}{p} \right\rfloor$ many. Each multiple of p^2 contributes an additional factor of p, of which there are $\left\lfloor \frac{n}{p^2} \right\rfloor$ many; each multiple of p^3 contributes yet another factor of p; of which there are $\left\lfloor \frac{n}{p^3} \right\rfloor$ many, etc. Adding up the number of these factors gives the infinite sum for $u_p(n!)$.

Even though the sum in Theorem 4.4.11 is stated as an infinite sum, it is really a finite sum, because as soon as $p^i > n$ then $\left\lfloor \frac{n}{p^i} \right\rfloor = 0$ (See Eq (4.8)), so the formula can be written as:

$$v_p(n!) = \sum_{i=1}^{\lfloor \log_p n \rfloor} \left\lfloor \frac{n}{p^i} \right\rfloor \tag{4.7}$$

Furthermore, $\lfloor \log_p n \rfloor$ can be calculated as follows:

$$\lfloor \log_p n \rfloor = \begin{cases} 0 & ; \text{if } n < p \\ 1 & ; \text{if } \sqrt{n} < p \leq n \\ \left\lfloor \frac{\ln n}{\ln p} \right\rfloor & ; \text{otherwise} \end{cases} \tag{4.8}$$

Listing 4.4.12

```
def legendre_exponent(n, p):
  m = int(ceil(log(n) / log(p)))
  return sum(n // p**i
             for i in range(1, m + 1))
```

Now given the implementation of `legendre_exponent` in Listing 4.4.12, we can implement `legendre_expansion`.

Listing 4.4.13

```
def legendre_expansion(n):
  return [ q for p in range(2,n+1)
           # i.e., if p is prime
           if p==smallest_factor(p, 2, p)
           for q in [p] * legendre_exponent(n,p)]
```

Example 4.4.14 is a step-by-step example how to compute the prime factorization of 20! using Legandre's formula, Theorem 4.4.11.

Example 4.4.14 (*Compute prime factorization of* 20!)

- First, we determine the prime factors (ignoring multiplicity) of 20! which are the primes up to an (if 20 were prime) including 20: $2 \leq p \leq 20$; *i.e.* 2, 3, 5, 7, 11, 13, 17, and 19.
- Next, we compute $\lfloor \log_2 20 \rfloor$, $\lfloor \log_3 20 \rfloor$, $\lfloor \log_5 20 \rfloor$, $\lfloor \log_7 20 \rfloor$, $\lfloor \log_{11} 20 \rfloor$, $\lfloor \log_{13} 20 \rfloor$, $\lfloor \log_{17} 20 \rfloor$, and $\lfloor \log_{19} 20 \rfloor$:

$$\begin{aligned} \lfloor \log_2 20 \rfloor &= 4 && ; \text{ because } 2^4 = 16 < 20 < 32 = 2^5 \\ \lfloor \log_3 20 \rfloor &= 2 && ; \text{ because } 3^2 = 9 < 20 < 27 = 3^3 \\ \lfloor \log_5 20 \rfloor &= 1 && ; \text{ because } 5 > \sqrt{20} \\ &\vdots && \vdots \\ \lfloor \log_{19} 20 \rfloor &= 1 && ; \text{ because } 19 > \sqrt{20} \end{aligned}$$

- Next, we compute $v_2(20!)$, $v_3(20!)$, $v_5(20!)$, $v_7(20!)$, $v_{11}(20!)$,

$v_{13}(20!)$, $v_{17}(20!)$, and $v_{19}(20!)$.

$$\begin{aligned}
v_2(20!) &= \sum_{i=1}^{4} \left\lfloor \frac{20}{2^i} \right\rfloor \\
&= \lfloor 20/2 \rfloor + \lfloor 20/4 \rfloor + \lfloor 20/8 \rfloor + \lfloor 20/16 \rfloor \\
&= 10 + 5 + 2 + 1 = 18 \\
v_3(20!) &= \sum_{i=1}^{2} \left\lfloor \frac{20}{3^i} \right\rfloor \\
&= \lfloor 20/3 \rfloor + \lfloor 20/9 \rfloor = 6 + 2 = 8 \\
v_5(20!) &= \lfloor 20/5 \rfloor = 4 \\
v_7(20!) &= \lfloor 20/7 \rfloor = 2 \\
v_{11}(20!) &= \lfloor 20/11 \rfloor = 1 \\
v_{13}(20!) &= \lfloor 20/13 \rfloor = 1 \\
v_{17}(20!) &= \lfloor 20/17 \rfloor = 1 \\
v_{19}(20!) &= \lfloor 20/19 \rfloor = 1
\end{aligned}$$

- Finally, we use $[18, 8, 4, 2, 1, 1, 1, 1]$ to write down the prime factorization of 20! as

$$20! = 2^{18} \times 3^8 \times 5^4 \times 7^2 \times 11^1 \times 13^1 \times 17^1 \times 19^1$$

Notice in Example 4.4.14 that when $p \in \{5, 7, 11, 17, 19\}$ a small optimization can be made. In these cases we have

$$p > \sqrt{20} \implies \left\lfloor \log_p 20 \right\rfloor = 1 \,.$$

Since $\left\lfloor \log_p 20 \right\rfloor$ is the upper limit of the summation $\sum_{i=1}^{\left\lfloor \log_p 20 \right\rfloor}$, so that in these cases, there is exactly one term in the summation, and consequently $v_p(20!) = \lfloor 20/p \rfloor$.

$$
\begin{aligned}
v_5(20!) &= \sum_{i=1}^{1} \left\lfloor \frac{20}{5^i} \right\rfloor = \lfloor 20/5 \rfloor = 4 \\
v_7(20!) &= \sum_{i=1}^{1} \left\lfloor \frac{20}{7^i} \right\rfloor = \lfloor 20/7 \rfloor = 2 \\
v_{11}(20!) &= \sum_{i=1}^{1} \left\lfloor \frac{20}{11^i} \right\rfloor = \lfloor 20/11 \rfloor = 1 \\
v_{13}(20!) &= \sum_{i=1}^{1} \left\lfloor \frac{20}{13^i} \right\rfloor = \lfloor 20/13 \rfloor = 1 \\
v_{17}(20!) &= \sum_{i=1}^{1} \left\lfloor \frac{20}{17^i} \right\rfloor = \lfloor 20/17 \rfloor = 1 \\
v_{19}(20!) &= \sum_{i=1}^{1} \left\lfloor \frac{20}{19^i} \right\rfloor = \lfloor 20/19 \rfloor = 1
\end{aligned}
$$

A further optimization is that when $20 \geq p > \frac{20}{2} = 10$ then $\lfloor 20/p \rfloor = 1$. In partciular, $p \geq 11 \implies \lfloor 20/p \rfloor = 1$, in which case the value of $v_p(20!)$ can be known immediately.

$$
\begin{aligned}
v_{11}(20!) &= \lfloor 20/11 \rfloor = 1 \\
v_{13}(20!) &= \lfloor 20/13 \rfloor = 1 \\
v_{17}(20!) &= \lfloor 20/17 \rfloor = 1 \\
v_{19}(20!) &= \lfloor 20/19 \rfloor = 1
\end{aligned}
$$

We can optimize `legendre_expansion` to avoid additional function calls including avoiding computation of logarithms.

Listing 4.4.15 (*Optimized version of* `legendre_expansion`)

```
def legendre_expansion(n):
  m = min(ceil(n / 2.0) + 1,
          n + 1)
  return [ q for p in range(2,m)
           # i.e., if p is prime
           if p == smallest_factor(p, 2, p)
           for q in [p] * legendre_exponent(n, p)] +
         # now just append the primes from m to n
         [ p for p in range(m, n+1)
           if p == smallest_factor(p, 2, p)]
```

4.5 Programming Challenges

Complete the exercises in the file `algebra/recursion.py`. Then test with the tests in the file `tests/recursion_test.py`.

WARNING to complete the `power` function in `algebra/recursion.py` you will need to understand some of the concepts from Chapter 5.

Chapter 5

Divide and Conquer

Chapter Objectives

- Describe the divide-and-conquer strategy: split the problem in half, solve each half recursively, and combine the results.
- Implement fast exponentiation via repeated squaring, achieving $O(\log n)$ multiplications.
- Generalize fast exponentiation to work over any monoid by passing `mult` and `ident` as arguments.
- Implement binary search on a sorted list in $O(\log n)$ comparisons.
- Compute Fibonacci numbers using three approaches: naive recursion, Binet's closed-form formula, and matrix exponentiation via `power`.
- Implement the `fibonacci` programming assignment.

In this chapter we look at a special class of recursive algorithms called *Divide and Conquer*.

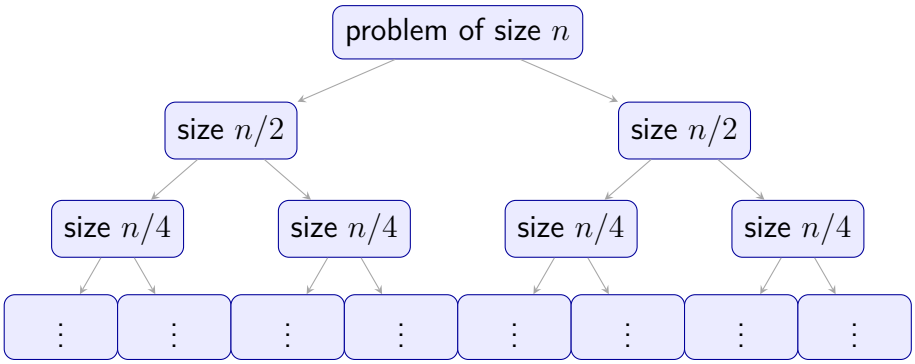


Figure 5.1: Divide and Conquer: recursively split the problem in half until it is trivial

5.1 Limitations of Recursion

In Chapter 4 we concentrated on solving mathematical problems using recursion. Recursion is a powerful tool which every programmer should understand, because often recursive algorithms are cleaner, easier to understand, and often shorter in terms of lines of code.

Nevertheless, there is a significant shortcoming of the Python language which limits the types of problems which can be solved using recursion. Every recursive call allocates memory on a stack, and when this stack is exhausted, the program will fail. The exact limit depends on many things, including the amount of memory of your computer.

We now look at a particular problem for which the Python language fails. In Section 4.3.5 we saw the function `sum_list` which we repeat here. The function works well for small lists, but strategy will fail for lists beyond a certain length.

Listing 5.1.1

```
1 def sum_list(numbers):
2   if not numbers:
3     return 0
4   else:
5     return numbers[0] + sum_list(numbers[1:])
```

Running the program on varying length lists, we see results:

Listing 5.1.2

```
>>> sum_list([0] * 2984)
0

>>> sum_list([0] * 2985)
Traceback (most recent call last):
  File "/usr/local/lib/python3.9/site-packages/IPython/core/
   interactiveshell.py", line 3251, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-20-9f182d9487cb>", line 1, in <module>
    sum_list([0] * 2985)
  File "<ipython-input-4-32a8787e273f>", line 5, in sum_list
    return numbers[0] + sum_list(numbers[1:])
  File "<ipython-input-4-32a8787e273f>", line 5, in sum_list
    return numbers[0] + sum_list(numbers[1:])
  File "<ipython-input-4-32a8787e273f>", line 5, in sum_list
    return numbers[0] + sum_list(numbers[1:])
  [Previous line repeated 2982 more times]
RecursionError: maximum recursion depth exceeded
```

If you try to reproduce this you may see that you need a number larger (or perhaps smaller) than 2985 to trigger the `RecursionError`.

Many computer languages have a feature called *tail call optimization* (TCO) which eliminates this limitation; see Section 4.2.3 for a detailed discussion. Python does not implement TCO.

Despite limitations in the Python language, there are ways to modify a function such as `sum_list` to avoid this annoying limitation. *E.g.*, if each recursive call divides the length of the list in half, then the problem goes away. The program is limited by memory, but not by stack space.

Listing 5.1.3

```
def sum_list(numbers):
  if not numbers:
    return 0
  elif len(numbers) == 1:
    return numbers[0]
  else:
    mid = len(numbers) // 2
    return sum_list(numbers[0:mid]) + sum_list(numbers[mid:])
```

Compare this with the linear `sum_list` in Section 4.3.5: both follow the Recursion Recipe, but the *reduction* strategy differs. Here the reduction halves the list on every call rather than removing one element,

giving a recursion depth of $\log_2 n$ instead of n.

We see now that the program no longer encounters the exception, `RecursionError`, for long lists. Why? Because the recursion depth is no long the length of the list, but rather $\log_2(n)$ where n is the length of the list.

The *base-2 logarithm* $\log_2(n)$ answers the question: how many times must we halve n before reaching 1? For example, $\log_2(8) = 3$ because $8 \to 4 \to 2 \to 1$ takes 3 halvings. Equivalently, $\log_2(n) = k$ means $2^k = n$. Because each recursive call splits the list in half, the depth of recursion is $\log_2(n)$, not n.

I.e., to sum a list of length 3,000,000 as below, the maximum recursion depth is 22, because $\log_2(3 \times 10^6) \approx 21.5$.

Listing 5.1.4

```
>>> sum_list([0] * 2985)
0

>>> sum_list([0] * 3000)
0

>>> sum_list([0] * 30000)
0

>>> sum_list([0] * 300000)
0

>>> sum_list([0] * 3000000)
0
```

5.2 Fast Exponentiation

The `slow_power` function in Section 4.3.4 needs $n-1$ multiplications to compute a^n. There is a better way which requires far fewer multiplications.

$$a^n = \begin{cases} 1 & ;\text{if } n = 0 \\ a & ;\text{if } n = 1 \\ (a^2)^{n/2} & ;\text{if } n > 1 \text{ and } n \text{ even} \\ a \times a^{n-1} & ;\text{if } n > 1 \text{ and } n \text{ odd} \end{cases} \tag{5.1}$$

We can implement `fast_power` as below.

Listing 5.2.1

```
def fast_power(a, n):
  if n == 0:
    return 1
  elif n == 1:
    return a
  elif n % 2 == 0:
    return fast_power(a*a, n // 2)
  else:
    return a * fast_power(a, n - 1)
```

Trace the Recursion Recipe: there are two termination cases ($n = 0$ and $n = 1$) and two reduction strategies — halving n when even, decrementing by one when odd. Notice the bookkeeping differs too: the even branch passes a^2 into the recursive call (absorbing the work into the argument), while the odd branch multiplies the result by a afterward.

The `fast_power` function needs between $\log_2 n$ and $2 \log_2 n$ to compute.

n	$\log_2 n$
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	9
512	10
1024	11

5.3 Fast Generic Exponentiation

Now we would like to generalize `fast_power` so that it can do other things than numerical multiplication. The function, `power`, takes additional arguments to perform the multiplication and the value to return if the exponent is 0.

In Python, functions are values like any other — they can be passed as arguments to other functions. A function that accepts another function as an argument (or returns one as its result) is called a *higher-order function*. In `power(a, n, mult, ident)`, the argument `mult` is a function: to multiply two values `x` and `y`, the `power` function calls `mult(x, y)`. This lets `power` work with any type of multiplication — numbers, strings, matrices — without changing its own code.

Listing 5.3.1

```
def power(a, n, mult, ident):
  if n == 0:
    return ???
  elif n == 1:
    return ???
  elif n % 2 == 0:
    return ???
  else:
    return ???
```

The Recursion Recipe structure of `power` is identical to `fast_power` — compare the two listings. The difference is that `mult` and `ident` replace hard-coded multiplication and the value 1, making the function work for any associative operation without changing the recursive skeleton.

With the `power` function you can perform string concatenation, matrix multiplication and other types of exponentiation. The `power` function is no longer limited to working with numbers as are `slow_power` and `fast_power`.

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

Can you implement `String` concatenation using the power model?

Can you Implement `List` concatenation using the power model?

5.4 Binary Search on List

A binary search algorithm is a *divide and conquer* algorithm. The algorithm often applied to fixed sized arrays (lists in Python). Suppose you

are given a list already in sorted order and you want to find the index of a particular target element in the array.

Listing 5.4.1

```
#        0 1 2  3  4   5   6   7   8   9  10  11  12  13  14
data = [1,3,3, 5, 8, 13, 14, 14, 14, 15, 16, 18, 21, 22, 23]
```

In this example `data` is a list of length 15. If you'd like to find the index of 16 you could start at the beginning, and search `data[0]`, thereafter `data[1]`, etc until you find that `data[10]` contains the value 16. For this approach you'd need to search 11 locations.

Now consider that you look first at `data[7]` and find the value 14. Index 7 was chosen because $\left\lfloor \frac{15}{2} \right\rfloor = 7$. Now you know that if 16 is in the list, then it is between indices 8 and 14; *i.e.*, the target item is not at an index between 0 and 7. In other words you've eliminated half of the potential locations with a single operation.

Since the only valid location is now known to be between 8 and 14, we can look at location $8 + \left\lfloor \frac{15-8}{2} \right\rfloor = 11$. We see that `data[11]` is 18. So we know the target value of 16 can only be between indices 8 and 10. Again, we have eliminated roughly half of the possibilities.

Since `data[9]` is 15, then the only possible location may be `data[10]`.

By this *binary search* approach we need only examine 4 locations to discover the index of the target value. In general the algorithm roughly requires that we read $\log_2 n$ values, where n is the size of the array.

A function to do the search we have just discussed might look like the following. Each recursive call to the local function `loop`, either finds the target element, returning its index, or halves the search space, in other words, eliminates half the remaining search space. This halving continues until the width of the interval is 0, concluding that the target element is not found in the list, so `None` is returned.

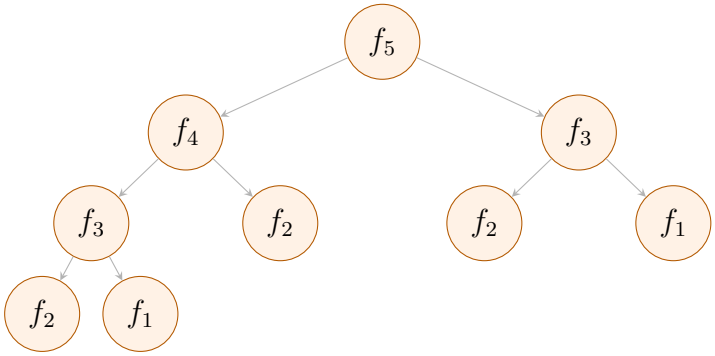


Figure 5.2: Recursive call tree for f_5: subproblems f_3 and f_2 are recomputed multiple times

Listing 5.4.2

```
  def binary_search_on_list(data, target):
   def loop(left, right):
       m = left + (right - left) // 2
       v = data[m]
       if v == target:
           return m
       elif left == right:
           return None
       elif v < target:
           return loop(m + 1, right)
       else:
           return loop(left, m - 1)

   return loop(0, len(data))
```

The inner function `loop` follows the Recursion Recipe: two termination cases (target found, or interval collapsed to zero width); reduction halves the search interval each call; recursive calls are `loop(m+1, right)` or `loop(left, m-1)`; no bookkeeping — the index is returned directly.

5.5 Fibonacci Sequence

In this section we look at a naive, direct approach for computing terms of the Fibonacci sequence. In Section 5.5.3 we will look at a faster computation.

Recall the Fibonacci sequence defined as follows:

$$F_n = \begin{cases} 1 & ;\text{if } n = 1 \\ 1 & ;\text{if } n = 2 \\ F_{n-1} + F_{n-2} & ;\text{if } n > 2 \end{cases} \tag{5.2}$$

5.5.1 Naive, Direct Approach

First we want to implement a Python function to return the n'th element of this sequence. The function should closely follow Equation (5.2). In this implementation if n is neither 1 nor 2, then `fibonacci` must call itself twice.

Unlike `factorial` and `gcd`, Fibonacci makes *two* recursive calls per step. Apply the Recursion Recipe (Pattern 4.2.1): two termination cases ($n = 1$ and $n = 2$); two reductions ($n \to n - 1$ and $n \to n - 2$); two recursive calls; bookkeeping sums the two results. This double branching is what causes the exponential call count shown in Figure 5.3 — a key motivation for the faster algorithms in Sections 5.5.3.

Listing 5.5.1

```
def fibonacci(n):
    ...
    return ???
```

If we count the number of times `fibonacci` calls itself we see the profile shown in Figure 5.3.

5.5.2 Fibonacci using Binet's Formula

Binet's formula can be used for computing the nth Fibonacci number. Implement this function and compare the results to the recursive implementation.

$$F(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

One difficulty with Binet's formula is that in a programming language like Python we cannot represent $\sqrt{5}$ exactly, and computations such as

n	`fibonacci(n)`	Number of calls
4	3	5
6	8	15
8	21	41
10	55	109
12	144	287
14	377	753
16	987	1973
18	2584	5167
20	6765	13529
22	17711	35421
24	46368	92735
26	121393	242785
28	317811	635621
30	832040	1664079
32	2178309	4356617
34	5702887	11405773
36	14930352	29860703
38	39088169	78176337
40	102334155	204668309

Figure 5.3: Profile of number of calls to `fibonacci` function

$\frac{1+\sqrt{5}}{2}$ are not exact—there is round-off error. Thus the result computed for $F(n)$ may not be an integer. This can be solved simply by rounding the result, as we know that the theoretical result must be an integer. In Python we can round with the `math.round` function.

5.5.3 Fibonacci as Matrix Multiplication

In Section 5.5 we saw a way to compute the nth Fibonacci number, but to do so we needed to compute all Fibonacci numbers leading up to the nth one. In Section 5.5.2 we saw one shortcut. In this section we look at a another shortcut based on the `power` function implemented in Section 5.2.

We may compute any three consecutive Fibonacci numbers as: F_{n-1}, F_n, and F_{n+1}, where $F_0 = 0$, $F_1 = 1$, $F_2 = 2$, $F_3 = 2$, etc, using the following equality:

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

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

1. How can we represent a 2×2 matrix programmatically?
   There are many possible representations. One simple, straightforward way is to represent the matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ as the Python 4-tuple `(a, b, c, d)`.

2. How can we multiply two matrices?
   The answer depends on the representation. If we use the 4-tuple representation as described above, then the function `matrix_multiply` shown below suffices.

3. How can we compute $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ with approximately $\log_2 n$ matrix multiplications?
   A naive implementation of M^n matrix exponentiation would require $n-1$ multiplications to compute M^n; however, s discussed in Section 5.3, this operation can be done more efficiently using the `power` function presented in that section.

Listing 5.5.2

```
def fibonacci_as_matrix(n):
    ...
    return power(???, ???, ???, ???)
```

`fibonacci_as_matrix` reuses `power` from Section 5.3: the Fibonacci matrix is the monoid element, `matrix_multiply` is the operation, and the identity matrix is the identity element — the same three arguments `power` always requires.

In the homework exercise `fibonacci_as_matrix` you are asked to somehow represent a 2×2 matrix. One easy way to do this is as a 4-tuple. Thus you should be able to implement the `matrix_multiply` function as follows.

Listing 5.5.3

```
    def matrix_multiply(t1, t1):
        (a, b,
         c, d) = x
        (e, f,
         g, h) = y
        return (a*e + b*g, a*f + b*h,
                c*e + d*g, c*f + d*h)
```

5.6 Programming Challenges

Complete the exercises in the file `algebra/fibonacci.py`. Then test with the tests in the file `tests/fibonacci_test.py`.

Warning: some of the functions in `algebra/fibonacci.py` make use of functions defined in `algebra/recursion.py`. Therefore, it is advised, that you complete `algebra/recursion.py` before attempting `algebra/fibonacci.py`.

Chapter 6

Combinatorics

Chapter Objectives

- Compute the binomial coefficient $\binom{n}{k}$ three ways: via the direct factorial formula, via Pascal's triangle recursion, and via the efficient linear-product formula.
- State and apply Pascal's identity $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ and the symmetry identity $\binom{n}{k} = \binom{n}{n-k}$.
- Implement `cancel_factors` to compute $\binom{n}{k}$ by building and cancelling prime-factor lists, avoiding large integer products.
- Use memoization (caching) to avoid redundant recomputation in recursive functions.
- Distinguish integer division from floating-point division and apply the correct form to keep computations exact.
- Implement the `choose` programming assignment.

In this chapter we look at some applications of recursive functions.

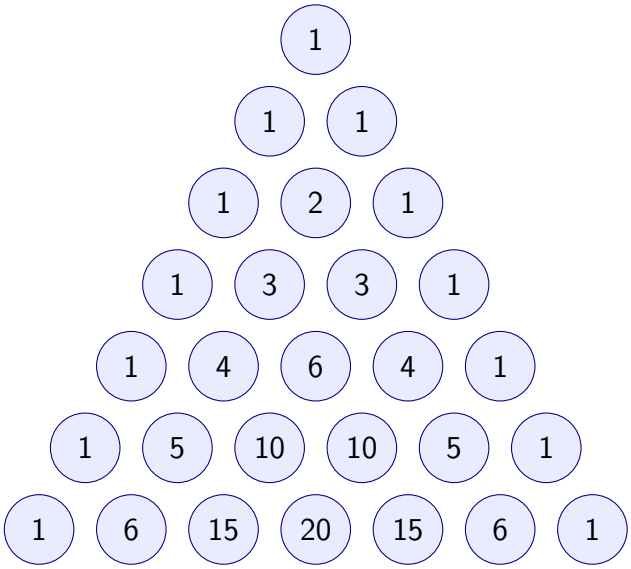


Figure 6.1: Pascal's Triangle: entry (n, k) is $\binom{n}{k}$, the number of k-element subsets of an n-element set

In this section we look at ways to count the number of fixed-sized subsets of a given finite set. Suppose a set, S, has n elements, and we want to count the k-element subsets of S. We will write functions to compute this quantity several different ways.

Example 6.0.1 (*Counting Subsets*)

Before looking at the general case of k-sized subsets of a set with n elements, lets look at the 3-element subsets of a set of size 5.

How many ways can we choose three different items from a set of five elements. Suppose $S = \{a, b, c, d, e\}$. There are 10 subsets of size three.

{a,b,c}	{a,b,d}
{a,b,e}	{a,c,d}
{a,c,e}	{a,d,e}
{b,c,d}	{b,c,e}
{b,d,e}	{c,d,e}

Recall the following formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}\,. \tag{6.1}$$

6.1 Direct Computation

We may compute $\binom{n}{k}$ from Equation 6.1 directly by computing the three factorials and performing the integer division.

Example 6.1.1 (*Direct Computation*)

Counting the 3-element subsets of S of Example 6.0.1

$$\binom{5}{3} = \frac{5!}{3! \times (5-3)!} = \frac{5!}{3! \times 2!} = \frac{5 \times 4 \times 3 \times 2}{3 \times 2 \times 2} = \frac{120}{12} = 10$$

You've already implemented the `factorial` function. You may use it to implement `choose_direct` in a trivially easy way.

Listing 6.1.2

```
def choose_direct(n, k):
  ...
  return ???
```

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? As the following table shows, the number of digits in the factorial grows quickly.

n	$n!$	$\log_{10} n$
2	2	1
4	24	2
6	720	3
8	40320	5
10	3628800	7
12	479001600	9
14	87178291200	11
16	20922789888000	14
18	6402373705728000	16
20	2.432×10^{18}	19
22	1.124×10^{21}	22
24	6.204×10^{23}	24
26	4.032×10^{26}	27
28	3.048×10^{29}	30

6.2 Some Useful Equalities

From Equation (6.1), we can derive a several of base cases. These equations will be useful in the termination conditions of the Python function `choose_pascal` in Section 6.3.

Suppose $n > 0$ and $n \geq k$.

$$\binom{n}{0} = \frac{n!}{0!\ (n-0)!} = \frac{\cancel{n!}}{0!\ \cancel{n!}} = \frac{1}{0!} = 1 \tag{6.2}$$

$$\begin{aligned}\binom{n}{1} &= \frac{n!}{1!\ (n-1)!} \\ &= \frac{n \times \cancel{(n-1)!}}{1 \times \cancel{(n-1)!}} = n\end{aligned} \tag{6.3}$$

$$\begin{aligned}\binom{n}{n-k} &= \frac{n!}{(n-k)!\ (n-(n-k))!} \\ &= \frac{n!}{k!\ (n-k)!} = \binom{n}{k}\end{aligned} \tag{6.4}$$

$$\begin{aligned}\binom{n}{n-1} &= \frac{n!}{(n-1)!\ (n-(n-1))!} \\ &= \frac{n\ \cancel{(n-1)!}}{\cancel{(n-1)!}\ 1!} = n\end{aligned} \tag{6.5}$$

$$\binom{n}{n} = \frac{\cancel{n!}}{\cancel{n!}\ \times (n-n)!} = \frac{1}{0!} = 1 \tag{6.6}$$

6.3 Pascal's Triangle

We may also compute $\binom{n}{k}$ using Pascal's triangle. This approach avoids computing large factorials.

Listing 6.3.1

```
def choose_pascal(n, k):
  ...
  return ???
```

$n=0$						1							
$n=1$					1		1						
$n=2$				1		2		1					
$n=3$			1		3		3		1				
$n=4$		1		4		6		4		1			
$n=5$	1		5		10		10		5		1		
$n=6$	1		6		15		20		15		6		1

The triangular grid of numbers, above, is drawn with each row centered. The array of numbers forms a triangle called Pascal's Triangle. Each row has 1's on the border, and each of the interior entries is the sum of the two numbers above (on the preceding row), above-left + above-right. We can extract the values of $\binom{n}{k}$ from Pascal's Triangle, if we understand that the n'th row contains the numbers $\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots, \binom{n}{n}$ — the k'th element on each row being $\binom{n}{k}$.

If we wish to mathematically describe Pascal's triangle, we need a formula for the entry in each position, *i.e.*, a formula for the entry in the n'th row k'th column. This mathematical formula is easier to reason about if we left-justify each row rather than centering it. We can index the rows by n, starting with 0, and index the columns by k, $0 \leq k \leq n$, so that the entry in the nth row kth column is $\binom{n}{k}$.

n							
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1
	0	1	2	3	4	5	6
				k			

With Pascal's triangle drawn in the left-justified way, it is clear that when $k = 0$ and $k = n$ we have $\binom{n}{k} = 1$, and otherwise $\binom{n}{k}$ is the sum of the entry directly above $\binom{n-1}{k}$ and the entry above and to the left $\binom{n-1}{k-1}$, leading to Equations (6.2), (6.6), and Theorem 6.3.2.

The triangular grid of numbers has 1's on the border, and each the interior entry is the sum of the two number above (on the preceding row), above-left + above.

Theorem 6.3.2 (*Pascal's Identity*)

If $1 \leq k < n$, then

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Proof. We substitute $\binom{n-1}{k-1}$ and $\binom{n-1}{k}$ into Eq (6.1) and then simplfy the sum of fractions.

$$\begin{aligned}
\binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)!\ ((n-1)-(k-1))!} + \frac{(n-1)!}{k!\ ((n-1)-k)!} \\
&= \frac{(n-1)!}{(k-1)!\ (n-k)!} + \frac{(n-1)!}{k!\ (n-k-1)!} \\
&= \left(\frac{k}{k}\right)\frac{(n-1)!}{(k-1)!\ (n-k)!} + \left(\frac{n-k}{n-k}\right)\frac{(n-1)!}{k!\ (n-k-1)!} \\
&= \frac{k\ (n-1)!}{k!\ (n-k)!} + \frac{(n-k)\ (n-1)!}{k!\ (n-k)!} \\
&= \frac{(k+n-k)\ (n-1)!}{k!\ (n-k)!} \\
&= \frac{n\ (n-1)!}{k!\ (n-k)!} \\
&= \frac{n!}{k!\ (n-k)!} = \binom{n}{k}
\end{aligned}$$

Pascal's Identity makes the Recursion Recipe (Pattern 4.2.1) directly readable from the formula. The two recursive calls are $\binom{n-1}{k-1}$ and $\binom{n-1}{k}$; in both, n decreases by 1, which is the reduction guaranteeing termination. The bookkeeping is simply adding the two results together. The termination conditions are the base cases from Section 6.3: $\binom{n}{0} = 1$ (Equation (6.2)), $\binom{n}{1} = n$ (Equation (6.3)), and $\binom{n}{n} = 1$ (Equation (6.6)).

The column of 1's on the left-hand side of Pascal's triangle correspond to the fact that $\binom{n}{0} = 1$, which agrees with Equation (6.2). The 1's along the diagonal of Pascal's triangle correspond to the fact that $\binom{n}{n} = 1$, which agrees with Equation (6.6).

From Theorem 6.3.2 we see that, $\binom{n}{k} = \binom{5}{3}$, the number of 3-element subsets of a set of 5 elements can be partitioned into two sets of subsets, one (left) of size $\binom{n-1}{k-1} = \binom{4}{2} = 6$, and the second (right) of size $\binom{n-1}{k} = \binom{4}{3} = 4$. We see an example of that if we look at the 3-element subsets of $S = \{a, b, c, d, e\}$ from Example 6.0.1. We can partition these 10 subsets into two sets of subsets: those which contain a and those which do not.

Containing a	**Excluding** a
{a,b,c}	{b,c,d}
{a,b,e}	{b,c,e}
{a,b,d}	{b,d,e}
{a,c,e}	{c,d,e}
{a,c,d}	
{a,d,e}	

How many 3-element subsets of S contain a? If we look at the left-hand column above, every entry is of the form $\{a, \ldots\}$ by construction. Ignoring the a, what is left is a 2-element subset of $\{b, c, d, e\}$. So asking how many 3-element subsets of S which contain a is the same as asking how many 2-element subsets of $S \setminus \{a\}$, or how many 2-element subsets of a set of size 4, *i.e.*, $\binom{4}{2}$.

How many 3-element subsets of S exclude a? This is the same as asking how many 3-element subsets are there of $S \setminus \{a\}$, $\binom{4}{3}$.

So we have

$$\binom{5}{3} = \binom{4}{2} + \binom{4}{3} \tag{6.7}$$

We can of course use the same logic to compute $\binom{5}{3}$, *etc.*, with recursive applications of Theorem 6.3.2, terminating with Equations (6.6) and (6.3).

Example 6.3.3 (*Computation with Pascal's Triangle*)

The right-hand side of (6.7) can be computed as follows.

$$\binom{4}{2} = \binom{3}{1} + \binom{3}{2} \tag{6.8}$$

$$\binom{4}{3} = \binom{3}{2} + \binom{3}{3} \tag{6.9}$$

The right-hand sides of (6.8) and (6.9) can be computed as follows.

$$\binom{3}{3} = 1 \qquad \text{by (6.6)}$$

$$\binom{3}{2} = \binom{2}{1} + \binom{2}{2} \tag{6.10}$$

$$\binom{3}{1} = 3 \qquad \text{by (6.3)} \tag{6.11}$$

Finally, the right-hand side of (6.10) can be computed as follows.

$$\binom{2}{2} = 1 \qquad \textit{by (6.6)}$$

$$\binom{2}{1} = 2 \qquad \text{by (6.3)} \tag{6.12}$$

If we put the steps together from Example 6.3.3, we have

$$\begin{aligned}
\binom{5}{3} &= \binom{4}{2} + \binom{4}{3} \\
&= \binom{3}{1} + \binom{3}{2} + \binom{3}{2} + \binom{3}{3} \\
&= 3 + \binom{2}{1} + \binom{2}{2} + \binom{2}{1} + \binom{2}{2} + 1 \\
&= 3 + 2 + 1 + 2 + 1 + 1 \\
&= 10
\end{aligned}$$

6.4 Prime Factor Cancellation

In Section 6.1 we computed $\binom{n}{k}$ directly using product and quotient of factorials, and in Section 6.3 we computed $\binom{n}{k}$ using Pascal's triangle. Another way to compute $\binom{n}{k}$ is similar to how you might compute it by hand.

Example 6.4.1 (*Compute $\binom{7}{3}$ by Cancellation*)

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

Rather than computing $n!$ directly as in Section 6.1, this time we express it as a list of prime factors.

Example 6.4.2

Rather than expressing 6! as 720, we instead express it as

$$\underbrace{(3 \times 2)}_{=6} \times 5 \times \underbrace{(2 \times 2)}_{=4} \times 3 \times 2$$

In Python we express this as the list `[2, 3, 2, 2, 5, 2, 3]`, or sorted as `[2, 2, 2, 2, 3, 3, 5]`.

Having the list of factors for the numerator of $\frac{n!}{k!\ (n-k)!}$, we perform the division indirectly, 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 $\binom{n}{k}$. Note that because $\frac{n!}{k!\ (n-k)!}$ is guaran-

teed to be an integer, we are guaranteed that all of the prime factors of the denominator are also found in the prime factors of the numerator.

The function `factorial_factors` from Section 4.4.4 computes the prime factors of $n!$ without explicitly computing $n!$. You may also use `legendre_expansion` from Section 4.4.5 rather than `factorial_factors`, if you prefer.

We would like to write a Python function, `choose_factors`, to compute $\binom{n}{k}$ using the method of cancellation. Given n and k we need to produce the list of prime factors in the numerator denominator and in the denominator. After the lists have been generated, they should be sorted into increasing order. For example given 7, and 5, we produce the two lists `[7,3,2,5,2,2,3,2]` and `[5,2,2,3,2]` and then sort them to form `[2,2,2,2,3,3,5,7]` and `[2,2,2,2,3,5]`.

Listing 6.4.3

```
def choose_factors(n, k):
    ...
    return ???
```

The function `choose_factors` computes $\binom{n}{k}$ by manipulating the factors, canceling as many as possible, and only multiplying the minimum number of factors together to obtain the desired result.

We can be sure that it is possible to cancel all the terms in the denominator because $\binom{n}{k}$ is guaranteed to be an integer; if some non-cancelable term remained in the denominator then $\binom{n}{k}$ would be a non-whole number fraction.

We know that all the integers in the denominator list are also in the numerator lists, so we just need to remove them from the numerator. Careful, the numbers may appear several times in the denominator list and a different number of times in the numerator list. One way to do this computation is shown in the following illustration.

Example 6.4.4 (*Canceling Common Factors*)

$$
\begin{aligned}
\frac{[2,2,2,2,3,3,5,7]}{[2,2,2,2,3,5]} &= \frac{[\cancel{2},2,2,2,2,3,3,5,7]}{[\cancel{2},2,2,2,2,3,5]} && 2=2,\ \text{cancel } \frac{2}{2}\\
&= \frac{[\cancel{2},2,2,3,3,5,7]}{[\cancel{2},2,2,3,5]} && 2=2,\ \text{cancel } \frac{2}{2}\\
&= \frac{[\cancel{2},2,3,3,5,7]}{[\cancel{2},2,3,5]} && 2=2,\ \text{cancel } \frac{2}{2}\\
&= \frac{[\cancel{2},3,3,5,7]}{[\cancel{2},3,5]} && 2=2,\ \text{cancel } \frac{2}{2}\\
&= \frac{[\cancel{3},3,5,7]}{[\cancel{3},5]} && 3=3,\ \text{cancel } \frac{3}{3}\\
&= \frac{[3,5,7]}{[5]} && 3\neq 5\\
&= 3\times\frac{[5,7]}{[5]} && \text{pop off } 3\\
&= 3\times\frac{[\cancel{5},7]}{[\cancel{5}]} && 5=5,\ \text{cancel } \frac{5}{5}\\
&= 3\times 7 && \text{denominator empty}\\
&= 21
\end{aligned}
$$

Another way (perhaps simpler) to do this in Python is using the `remove` method; *e.g.*, `mylist.remove(3)`. But *WARNING*, `remove` is destructive, it changes the given list, so you need to make a copy; *e.g.*, `mylist.copy()`.

Listing 6.4.5

```
def cancel_factors(numerator_factors, denominator_factors):
  ...
  return ...
```

Hint: given a list, `s`, of number, you my compute their product of the elements of `s` with the `prod(s)`, but you may need `from math import prod`.

6.5 An Efficient, Linear Computation

The function `choose_linear` is built from two theorems: Theorem 6.5.3 (symmetry) and Theorem 6.5.2 (the recursive step), with Equation (6.3), $\binom{n}{1} = n$, as the termination condition.

Listing 6.5.1

```
def choose_linear(n, k):
  ...
  return ???
```

As an example of (6.5.2), suppose we need to compute $\binom{17}{12}$. We chose the minimum of 12 and $17 - 12$ which is 5. We know by (6.4) that $\binom{17}{12} = \binom{17}{5}$, and since $\binom{17}{5}$ is easier to compute than $\binom{17}{12}$, we compute $\binom{17}{5}$.

It turns out that if we already know $\binom{n-1}{k-1}$, then we can easily compute $\binom{n}{k}$. How is $\binom{n}{k}$ related to $\binom{n-1}{k-1}$?

Theorem 6.5.2 (*Efficient Computation of* $\binom{n}{k}$)

If $\binom{n}{k} = \frac{n!}{k!\ (n-k)!}$, and if $n \geq 1$ and $k \geq 1$, then

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

Proof.

$$\begin{aligned}
\binom{n-1}{k-1} &= \frac{(n-1)!}{(k-1)!\ ((n-1)-(k-1))!} \\
&= \frac{(n-1)!}{(k-1)!\ (n-k)!} && (6.13)\\
\binom{n}{k} &= \frac{n!}{k!\ (n-k)!} \\
&= \frac{n\ (n-1)!}{k\ (k-1)!\ (n-k)!} = \frac{n}{k} \times \frac{(n-1)!}{(k-1)!\ (n-k)!} \\
&= \frac{n}{k}\binom{n-1}{k-1} && \text{by (6.13)}\\
& && (6.14)
\end{aligned}$$

The formula $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$ makes the Recursion Recipe (Pattern 4.2.1) visible in the mathematics itself: the *recursive call* is $\binom{n-1}{k-1}$ (both arguments decrease by 1); the *bookkeeping* is multiplication by $\frac{n}{k}$; and the *termination* is $k = 1$, since $\binom{n}{1} = n$ by Equation (6.3).

Theorem 6.5.3 (*Symmetry of* $\binom{n}{k}$)

For all integers $n \geq k \geq 0$,

$$\binom{n}{k} = \binom{n}{n-k}.$$

Proof. Direct substitution into Equation (6.1):

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

The reason the method explained in Theorem 6.5.2 is efficient is because it involves $(k-1)$-many integer multiplications and $(k-2)$-many integer divisions; whereas computation following Theorem 6.3.2 involves

2^k-many computations of $\binom{n-j}{k-i}$ of various forms.

$$\binom{n}{k} = \underbrace{\left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-(k-1)}{1}\right)}_{k\text{-many terms}} \tag{6.15}$$

For reasons explained in Section 6.6, the multiplication in Equation (6.15) must be done from right to left, as follows.

$$\binom{n}{k} = \left(\cdots\left(\left((n-k+1)\left(\frac{n-k+2}{2}\right)\right)\frac{n-k+3}{3}\right)\cdots\frac{n}{k}\right)$$

Together, Theorems 6.5.2 and 6.5.3 determine the structure of `choose_linear`. First, apply Theorem 6.5.3: since $\binom{n}{k} = \binom{n}{n-k}$, replace k with $\min\{k, n-k\}$ before the recursion begins. This bounds the recursion depth by $\lfloor n/2 \rfloor$ rather than k, halving the worst-case number of steps when $k > n/2$. Then apply Theorem 6.5.2 repeatedly, multiplying by $\frac{n}{k}$ and decrementing both arguments by 1 at each step, until the termination case $k = 1$ is reached, at which point $\binom{n}{1} = n$ by Equation (6.3).

Of course $\binom{n}{k}$ *must* be an integer: it counts the number of k-element subsets of an n-element set, and one cannot have a fractional number of subsets. But that combinatorial argument does not explain why the formula agrees. Looking at Equation (6.15), the numerator is

$$n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)\,,$$

a product of k consecutive integers, and the denominator is $k!$. It is not algebraically obvious that this ratio is always a whole number — and the following theorem and corollary prove that it is.

Theorem 6.5.4 ($\binom{n}{k}$ *is an integer*)

For all integers $n \geq k \geq 0$, the product $k!(n-k)! \mid n!$; equivalently $\binom{n}{k} \in \mathbb{N}$.

Proof. We will prove by simple induction:

$$P[n] = \forall n \in \mathbb{N}, 0 \leq k \leq n \implies \binom{n}{k} \in \mathbb{N}\,.$$

- **Base case:** $P[0]$, for $n = 0$, $0 \leq k \leq n \implies k = 0$.

$$\binom{n}{k} = \binom{0}{0} = \frac{0!}{0!\ (0-0)!} = 1 \in \mathbb{N}\,.$$

- **Inductive case:** Prove $P[n] \implies P[n+1]$.

  We assume $P[n]$ holds; assuming $\binom{n}{k} \in \mathbb{N}$ (for $k = 0 \ldots n$), show $\binom{n+1}{k} \in \mathbb{N}$ for $(k = 0 \ldots n+1)$.

  - If $k = 0$ we have

$$\binom{n+1}{0} = \frac{(n+1)!}{(0)!\ ((n+1)-0)!} = \frac{(n+1)!}{(n+1)!} = 1 \in \mathbb{N}$$

    .

  - If $k = n+1$ we have

$$\begin{aligned}\binom{n+1}{n+1} &= \frac{(n+1)!}{(n+1)!\ ((n+1)-(n+1))!} \\ &= \frac{(n+1)!}{(n+1)!0!} = 1 \in \mathbb{N}\end{aligned}$$

    .

  - Finally, let $0 < k \leq n$. Because $P[n]$ holds, we have $\binom{n}{k} \in \mathbb{N}$ and $\binom{n}{k-1} \in \mathbb{N}$. But by Pascal's identity (Theorem 6.3.2) we have $\binom{n+1}{k} = \underbrace{\binom{n}{k}}_{\in\mathbb{N}} + \underbrace{\binom{n}{k-1}}_{\in\mathbb{N}}$ which is the sum of two natural numbers. Thus $\binom{n+1}{k} \in \mathbb{N}$.

Corollary 6.5.5

If $0 < k < n$, then $(n-k+1)(n-k+2)(n-k+3)\cdots n$ is divisible by $k!$.

Proof. First we recognize that

$$(n-k+1)(n-k+2)(n-k+3)\cdots n = \frac{n!}{(n-k)!},$$

and we wish to show that $\frac{\frac{n!}{(n-k)!}}{k!} = \frac{n!}{k!\ (n-k)!}$ is an integer. This is a restatement of Theorem 6.5.4.

Example 6.5.6

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

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

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

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

6.6 Integer vs. Floating-Point Division

What is the correct way to perform the computation indicated in Equation (6.16).

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}. \tag{6.16}$$

The result $\binom{n}{k}$ is necessarily an integer, but $\frac{n}{k}$ may not be an integer. Therefore, to avoid floating point computation, we must first compute $\binom{n-1}{k-1}$ which is an integer, then multiply by n and divide by k.

Use integer division (`//`), and divide *only after* multiplying: k need not divide $\binom{n-1}{k-1}$ alone, nor need it divide n alone, but it is guaranteed to divide $n \times \binom{n-1}{k-1}$. Dividing before multiplying loses the remainder and produces a wrong or non-integer result.

$$\binom{n}{k} = \frac{n \times \binom{n-1}{k-1}}{k}\,. \tag{6.17}$$

If we compute $\binom{5}{2}$ as literally indicated in Equation (6.16), then we have the following computation.

$$\binom{5}{2} = \frac{5}{2}\binom{4}{1} = \frac{5}{2} \times 4\,.$$

Now suppose $\frac{5}{2}$ is computed in Python as `5 // 2`. The result will be 2, because integer division truncates the remainder. And if $\frac{5}{2}$ is implemented as `5 / 2`, the result will be 2.5, which is a floating point number rather than an integer. In either case, the result of computing $\binom{5}{2}$ will either be $2 \times 4 = 8$ or $2.5 \times 4 = 10.0$—the first of which is wrong, the second is misleading.

If $\frac{n}{k}$ is computed as a floating point division and then rounded to the nearest integer, then Python will compute $\binom{57}{25}$ as 9,929,472,283,517,788 rather than as 9,929,472,283,517,787, differing in the right-most digit.

Not only does floating point division result in a misleading type, but in some extreme cases it leads to an incorrect result.

However, if we compute $\binom{5}{2}$ as indicated in Equation (6.17), then the entire computation can be done as integer operations, as shown in the following computation.

$$\binom{5}{2} = \frac{5\binom{4}{1}}{2} = \frac{5 \times 4}{2} = \frac{20}{2} = 10$$

6.7 Memoization

Mathematical functions differ from Python function in an important way. Mathematical functions have no side effect. Evaluating $\cos\frac{\pi}{3}$ does not have any effect on the world: it does not print anything, it does not set

any global variables. Additionally every time $\cos \frac{\pi}{3}$ is *evaluated* mathematically, the exact same value is *returned*.

Contrast mathematical functions with Python functions. A Python function may call `print`, or set a global variable. If the computation of the return value of a function in question makes use of a global variable, it might happen that the function returns a different value each time it is called. Consider the function which returns the time-of-day, or a function which returns a randomly selected element of a list.

Some Python functions can be *modeled* as mathematical functions and others cannot be. If the function is modeled by a mathematical function, it can be replaced by a sufficiently large look-up table. 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.

The following is an implementation of the `factorial` function which uses a *dictionary* to memoize the values of previous calls to the `factorial` function. *E.g.*, once `factorial(8)` has been computed once, it need never be computed again; its value can be found in the look-up table `factorial_cache`.

Listing 6.7.1

```
factorial_cache = dict()

def factorial(n):
  global factorial_cache

  if n in factorial_cache:
    return factorial_cache[n]
  elif n < 2:
    return 1
  else:
    print(f"computing factorial({n})")
    factorial_cache[n] = n * factorial(n-1)
    return factorial_cache[n]
```

This implementation of `factorial` remembers values from previous iterations to avoid re-computing them. If we try to evaluate `factorial(6) + factorial(8)`, we will see that when `factorial(8)` is evaluating, it does not need to re-compute `factorial(5)` among others.

Listing 6.7.2

```
In [6]: factorial(6) + factorial(8)
computing factorial(6)
computing factorial(5)
computing factorial(4)
computing factorial(3)
computing factorial(2)
computing factorial(8)
computing factorial(7)
Out[6]: 41040
```

Can we use memoization to make the Fibonacci generator faster?

6.8 Programming Challenges

Complete the exercises in the file `algebra/choose.py`. Then test with the tests in the file `tests/choose_test.py`.

Warning: some of the functions in `algebra/choose.py` make use of functions defined in `algebra/recursion.py`. Therefore, it is advised, that you complete `algebra/recursion.py` before attempting `algebra/choose.py`.

Chapter 7

Polynomials

Chapter Objectives

- Represent polynomials as little-endian coefficient lists, where `coefs[i]` is the coefficient of x^i.
- Implement `poly_degree` recursively following the Recursion Recipe.
- Implement polynomial arithmetic: `poly_equal`, `poly_add`, `polys_add`, `poly_scale`, `poly_mult_monomial`, and `poly_mult`.
- Implement `poly_eval` and `poly_to_function`, explaining why the latter is a higher-order function.
- Implement `poly_by_roots` to construct a polynomial from its roots by repeated multiplication.
- Implement the `polynomial` programming assignment.

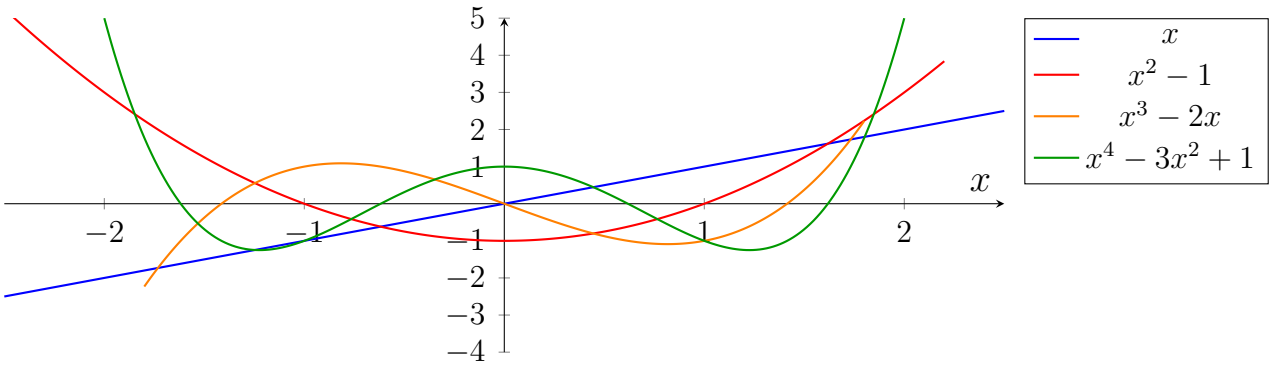


Figure 7.1: Graphs of polynomial functions of degrees 1, 2, 3, and 4

Definition 7.0.1 (*polynomial*)

A *polynomial* of degree n is a function $p : \mathbb{R} \to \mathbb{R}$ by

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \, .$$

Note that we could as well think of the polynomials from $\mathbb{C} \to \mathbb{C}$ or operating on any of several different spaces such as set of matrices or any other set for which $x^0, x, x^2, \ldots, x^n$ makes sense. For this chapter we will only consider polynomials acting on real numbers.

From the definition we see that a polynomial of degree n is completely determined by $n + 1$ coefficients: $a_0, a_1, \ldots, a_n$.

We can also determine an n-order polynomial by n roots: $r_1, r_2, \ldots, r_n$ i.e.

$$p(x) = (x - r_1)(x - r_2) \ldots (x - r_n)$$

7.1 Arithmetic on Polynomials

We may add, subtract, and multiply polynomials.

Given two polynomials, we may add the coefficients of matching powers of x. However, multiplication is more complicated.

Example 7.1.1 (*Polynomial Addition and Multiplication*)

$$\begin{aligned} p(x) &= 1 + 2x + 3x^2 \\ q(x) &= 3 - 3x + x^2 + 4x^3 \\ p(x) + q(x) &= 4 - x + 4x^2 + 4x^3 \\ p(x)q(x) &= 3 + 3x + 4x^2 - 3x^3 + 11x^4 + 12x^5 \end{aligned}$$

One thing to be careful of is that if x^n is not mentioned in a polynomial, such as the fact that there is no x^3 term in $p(x)$ above, then to perform the addition we must recognize the term as $0x^n$.

We will further discuss addition of polynomials in Section 7.4, multiplication of polynomial in Section 7.8, and division of polynomials in Section 8.1.

7.2 Representing a Polynomial in Python

We will represent a polynomial in Python as a list of coefficients.

Example 7.2.1 (*Polynomial Representation*)

We will represent

$$\begin{aligned} p(x) &= x^4 - 3x^3 + 2x^2 - 7x + 5 \\ &= 5 - 7x + 2x^2 - 3x^4 + x^4 \end{aligned}$$

as `[5, -7, 2, -3, 1]`, not as `[1, -3, 2, -7, 5]`.

Representing with the most significant coefficient first (on the left) is called *big-endian*, while placing the least significant coefficient digit first (on the left) is called *little-endian*. We will use the *little-endian* convention.

Be careful, this could be a confusing point. We DO NOT represent $p(x)$ as `[1, -3, 2, -7, 5]`. The first element of the list is the constant term; the second element of the list is the linear term, etc. We represent $p(x)$ as `[5, -7, 2, -3, 1]`.

The rule is: `coefs[i]` is the coefficient of x^i.

Index	Term	Example: `[5, -7, 2, -3, 1]`
`coefs[0]`	x^0 (constant)	5
`coefs[1]`	x^1 (linear)	-7
`coefs[2]`	x^2 (quadratic)	2
`coefs[3]`	x^3 (cubic)	-3
`coefs[4]`	x^4 (quartic)	1

Furthermore we choose to represent the zero polynomial as `[0]`, not as `[]`. The code we write should take special care to never attempt to produce `[]` as a polynomial.

Given two polynomials (*i.e.*, given two lists of coefficients) we want to write functions to add and multiply the two polynomials, returning each time a list of coefficients.

7.3 Polynomial Equivalence

Before addressing equivalence directly, let's consider how to compute the degree of a polynomial. The degree is largest exponent with a non-zero coefficient, except in one case. The 0 polynomial as degree 0.

Listing 7.3.1

```
def poly_degree(coefs):
  assert len(coefs) > 0, "invalid polynomial"
  if coefs == [0]:
    return 0
  elif coefs[-1] == 0:
    return poly_degree(coefs[0:-1])
  else:
    return len(coefs) - 1
```

`poly_degree` follows the Recursion Recipe (Pattern 4.2.1): the termination cases are `coefs == [0]` (degree 0) and `coefs[-1] != 0` (return `len(coefs) - 1`); the reduction strips the trailing zero, passing `coefs[0:-1]` to the recursive call; there is no bookkeeping—the recursive call's return value is the answer directly.

Now, how can we test polynomial equivalence? We cannot simply use the Python `==` operator to test polynomial equivalence? Why? Because there are multiple ways of representing the same polynomial.

Let's determine whether two polynomials are equal. Clearly the polynomials are equal if the lists of coefficients are equal. However, the two polynomials, p and q, in Example 7.3.2 are also equivalent.

Example 7.3.2 (*Equivalent Polynomials*)

$$p(x) = -2 + 3x - x^2 + x^4 \quad \rightarrow \texttt{[2, 3, -1, 0, 1]}$$
$$q(x) = -2 + 3x - x^2 + x^4 + 0x^5 \quad \rightarrow \texttt{[-2, 3, -1, 0, 1, 0]}$$

To determine whether two lists of coefficients represent the same polynomial, we must determine whether they are equal ignoring trailing zeros.

Listing 7.3.3

```
def poly_equal(coefs1, coefs2):
  ...
  return ???
```

7.4 Polynomial Addition

In Python we may use the `+` operator to add to lists, but it does not add component-wise; rather it appends the lists `[1, 2, 3] + [20, 10]` evaluates to `[1, 2, 3, 20, 10]`. To add two polynomials, given the two lists of coefficients, we wish to add as in Example 7.4.1.

Example 7.4.1 (*Adding Polynomials*)

`poly_add([1, 2, 3], [20, 10])` to produce `[21, 12, 3]`. Why? `[1, 2, 3]` specifies the polynomial $1 + 2x + 3x^2$ and `[20, 10]` specifies $20 + 10x$. The sum of these two polynomials is

$$(1 + 2x + 3x^2) + (20 + 10x) = 21 + 12x + 3x^2,$$

which is represented in Python as `[21, 12, 3]`.

Listing 7.4.2 (*Polynomial Addition*)

```
def poly_add(coefs1, coefs2):
  ...
  return ???
```

If we can add two polynomials using `poly_add`, how can we add a list of zero or more polynomials.

Listing 7.4.3 (*Adding Multiple Polynomials*)

```
def polys_add(coefss):
  ...
  return ???
```

7.5 Polynomial Scaling

By *scaling* we mean to multiply a polynomial by a constant, as opposed to another function or another polynomial.

Example 7.5.1 (*Scaling a Polynomial*)

$$p(x) = -2 + 3x - x^2 + x^4 \quad \rightarrow \texttt{[-2, 3, -1, 0, 1]}$$
$$10p(x) = -20 + 30x - 10x^2 + 10x^4 \quad \rightarrow \texttt{[-20, 30, -10, 0, 10]}$$

Listing 7.5.2

```
def poly_scale(s, coefs):
  ...
  return ???
```

7.6 Polynomial Shifting

By *shifting*, we mean to multiply a polynomial by x^n for some $n > 0$. The simplest shifting is to multiply the polynomial by x, which corresponds to appending a 0 on the left. multiplying by x^2 corresponds to prepending `[0, 0]` on the left.

Example 7.6.1 (*Multiplication by x^n using shifting*)

$$p(x) = -2 + 3x - x^2 + x^4 \quad \rightarrow \texttt{[-2, 3, -1, 0, 1]}$$
$$x \times p(x) = -2x + 3x^2 - x^3 + x^5 \quad \rightarrow \texttt{[0, -2, 3, -1, 0, 1]}$$
$$x^2 \times p(x) = -2x^2 + 3x^3 - x^4 + x^6 \quad \rightarrow \texttt{[0, 0, -2, 3, -1, 0, 1]}$$

We can combine scaling and shifting. Example 7.6.2 demonstrates to multiplying a polynomial by $4x^2$. We must scale by 4 and shift as if multiplying by x^2.

Example 7.6.2 (*Multiplication using scaling and shifting*)

$$\begin{aligned} p(x) &= -2 + 3x - x^2 + x^4 \\ &\rightarrow \texttt{[-2, 3, -1, 0, 1]} \\ 4x^2 \times p(x) &= -8x^2 + 12x^3 - 4x^4 + 4x^6 \\ &\rightarrow \texttt{[0, 0, -8, 12, -4, 0, 4]} \end{aligned}$$

The function `poly_mult_monomial` multiplies a polynomial (list of coefficients) by a monomial, *i.e.*, by a polynomial of the form ax^n. The function has 3 arguments: the coefficients, the degree of ax^n, the coefficient of ax^n.

Listing 7.6.3

```
def poly_mult_monomial(coefs, degree, scalar):
    assert degree >= 0
    return ...
```

7.7 Polynomial Evaluation

We can think of a polynomial as an algebraic object. The set of all such objects form an algebraic structure which supports certain algebraic options such as addition and multiplication. However, we can also think of a polynomial as a function.

Example 7.7.1 (*Evaluating a Polynomial*)

Given $x = 1$ we can *evaluate*

$$p(x) = 3x^3 - x + 3 \tag{7.1}$$

to get $3(1)^3 - (1) + 3 = 5$, or we can evaluate it at $x = -2$ to get $3(-2)^3 - (-2) + 3 = -19$.

What is the best way to represent this in Python? There are two straightforward approaches.

Listing 7.7.2

```
def poly_eval(coefs, x):
  ...
  return ???
```

The function, `poly_eval`, accepts a list of coefficients and a value of x, and computes the value of the polynomial at the given value.

Example 7.7.3 (*Evaluating a Polynomial*)

The polynomial, Equation (7.1), $p(x) = 3x^3 - x + 3$ is represented by the list `[3, -1, 0, 3]`, so we can evaluate $p(1)$ by calling `poly_eval([3, -1, 0, 3], 1)` which should return 5, whereas `poly_eval([3, -1, 0, 3], -2)` should evaluate to -19.

The second approach is to implement the function `poly_to_function`, which does not evaluate the polynomial as does `poly_eval`, but rather returns a unary function which can then be called with the value of x.

Listing 7.7.4

```
def poly_to_function(coefs):
  ...
  return ???
```

As an example of how to use `poly_to_function`, we again take the example of Equation (7.1).

Listing 7.7.5

```
>>> p = poly_to_function([3, -1, 0, 3])
>>> p(1)
5
>>> p(-2)
-19
```

As introduced in Section 1.2.2, a *higher-order function* is one that accepts a function as an argument or returns a function as its result. The function `poly_to_function` is an example of the latter: it takes a polynomial (a list of coefficients) and returns a callable Python function.

`poly_to_function` is a higher-order function (Section 1.2.2): it takes data and returns a function. Once you have a callable, you can pass it to `deriv` or `limit` just like any other Python function—the coefficient representation is hidden behind a clean interface.

7.8 Polynomial Multiplication

The function `poly_mult` multiplies two given two polynomials in terms of their coefficients, returning the coefficients of the polynomial representing their product. This function can be implemented easily by using multiple calls to `poly_mult_monomial`, and finally a single call to `polys_add`.

Listing 7.8.1 (*Polynomial Multiplication*)

```
def poly_mult(coefs1, coefs2):
    partials = [poly_mult_monomial(...)
                for i in range(...)]
    return ...
```

`poly_mult` is built entirely from helpers introduced earlier in this chapter: `poly_mult_monomial` (Polynomial Shifting section) handles one term of the outer loop, and `polys_add` (Listing "Adding Multiple Polynomials") reduces the list of partial products to a single polynomial. Notice the pattern: a complex operation composed from smaller, already-tested pieces.

Multiplication of polynomials is more complicated than addition and scaling. We must loop through one polynomial and multiply each term by all the terms in the other. Since polynomial multiplication is commutative, $p(x)q(x) = q(x)p(x)$ we may choose either polynomial as the outer loop.

$$
\begin{aligned}
p(x) &= 1 + 2x + 3x^2 \\
q(x) &= 3 - 3x + x^2 + 4x^3 \\
p(x)q(x) &= 1 \cdot q(x) + 2x \cdot q(x) + 3x^2 \cdot q(x) \\
q(x)p(x) &= 3 \cdot p(x) - 3x \cdot p(x) + x^2 \cdot p(x) + 4x^3 \cdot p(x)
\end{aligned}
$$

Example 7.8.2 (*Multiplying Polynomials* $p \times q$)

Let's perform both computations to provide an intuition that the two approaches give the same result:

$$\begin{aligned}
p(x)q(x) &= 1 \times q(x) + 2x \times q(x) + 3x^2 \times q(x) \\
&= 1 \times (3 - 3x + x^2 + 4x^3) \\
&\quad + 2x \times (3 - 3x + x^2 + 4x^3) \\
&\quad + 3x^2 \times (3 - 3x + x^2 + 4x^3) \\
&= 3 - 3x + x^2 + 4x^3 \\
&\qquad + 6x - 6x^2 + 2x^3 + 8x^4 \\
&\qquad + 9x^2 - 9x^3 + 3x^4 + 12x^5 \\
&= 3 + (-3x + 6x) + (1x^2 - 6x^2 + 9x^2) \\
&\qquad + (4x^3 + 2x^3 - 9x^3) + (8x^4 + 3x^4) + 12x^5 \\
&= 3 + (-3{+}6)x + (1 - 6{+}9)x^2 + (4{+}2{-}9)x^3 \\
&\qquad + (8{+}3)x^4 + 12x^5 \\
&= 3 + 3x + 4x^2 - 3x^3 + 11x^4 + 12x^5
\end{aligned}$$

Now we multiply $q(x)p(x)$.

Example 7.8.3 (*Multiplying Polynomials* $q \times p$)

$$\begin{aligned}
q(x)p(x) &= 3 \times p(x) - 3x \times p(x) + x^2 \times p(x) + 4x^3 \times p(x) \\
&= 3 \times (1 + 2x + 3x^2) \\
&\quad - 3x \times (1 + 2x + 3x^2) \\
&\quad + x^2 \times (1 + 2x + 3x^2) \\
&\quad + 4x^3 \times (1 + 2x + 3x^2) \\
&= 3 + 6x + 9x^2 \\
&\quad - 3x - 6x^2 - 9x^3 \\
&\quad + x^2 + 2x^3 + 3x^4 \\
&\quad + 4x^3 + 8x^4 + 12x^5 \\
&= 3 + (6-3)x + (9-6+1)x^2 + (-9+2+4)x^3 \\
&= \qquad +(3+8)x^4 + 12x^5 \\
&= 3 + 3x + 4x^2 - 3x^3 + 11x^4 + 12x^5
\end{aligned}$$

So we see that both computations result in $3 + 3x + 4x^2 - 3x^3 + 11x^4 + 12x^5$.

To multiply two polynomials, we can simply create partial sums and add them up. We follow the same scheme as seen in Section 7.1.

$$\begin{aligned}
p(x) &= 1 + 2x + 3x^2 \\
q(x) &= 3 - 3x + x^2 + 4x^3 \\
p(x)q(x) &= 1 \times q(x) + 2x \times q(x) + 3x^2 \times q(x)
\end{aligned}$$

Note that each of $1 \times q(x)$, $2x \times q(x)$, and $3x^2 \times q(x)$ is simply a scaling and a shift, (recall the function `poly_mult_monomial`) which results in three polynomials which need to be added together. Once we have the list of polynomials, we can add them together with a single call to the function `polys_add`.

7.9 Generating a Polynomial given the Roots

A root, r, of a polynomial, $P(x)$ is a value for which the polynomial is zero, $P(r) = 0$. In Chapter 10 we will discuss algorithms for finding the roots, given the polynomial. The inverse problem is straightforward: given the roots, $r_1, r_2, \ldots, r_m$, find a polynomial which has those and only those roots. In fact the polynomial

$$P(x) = (x - r_1)(x - r_2) \ldots (x - r_m)\,,$$

is one such polynomial. Because, for example, if $x = r_2$, then one of the factors is now $(r_2 - r_2) = 0$. Any scalar multiple of $\alpha P(x)$ that polynomial has exactly the same roots.

The Python function `poly_by_roots` computes such a polynomial assuming the scalare $\alpha = 1$. The job of `poly_by_roots` is to compute the list of monomials $(x - r_1)$, $(x - r_2)$, $\ldots$, $(x - r_m)$, and multiply them together (for example using the function `polys_mult`, and return the resulting polynomial. Hint, given a root in variable `r`, the polynomial $(x - r) = -rx^0 + 1x^1$ is expressed in Python by the list `[-r, 1]`.

Listing 7.9.1 (*Polynomial By Roots*)

```
1  def poly_by_roots(rs):
2    ...
3    return ???
```

7.10 Summary

In this chapter we have covered the following topics.

- Review of polynomials from classical algebra perspective
- Polynomial by coefficients vs Polynomial by roots
- Programmatic representation of a polynomial
- Equivalence

- Representing the zero and one polynomials
- Scaling
- Padding and Chopping
- Addition 2-ary vs n-ary
- Shifting
- Multiplication
  - Multiplication by monomial
  - Sum of monomial multiplication
- Evaluation
- Polynomial to function

7.11 Programming Challenges

Complete the exercises in the files `algebra/polynomial.py` and `algebra/multiply.py`. The tests can be found in the files `tests/polynomial_test.py` and `tests/multiply_test.py`.

Chapter 8

Polynomial Euclidian Division

Chapter Objectives

- State the Euclidean Division Theorem for polynomials: for any polynomials f and $g \neq 0$ there exist unique polynomials q (quotient) and r (remainder) with $f = qg + r$ and $\deg r < \deg g$.
- Implement `poly_divide` recursively by applying the Recursion Recipe: identify the four termination cases and the recursive reduction step.
- Implement `divide_out_root` via the iterative synthetic-division algorithm.
- Implement `divide_out_roots` by combining `poly_by_roots` and `poly_divide`.
- Implement the `division` programming assignment.

8.1 Polynomial Quotient and Remainder

Theorem 8.1.1 (*Euclidian Division Theorem for Integers*)

For any positive integers n, d with $d \neq 0$, there are unique integers q, r (quotient and remainder) for which

$$n = dq + r\,,$$

such that $0 \leq r < |d|$.

This type of (quotient, remainder) division, called *Euclidian division* is similar to division on integers.

Example 8.1.2

If we wish to divide 14 by 4, we find that 4 divides 14, 3 times with a remainder of 2. This is because

$$14 = 3 \times 4 + 2\,.$$

Similar to Theorem 8.1.1, there is a corresponding theorem for polynomials. We may also divide one polynomial by another (non-zero) polynomial to obtain a quotient and a remainder.

$$\begin{aligned} N(x) &= x^4 - x^2 + 3x - 2 \\ D(x) &= x^2 + x - 1 \\ \frac{N(x)}{D(x)} &= x^2 - x + 1 + \frac{x-1}{D(x)} \end{aligned}$$

Theorem 8.1.3 (*Euclidian Division Theorem for Polynomials*)

For any pair of polynomials $N(x), D(x)$ with $D(x) \neq 0$, there is unique polynomials $Q(x), R(x)$ (quotient and remainder) for which

$$N(x) = D(x)Q(x) + R(x)\,,$$

such that either $R(x) = 0$ or $\mathit{degree}(R) < \mathit{degree}(D)$.

One way to compute $\frac{N(x)}{D(x)}$ is with polynomial long division. Following are a couple of examples.

Example 8.1.4

We would like to divide $N(x) = x^4 - x^2 + 3x - 2$ by $D(x) = x^2 + x - 1$, to obtain $Q(x)$ and $R(x)$ such that $degree(R) < degree(D)$ and $N(x) = D(x)Q(x) + R(x)$.

$$\begin{array}{r|l}
x^2 + x - 1 & \begin{array}{r} x^2 - x + 1 \\ \hline x^4 - x^2 + 3x - 2 \\ -x^4 - x^3 + x^2 \\ \hline -x^3 + 3x \\ x^3 + x^2 - x \\ \hline x^2 + 2x - 2 \\ -x^2 - x + 1 \\ \hline x - 1 \end{array}
\end{array}$$

So we have $Q(x) = x^2 - x + 1$ and $R(x) = x - 1$.

Example 8.1.5

Suppose we wish to divide $N(x) = x^4 - 3x^3 + 2x^2 - 7x + 5$ by $D(x) = x - 3$ to obtain $Q(x)$ and $R(x)$ such that $N(x) = D(x)Q(x) + R(x)$, we may peform the following tedious computation to obtain $Q(x) = x^2 + 2x - 1$ and $R(x) = 2$.

$$\begin{array}{r|l}
x - 3 & \begin{array}{r} x^3 + 2x - 1 \\ \hline x^4 - 3x^3 + 2x^2 - 7x + 5 \\ -x^4 + 3x^3 \\ \hline 2x^2 - 7x \\ -2x^2 + 6x \\ \hline -x + 5 \\ x - 3 \\ \hline 2 \end{array}
\end{array}$$

8.1.1 Recursive Procedure

The function `poly_divide` (which the student must complete) computes the quotient polynomial and remainder polynomial when dividing a nu-

merator polynomial by a denominator polynomial.

Listing 8.1.6

```
Polynomial = List[float]

def poly_divide(numerator: Polynomial,
                denominator: Polynomial
               ) -> Tuple[Polynomial, Polynomial]:
  denominator = poly_chop(denominator)
  numerator = poly_chop(numerator)

  # if denominator is the 0 polynomial
  if denominator == [0]:
    raise Exception(f"polynomial division by zero: {numerator}/{
     denominator}")

  # if numerator is the 0 polynomial
  elif numerator == [0]:
    return [0], [0]

  # if constant polynomial / constant polynomial
  elif poly_degree(numerator) == poly_degree(denominator) == 0:
    return [numerator[0] / denominator[0]], [0]

  # if degree of numerator < degree of denominator
  elif poly_degree(numerator) < poly_degree(denominator):
    return [0], numerator

  # poly_degree(numerator) >=  poly_degree(denominator)
  else:
    ...
    return ...
```

The `else` part of `poly_divide` is not very difficult if we think of it as a recursive function. Imagine that we want to divide two polynomials into a quotient and remainder, $Q(x)$ and $R(x)$ both polynomials.

$$\begin{aligned}\frac{N(x)}{D(x)} &= \frac{20x^6 + 3x^5 - 4x^2 + 3}{5x^4 + 3x^2 - x + 1} \\ &= Q(x) + \frac{R(x)}{D(x)}\end{aligned}$$

In fact, this can be done in many different ways, but if we require the $degree(R(x))$ be less than $degree(D(x))$, then there is a uniqly determined quotient and remainder.

1. As a first step we just consider $\frac{20x^6}{5x^4} = 4x^2$. So we immediately know that the x^2 coefficient of the quotient, $Q(x)$, is exactly $\frac{20}{5} = 4$. Call this term $S_1(x)$.

$$S_1(x) = 4x^2 \tag{8.1}$$

   To compute $S_1(x)$ we have computed its coefficient and its exponent (*i.e.*, its degree).

   - The coefficient of $S_1(x)$ is the quotient of two other coefficients: the highest degree coefficient of $N(x)$ divided by the coefficient of the highest order coefficient of $D(x)$, *i.e.*, $\frac{20}{5} = 4$.
   - The degree of $S_1(x)$ is simply the difference of the degrees of $N(x)$ and $D(x)$; *i.e.* $\mathit{degree}(S_1(x)) = \mathit{degree}(N(x)) - \mathit{degree}(D(x))$.

2. As a second step, compute

$$\begin{aligned} N_1(x) &= N(x) - D(x) \times S_1(x) \\ &= \left(20x^6 + 3x^5 - 4x^2 + 3\right) - \left(5x^4 + 3x^2 - x + 1\right) \times 4x^2 \\ &= \left(20x^6 + 3x^5 - 4x^2 + 3\right) - \left(20x^6 + 12x^4 - 4x^3 + 4x^2\right) \\ &= 3x^5 - 12x^4 + 4x^3 - 8x^2 + 3 \end{aligned}$$

   Note that $(5x^4 + 3x^2 - x + 1) \times 4x^2$ can be computed in Python as a scale and shift or by a call to `poly_mult_monomial`.

   Also note that by construction, $20x^6$ and $-20x^6$ cancel. *I.e.*, the highest power term of $N(x)$ cancels by construction with the highest power term of $D(x) \times S_1(x)$. This cancelation assures us that that we may *force* this coefficient to 0 even if (especially if) it is computed as some number such as 1.0×10^{-16} because of round-off error.

3. As a third step, compute $Q_1(x)$ and $R(x)$ with a recursive call to

`poly_divide`, computing the quotient and remainder such that

$$\begin{aligned}\frac{N_1(x)}{D(x)} &= \frac{3x^5 - 12x^4 + 4x^3 - 8x^2 + 3}{5x^4 + 3x^2 - x + 1} \\ &= \underbrace{Q_1(x)}_{0.6x-2.4} + \frac{\overbrace{R(x)}^{2.2x^3-0.2x^2-3x+5.4}}{D(x)} \\ &= 0.6x - 2.4 + \frac{2.2x^3 - 0.2x^2 - 3x + 5.4}{D(x)}\end{aligned}$$

So the recursive call to `poly_divide` returns the two values:

$$\begin{aligned}R(x) &= 2.2x^3 - 0.2x^2 - 3x + 5.4 \\ Q_1(x) &= 0.6x - 2.4\end{aligned} \tag{8.2}$$

4. Next, we can compute $Q(x)$.

$$\begin{aligned}Q(x) &= \overbrace{S_1(x)}^{4x^2} + \overbrace{Q_1(x)}^{0.6x-2.4} \\ &= 4.0x^2 + 0.6x - 2.4\end{aligned} \tag{8.3}$$

5. The return value is the tuple $(Q(x), R(x))$ from Equations (8.3) and (8.2).

Apply the Recursion Recipe (Pattern 4.2.1) to `poly_divide`: the termination cases are the four `elif` branches already coded (division by zero, zero numerator, constant divided by constant, or degree of numerator less than degree of denominator); the reduction is step 2, computing $N_1(x) = N(x) - D(x) \times S_1(x)$, which is guaranteed to have strictly lower degree than $N(x)$; the recursive call is step 3; and step 4 assembles the bookkeeping, combining $Q(x) = S_1(x) + Q_1(x)$.

The student might be skeptical that this algorithm works because step 3 (the recursive call) seems magical. Dont panic! The recursive step absolutely sufficies to implement all the necessary looping, provided

the termination conditions are coded correctly. There are several termination conditions which are already enumerated in the code shown for `poly_divide`:

- Cannot divide by 0, throw an exception.
- Otherwise, if the numerator is 0 (the zero polynomial), the quotient and remainder are 0 (the zero polynomial).
- Otherwise, a constant divided by a constant is a constant.
- Otherwise, a denominator such as $1 + x + 0x^2$ can be simplified to $1 + x$.
- Otherwise, if the degree of the denominator is higher than the order of the numerator, ($degree(D(x)) > degree(N(x))$), then the quotent is 0 (the zero polynomial) and remainder is the numerator.

There is a subtle case in which care is needed. When the numerator had degree n and denominator has degree m with $m \leq n$, then quotient is usually of order $n - m$. However, sometimes the leading coefficient of the quotient is 0. For example $\frac{x^4+x^3+x^2+2x+1}{x^3+x+1} = x + 1$; *i.e.*, the coefficient of the x^2 term in the quotient is 0.

Make sure when you implement your `poly_divide` function correctly performs the division $\frac{x^4+x^3+x^2+2x+1}{x^3+x+1} = x + 1$. *I.e.*, dividing `[1,2,1,1,1]` by `[1,1,0,1]` should return a quotient of `[1,1]`, a first order polynomial, rather than a second order polynomial.

8.1.2 Illustration of Recursive Procedure

Trace the Recursion Recipe through this example: the reduction produces $N_1(x) = -x^3 + 3x - 2$, whose degree 3 is strictly less than the degree 4 of $N(x)$, confirming progress toward termination. Each recursive call reduces the degree of the numerator by at least 1, so the recursion is guaranteed to reach the base case ($degree(N(x)) < degree(D(x))$) in a finite number of steps.

Recall Example 8.1.4 where we computed $\frac{N(x)}{D(x)} = \frac{x^4-x^2+3x-2}{x^2+x-1}$. The computation worked as follows:

$$
\begin{array}{r|l}
 & x^2 - x + 1 \\
\hline
x^2 + x - 1 & x^4 - x^2 + 3x - 2 \\
 & -x^4 - x^3 + x^2 \\
\hline
 & -x^3 + 3x \\
 & x^3 + x^2 - x \\
\hline
 & x^2 + 2x - 2 \\
 & -x^2 - x + 1 \\
\hline
 & x - 1
\end{array}
$$

The steps were as follows.

1. Compute $S_1(x) = x^2$, the leading term in the quotient.
2. Compute the subtrahend, $S_1(x) \times D(x)$ by multiplying $x^2 \times D(x) = x^2 \times (x^2 + x - 1)$. We obtained $D(x) \times S_1(x) = x^4 + x^3 - x^2$.
3. Compute the difference, $N_1(x) = N(x) - D(x) \times S_1(x) = (x^4 - x^2 + 3x - 2) - (x^4 + x^3 - x^2) = 0x^4 - x^3 + 3x - 2$.
4. Explicitly remove the $0x^4$ term, resulting in $N_1(x) = -x^3 + 3x - 2$.
5. Make recursive call to divide $\frac{N_1(x)}{D(x)} = \frac{-x^3+3x-2}{x^2+x-1}$ obtaining $Q_1(x) = -x + 1$ and $R(x) = x - 1$.
6. Return the resulting quotient $Q(x) = S_1(x) + Q_1(x) = x^2 - x + 1$ and remainder $R(x) = x - 1$.

But why is the remainder obtained in step 4, the correct remainder to return in step 5? And why is the quotient equal to $Q(x) = S_1(x)+Q_1(x)$?

Because, if we perform the recursive call division as a top level division, we have the quotent $-x + 1$ and remainder $x - 1$. And that is exactly what the recursive call computes.

$$
\begin{array}{r|l}
 & -x + 1 \\
\hline
x^2 + x - 1 & -x^3 + 3x - 2 \\
 & x^3 + x^2 - x \\
\hline
 & x^2 + 2x - 2 \\
 & -x^2 - x + 1 \\
\hline
 & x - 1
\end{array}
$$

8.1.3 Iterative Procedure

In Section 8.1.1 we looked at a procedure to compute Euclidian Division on polynomials as a recursive function. The procedure can also be seen as an iterative function.

We are given

$$N(x) = \left(20x^6 + 3x^5 - 4x^2 + 3\right)$$
$$D(x) = \left(5x^4 + 3x^2 - x + 1\right)$$

And we are asked to compute

$$\frac{N(x)}{D(x)} = \frac{20x^6 + 3x^5 - 4x^2 + 3}{5x^4 + 3x^2 - x + 1}\,.$$

We begin by computing computed $S_1(x)$ and $N_1(x)$ such that

$$\begin{aligned}\frac{N(x)}{D(x)} &= \frac{20x^6 + 3x^5 - 4x^2 + 3}{5x^4 + 3x^2 - x + 1} \\ &= S_1(x) + \frac{N_1(x)}{D(x)}\end{aligned}$$

Next, we compute $S_2(x)$ and $N_2(x)$ such that $\frac{N_1(x)}{D(x)} = S_2(x) + \frac{N_2(x)}{D(x)}$. And then, we compute $S_3(x)$ and $N_3(x)$ such that $\frac{N_2(x)}{D(x)} = S_3(x) + \frac{N_3(x)}{D(x)}$.

In each case, we compute $S_1(x)$, $S_2(x)$, $S_3(x), \ldots, S_k(x)$ so that

$$\mathit{degree}(S_1) > \mathit{degree}(S_2) > \mathit{degree}(S_3) > \cdots \mathit{degree}(S_k)\,.$$

But when does the recursion end? Notice that $\mathit{degree}(N_1) < \mathit{degree}(N(x))$ but $\mathit{degree}(N_1) \geq \mathit{degree}(D)$. We must continue the recursion until we find N_k such that $\mathit{degree}(N_k) < \mathit{degree}(D)$. At which point we will have

$$\frac{N(x)}{D(x)} = S_1(x) + S_2(x) + \cdots + S_k(x) + \frac{N_k(x)}{D(x)}$$

So that

$$Q(x) = S_1(x) + S_2(x) + \cdots + S_k(x) \tag{8.4}$$
$$R(x) = N_k(x) \tag{8.5}$$

We can compute $Q(x)$ and $R(x)$ iteratively by computing the sequence of S_i values and the corresponding N_i values, and then apply Equations (8.4) and (8.5). We need to compute $S_1(x), S_2(x), \ldots, S_k(x)$ and $N_1(x), N_2(x), \ldots, N_k(x)$ until we find $N_k(x)$ such that $degree(N_k(x)) < degree(D(x))$.

To compute the quotient and remainder polynomials iteratively, suppose we have a function *che* which finds the coefficient of the largest non-zero exponent. For example: $che(5x^3 - x + 1) = 5$ and $che(0x^4 + 0x^3 - 4x^2 + x - 2) = -4$.

We may use the following formulas.

$$
\begin{aligned}
N_0(x) &= N(x) \\
\alpha_i &= \frac{che(N_{i-1})}{che(D)} \\
j_i &= degree(N_{i-1}) - degree(D) \\
S_i(x) &= \alpha_i x^{j_i} && \text{if } degree(S_{i-1}) \geq degree(D) \\
N_i(x) &= N_{i-1} - S_i(x)\ D(x) && \text{if } degree(S_{i-1}) \geq degree(D)
\end{aligned}
$$

8.2 Synthetic Division

As alluded to above, *synthetic division* is a special case of polynomial division in the special case that the denominator is of the form $x - r$. This algorithm is more amenable to polynomial division implementation in a computer language such as Python. In Chapter 10, we will only need to divide by linear polynomials such as $x - 3$. So let's look at a slightly simpler example of polynomial long division.

Synthetic Division: Statement of the Problem

We will discuss the algorithm for dividing a general polynomial

$$N(x) = \sum_{i=0}^{n} a_i x^i$$

by a linear polygon

$$D(x) = x - r\,.$$

Such a division results in a quotient polynomial, $Q(x)$ and a remainder polynomial, $R(x)$. Moreover, the remainder is 0 if $x - r$ evenly divides $N(x)$.

Synthetic Division: Example

To find the quotient of $\frac{N(x)}{D(x)} = \frac{x^4-3x^3+2x^2-7x+5}{x-3}$, we may perform the following computation.

$$\begin{array}{r|rrrrr} & 1 & -3 & 2 & -7 & 5 \\ 3 & & 3 & 0 & 6 & -3 \\ \hline & 1 & 0 & 2 & -1 & 2 \end{array}$$

Goal

Determine the values of $u_0 \ldots u_{n-1}$, $v_0 \ldots v_{n-1}$, and w. The value of w is the remainder. If $w = 0$ then $x - r$ exactly divides $N(x)$.

$$\begin{array}{r|ccccccc} & a_n & a_{n-1} & a_{n-2} & a_{n-3} & \ldots & a_1 & a_0 \\ r & & u_{n-1} & u_{n-2} & u_{n-3} & \ldots & u_1 & u_0 \\ \hline & v_{n-1} & v_{n-2} & v_{n-3} & v_{n-4} & \ldots & v_0 & w \end{array}$$

$$\begin{aligned} v_{n-1} &= a_n & u_{n-1} &= r\ v_{n-1} && (8.6)\\ v_{n-2} &= a_{n-1} + u_{n-1} & u_{n-2} &= r\ v_{n-2} \\ v_{n-3} &= a_{n-2} + u_{n-2} & u_{n-3} &= r\ v_{n-3} \\ &\vdots & &\vdots \\ v_k &= a_{k+1} + u_{k+1} & u_k &= r\ v_k && (8.7)\\ &\vdots & &\vdots \\ v_1 &= a_2 + u_2 & u_1 &= r\ v_1 \\ v_0 &= a_1 + u_1 & u_0 &= r\ v_0 \\ w &= a_0 + u_0 && && (8.8) \end{aligned}$$

How to Interpret the Results

The list $[v_0, v_1, \ldots, v_{n-1}]$ represents the quotient polynomial, and remainder is w. The intermediate values, $[u_0, u_1, \ldots, u_{n-1}]$ can be discarded. In fact, it is not even necessary to construct the u list, if you are clever about your implementation.

$$\frac{N(x)}{x-r} = \frac{\sum_{i=0}^{n} a_i x^n}{x-r} = \underbrace{\sum_{i=0}^{n-1} v_i x^i}_{\text{quotient}} + \frac{\overbrace{w}^{\text{remainder}}}{x-r}$$

Summary of Synthetic Division Algorithm

The Python function `divide_out_root` implements the synthetic division algorithm.

Listing 8.2.1

```
def divide_out_root(r, coefs):
  ...
  return ???
```

A hint for how to implement `divide_out_root`

1. Assign `n=len(coefs)-1`.
2. Allocate two lists, `v` and `u` of size n; the initial contents are not important.
3. Assign `v[n-1]` and `u[n-1]` according to Equation 8.6.
4. Loop iterating `k` from `n-2` down to `0`, including `0`.
   - In each iteration, assign `v[k]` and `u[k]` according to Equation 8.7.
5. Finally compute `w` according to Equation 8.8.
6. Return `(v,w)`.

Example 8.2.2 (*Synthetic Division*)

Let's look at this more closely. The coefficients of the numerator are entered on line 1. The negative of the constant term is written on the left.

1. Enter the coefficient and value of r.

	1	-3	2	-7	5
3					

2. Bring down the leading coefficient, $v_{n-1} = v_3 = a_4$.

	1	-3	2	-7	5
3					
	1				

3. Compute u_3 and v_2:

$$u_3 = r\ v_3 = 3 \times 1 = 3$$
$$v_2 = a_3 + u_3 = -3 + 3 = 0$$

	1	-3	2	-7	5
3		3			
	1	0			

4. Compute u_2 and v_1:

$$u_2 = r\ v_2 = 3 \times 0 = 0$$
$$v_1 = a_2 + u_2 = 2 + 0 = 2$$

	1	-3	2	-7	5
3		3	0		
	1	0	2		

5. Compute u_1 and v_0:

$$u_1 = r \; v_2 = 3 \times 2 = 6$$
$$v_0 = a_1 + u_1 = -7 + 6 = -1$$

	1	-3	2	-7	5
3		3	0	6	
	1	0	2	-1	

6. Compute u_0 and w:

$$u_0 = r \; v_1 = 3 \times -1 = -3$$
$$w = a_0 + u_0 = 5 - 3 = 2$$

	1	-3	2	-7	5
3		3	0	6	-3
	1	0	2	-1	2

7. Extract the leading coefficients from row 3: $(1, 0, 2, -1)$. The quotient polynomial is $x^3 + 2x - 1$. The right-most entry in row 3 is the remainder: $w = 2$.

$$\frac{x^4 - 3x^3 + 2x^2 - 7x + 5}{x - 3} = x^3 + 2x - 1 + \frac{2}{x - 3}$$

Again, recall that in Python we represent polynomials starting with the constant term. Thus the representation of $x^3 + 2x - 1$ in Python is `[-1, 2, 0, 1]` which is the *reverse order* to what appears in the synthetic division algorithm.

Ideally the function, `divide_out_roots`, would call `divide_out_root` multiple times. However, there is a problem. Each call to `divide_out_root` can return a remainder due to round-off error. It is not clear how to combine these remainders into a single remainder.

Therefore, the function, `divide_out_roots`, must be implemented

differently. First the function must construct an order-m polynomial from the m-many given roots. Such a polynomial can be created with a call to `poly_from_roots`. Then a single call to `poly_divide` will return the quotient polynomial, and a single remainder.

Listing 8.2.3 (*Dividing out multiple roots*)

```
Polynomial = List[float]

def divide_out_roots(roots:List[float],
                     coefs:Polynomials
    ) -> Tuple[Polynomial, Polynomial]:
  ...
  return ???
```

8.3 Programming Challenges

Complete most of the the exercises in the file `algebra/division.py` and test with the tests in the file `tests/division_test.py`.

Chapter 9

Limits

Chapter Objectives

- State the ε-δ definition of a limit and explain it computationally: making $|f(x) - L|$ arbitrarily small by choosing δ sufficiently small.
- Implement `limit(f, a, epsilon, delta)` by halving δ repeatedly until successive values of f agree to within ε.
- Implement `deriv(f, x, epsilon, delta)` as a higher-order function that calls `limit` with a nested secant-slope function.
- Implement `poly_derivative(coefs)` for the exact symbolic derivative of a polynomial in little-endian form.
- Implement `find_x_intercept` via binary search and `solve_for_x` by reducing to `find_x_intercept` with a shifted function.
- Implement the `limit` programming assignment.

In this chapter we will talk about methods for detecting the value of a function *near* an input value of interest.

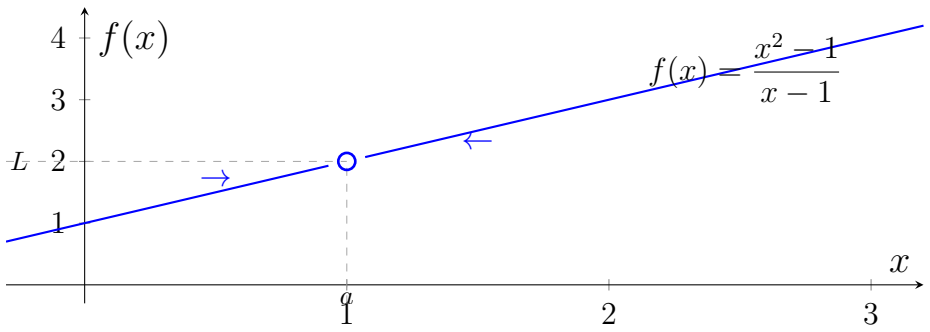


Figure 9.1: $\lim_{x \to a} f(x) = L$ even though $f(a)$ is undefined

9.1 Background

Image you have a Python function named `f`, and when you evaluate the function at 1, an exception is thrown. However, when you evaluate f near 1, numerical values are returned as shown here:

Listing 9.1.1

```
>>>f(1)
Traceback (most recent call last):
  File "...", line 364, in runcode
    coro = func()
  File "<input>", line 1, in <module>
  File "<input>", line 2, in f
ZeroDivisionError: division by zero
>>> f(1.1)
2.1
>>> f(1.01)
2.009999999999999
>>> f(1.001)
2.0009999999999177
>>> f(1.00001)
2.0000100000008274
```

In this example, a reasonable person would think that the value of `f` near 1 is 2. Is it the case that when input values are chosen even closer to 1, that the output values will be closer to 2, or is this an illusion. For example, it might be that output values really get closer to 2.0000000000001 rather than 2.0.

The example above was run with `f` defined as follows:

Listing 9.1.2

```
def f(x):
  return (x*x - 1) / (x - 1)
```

Since $\frac{x^2-1}{x-1} = x + 1$ for all values of x except $x = 1$, then the Python function `f` is able to successfully compute the quotient, and the return value is very close to $x + 1$. However, if the function `g` is defined as follows:

Listing 9.1.3

```
def f(x):
  return ((x - 1)*(x + 1.000000001)) / (x - 1)
```

then we get very similar output. A program will have a difficult time detecting that the values are approaching 2.000000001 rather than approaching 2.0.

Listing 9.1.4

```
>>> g(1.1)
2.100000001
>>> g(1.01)
2.010000001
>>> g(1.001)
2.001000001
>>> g(1.00001)
2.000010001
```

What we roughly mean by

$$\lim_{x \to 1} f(x) = 2.0$$

is that if x is close to 1 then $f(x)$ is close to 2.0. Otherwise stated, the absolute difference $|f(x+\delta) - 2|$ can be made arbitrarily small by choosing sufficiently small values of δ.

Not only do the values of $f(x)$ approaches 2.0 as x approaches 1, but there is no better answer. *E.g.*, if I falsely claim that $f(x)$ approaches 2.001, then I can find a value sufficiently close to 1 (say $1+\delta$) such that the difference $|f(x+\delta) - 2.001|$ cannot be made arbitrarily small, in fact we will not be able to make $|f(x+\delta) - 2.001|$ smaller than 0.001. By this experiment we conclude that

$$\lim_{x \to 1} f(x) \neq 2.001\,.$$

Listing 9.1.5

```
>>> abs(f(1.0001) - 2.001)
0.000900000000607637
>>> abs(f(1.00001) - 2.001)
0.0009899999991724862
>>> abs(f(1.000001) - 2.001)
0.0009989999110993075
>>> abs(f(1.0000001) - 2.001)
0.0009999000799276736
```

However, we may make the difference $|f(x+\delta) - 2.0|$ as small as we like, meaning that

$$\lim_{x \to 1} f(x) = 2.0\,.$$

Listing 9.1.6

```
>>> abs(f(1.01) - 2.0)
0.009999999999998899
>>> abs(f(1.001) - 2.0)
0.0009999999999177334
>>> abs(f(1.0001) - 2.0)
9.99999993922529e-05
>>> abs(f(1.000001) - 2.0)
1.000088900582341e-06
>>> abs(f(1.000001) - 2.0)
0.0
```

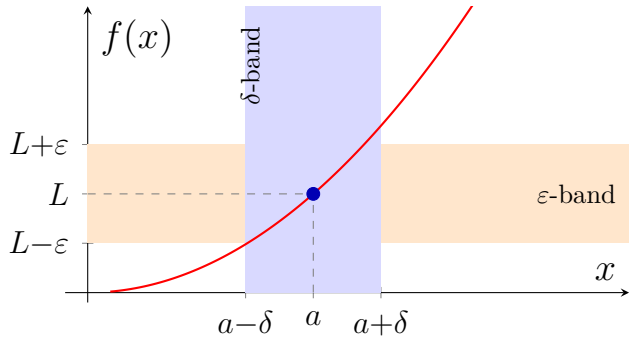
9.2 The Limit

The limit is the essence of calculus. The limit is a way to precisely, rigorously, mathematical describe what we mean by saying *close to*. With the limit, we have a way to talk about what a function *does* at input values close to a fixed value without having to consider what the function *does* at the value in question.

What does it mean to be sufficiently close to a value or arbitrarily close to a value.

Mathematicians need the concept of limit in order to talk about continuity; *i.e.* is the limit of a function at a the same as the value of the function at a. See Definition 9.3.1

We will use the definition of limit, as implemented by the Python



function `limit` to define the derivative (Definition 9.5.1).

In Section 9.1, we built an intuition of what happens to the values returned by a function when we input values close to some values in question. Definition 9.2.1 makes this intuition explicit.

Definition 9.2.1 (*limit*)

$$\lim_{x\to a} f(x) = L$$

means that $\forall\ \varepsilon > 0\ \exists\ \delta > 0$ such that

$$0 < |x - a| < \delta \implies |f(x) - L| < \varepsilon\,.$$

The programmer should think about this definition similar to the examples in Section 9.1. If the value of the limit is L, then we can make the value $|f(x+\delta) - L|$ small as we like; *i.e.*, we can make $|f(x+\delta) - L|$ arbitrarily small by making delta sufficiently small.

If f is a known mathematical function it is sometimes possible to prove that its limit at a given $x = a$ is equal to a known L. For example

Proposition 9.2.2

Let $f(x) = \frac{x^2-1}{x-1}$, then $\lim_{x\to 1} f(x) = 2$.

Proof. Let $\varepsilon > 0$ and let $\delta = \varepsilon$. Now take x such that $0 < |x-1| < \delta$.
We need to show that $|f(x) - 2| \le \varepsilon$.
There are two possibilities: either $0 < x-1 < \delta$ or $0 > 1-x > \delta$.

1. Suppose $0 < x - 1 < \delta$. So $x - 1 \neq 0$.

2. Suppose $0 > 1 - x > \delta$. So $x - 1 \neq 0$.

So in either case $x - 1 \neq 0$.

$$\begin{aligned} |f(x) - 2| &= \left| \frac{x^2 - 1}{x - 1} - 2 \right| \\ &= \left| \frac{(x+1)(x-1)}{x - 1} - 2 \right| \\ &= |x + 1 - 2| && \text{because } x - 1 \neq 0 \\ &= |x - 1| < \delta = \epsilon \end{aligned}$$

9.3 Continuous Functions

Intuitively, some functions have the property that we can draw the graph of the function without lifting the pen. These are what we call *continuous* functions—functions without any sort of gap. That's the intuition, but we can make the definition formally correct.

Definition 9.3.1 (*continuous*)

If a real-valued function $f : [x_0, x_1] \to \mathbb{R}$ has the property that

$$\exists a \in [x_0, x_1] \ \lim_{x \to a} f(x) = f(a)\,,$$

then we say that f is continuous at a.

Furthermore, if f is continuous at a for all $a \in [x_0, x_1]$, then we say that f is continuous on $[x_0, x_1]$.

One of our goals in this chapter (Chapter 9) is to appoximate solutions so an equation such as

$$f(x) = y$$

i.e., for a given y to find the x such $f(x) = y$. To do this we must know that such a solution exists. Some functions are so-called `Darboux` function.

Definition 9.3.2 (*Darboux*)

Suppose $f : [x_0, x_1] \to \mathbb{R}$, and

$$M_0 = \min\{f(x_0), f(x_1)\}$$
$$M_1 = \max\{f(x_0), f(x_1)\}\,.$$

Then, f is called a *Darboux function* if

$$\forall a \in [M_0, M_1], \exists x \in [x_0, x_1], f(x) = a\,.$$

A `Darboux` function is a function which obeys the *Intermediate Value Theorem* 9.3.3.

Theorem 9.3.3 (*Intermediate Value Theorem*)

If $f : [x_0, x_1] \to \mathbb{R}$ is continuous on $[x_0, x_1]$, then f is a Darboux function.

It may be that a function is continuous on an interval *except* at a particular value. For example, $f(x) = \frac{x^2-1}{x-1}$ (from Proposition 9.2.2) is such a function. It is continuous everywhere *except* at $x = 1$.

Another such function is $f(x) = \frac{\sin(x)}{x}$, which obviously cannot be evaluate at $x = 0$ simply by computing $\frac{sin(0)}{0}$. However we find that for all other values of x we have no problem evaluating $\frac{\sin(x)}{x}$.

9.4 Approximating a Function *Near* a Value

Even if we may prove certain limits explicitly given the function (*e.g.*, Proposition 9.2.2), in programming we may not know the internals of the function—the function may be given as a *black-box*. We may only be able to call the function and test its return value. To approximate the limit of a function f at a point a, given a value for ε, we may take ever

smaller values of δ.

$$\delta_1 = \frac{\delta}{2} \tag{9.1}$$

$$\delta_2 = \frac{\delta}{4} \tag{9.2}$$

$$\delta_3 = \frac{\delta}{8} \tag{9.3}$$

$$\vdots \tag{9.4}$$

$$\delta_n = \frac{\delta}{2^n} \tag{9.5}$$

We may continue to test for larger values of n until $|f(a) - L| < \varepsilon$. *I.e.*, we find an integer n such that $|f(x + \delta_n) - f(a + \delta_{n+1})| < \varepsilon$. At that point we may assume that

$$f(a + \delta_{n+1}) - \varepsilon \;\leq\; \lim_{x \to a} f(x) \;\leq\; f(a + \delta_{n+1}) + \varepsilon \,.$$

Another, perhaps more intuitive way of thinking about this is

$$\lim_{x \to a} f(x) = f(a + \delta_{n+1}) \pm \varepsilon \,.$$

From the preceding description, we can implement the Python function `limit`.

Listing 9.4.1

```
1  def limit(f, a, epsilon, delta):
2    ...
3    return ???
```

The function, `limit`, allows us to compute a number which is within ε distance from $\lim_{x \to a} f(x)$. The caller of the function can decide which value of ε to use; *i.e.* the caller can decide which accuracy is needed. The caller must also specify the initial value of δ used in Equations (9.1) thorough (9.5).

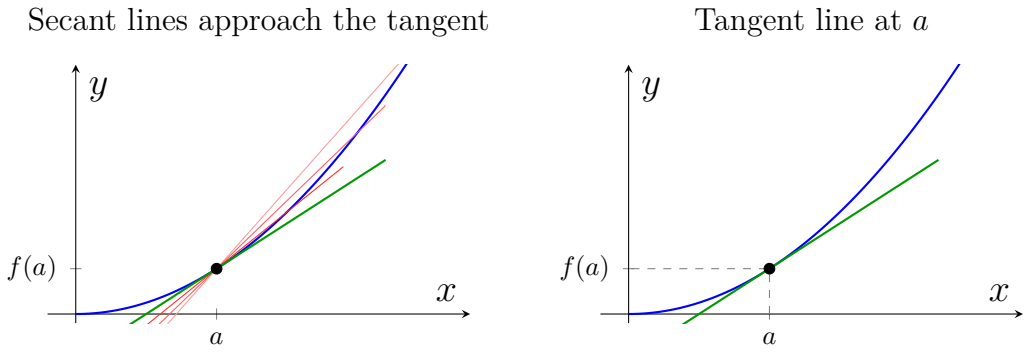


Figure 9.2: We can approximate the tangent to a curve at a point by successive approximations of secant lines. The slope of the tangent line at a point is equal to the derivative evaluated at that x-value.

9.5 The Derivative

In this section we will use the limit as discussed in Sections 9.1 and 9.4 to define the derivative and outline a way to compute it in Python.

The *derivative* of a function at a point is the slope of the tangent line at that point. But what does it mean to say *tangent line at a point*? Two points determine a line; one point does not.

We wish to determine the slope of the line in Figure 9.2 (Right). The way this is done is to first compute the slopes of the secant lines in Figure 9.2 (Left). Each secant line can be completely determined by the two points, $(x, f(x))$ and $(x+h, f(x+h))$. The slope of such a secant line is simply computed by $\frac{\Delta y}{\Delta x}$, Equation (9.6).

$$\begin{aligned} \frac{\Delta y}{\Delta x} &= \frac{f(x+h) - f(x)}{(x+h) - x} \\ &= \frac{f(x+h) - f(x)}{h} \end{aligned} \tag{9.6}$$

If we consider the slope, m, of the secant line to be a function of h, it can be expressed as Equation (9.7), where h represents the distance from x where a second point is designated. The secant line is then the line through $(x, f(x))$ and $(x+h, f(x+h))$.

$$m(h) = \frac{f(x+h) - f(x)}{h} \tag{9.7}$$

Equation (9.7) and Figure 9.2 (Left) lead us to Definition 9.5.1 of the derivative. In the figure, if we look at the secant line through

$$(x+h, f(x+h))\,,$$

then through

$$(x+h', f(x+h'))\,,$$

then through

$$(x+h'', f(x+h''))\,,$$

we see that the slopes of this sequence of lines constitute better and better approximations for the slope of the tangent line in Figure 9.2 (right). This iteration h, h', h'', ..., is represented mathematically as

$$\lim_{h\to 0} m(h)\,.$$

Notice, we cannot simply evaluate $m(0)$ as we would have

$$\frac{f(x+0) - f(x)}{0} = \frac{f(x) - f(x)}{0} = \frac{0}{0}\,.$$

However, we've already seen a function, such as m might have a limit at 0 even if we cannot evaluate $m(0)$ directly.

Definition 9.5.1 (*derivative*)

Let f be a real valued function which is continuous at $a \in \mathbb{R}$, we define another function $\frac{d}{dx}f$ as follows.

$$\frac{d}{dx}f(a) = \lim_{h\to 0} \frac{f(a+h) - f(a)}{h} \tag{9.8}$$

$$= \lim_{x\to a} \frac{f(x) - f(a)}{x - a} \tag{9.9}$$

The two equalities, (9.8) and (9.8), are equivalent; thus you the programmer may use which ever you with to define the `deriv` function.

Listing 9.5.2

```
def deriv(f, x, epsilon, delta):
  def g(h):
    return ...

  return limit(g, ...)
```

`deriv` is a higher-order function (Section 1.2.2): it accepts `f` as an argument, defines a nested function `g` that captures `f` and `x` in its closure, and then delegates to `limit` —reusing the already-implemented function rather than duplicating its logic.

Equation (9.7) is useful in implementing the Python function `deriv`, because in order to call the function `limit` we need a function of a single variable. We can construct the `deriv` by implementing the mathematical expression.

$$\frac{d}{dx}f(x) = \lim_{h \to 0} m(h)$$

9.6 Properties of the Derivative

Viewed as an operator, the derivative has several nice features which we list here but will not prove.

$$\frac{d}{dx}\alpha f(x) = \alpha \frac{d}{dx}f(x) \tag{9.10}$$

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x) \tag{9.11}$$

$$\frac{d}{dx}(f(x)g(x)) = \left(\frac{d}{dx}f(x)\right)g(x) + f(x)\left(\frac{d}{dx}g(x)\right) \tag{9.12}$$

$$\frac{d}{dx}(f \circ g)(x) = \left(\left(\frac{d}{dx}f\right)(g(x))\right)\left(\left(\frac{d}{dx}g\right)(x)\right) \tag{9.13}$$

$$\frac{d}{dx}1 = 0 \tag{9.14}$$
$$\frac{d}{dx}x = 1 \tag{9.15}$$
$$\frac{d}{dx}x^2 = 2x \tag{9.16}$$
$$\frac{d}{dx}x^3 = 3x^2 \tag{9.17}$$
$$\frac{d}{dx}x^n = nx^{n-1} \qquad \forall n \in \mathbb{N} \tag{9.18}$$

Proof of Equations (9.14) *through* (9.18). We wish to prove by induction on n that

$$\frac{d}{dx}x^n = nx^{n-1} \quad \forall n = 0, 1, \ldots .$$

Base case: $n = 0$

$$\begin{aligned}\frac{d}{dx}1 &= \lim_{h\to 0}\frac{1-1}{h}\\ &= \lim_{h\to 0}\frac{0}{h}\\ &= \lim_{h\to 0} 0\\ &= 0\end{aligned}$$

Base case: $n = 1$

$$\begin{aligned}\frac{d}{dx}x &= \lim_{h\to 0}\frac{(x+h)-x}{h}\\ &= \lim_{h\to 0}\frac{h}{h}\\ &= \lim_{h\to 0} 1\\ &= 1\end{aligned}$$

Inductive case: Suppose $\frac{d}{dx}x^n = nx^{n-1}$**, and prove**

$\frac{d}{dx}x^{n+1} = (n+1)x^n$

$$\begin{aligned}
\frac{d}{dx}x^{n+1} &= \frac{d}{dx}(x \cdot x^n) \\
&= x\left(\frac{d}{dx}x^n\right) + \left(\frac{d}{dx}x\right)x^n && \text{by (9.12)} \\
&= x\left(\frac{d}{dx}x^n\right) + (1)x^n && \text{by base case } n = 1 \\
&= x\left(nx^{n-1}\right) + (1)x^n && \text{by inductive assumption} \\
&= nx\left(x^{n-1}\right) + (1)x^n && \text{by (9.10)} \\
&= nx^n + (1)x^n && \text{because } x\ x^{n-1} = x^n \\
&= (n+1)x^n
\end{aligned}$$

From Equations (9.10) and (9.14) we can conclude that the derivative of any constant function is 0. And from Equations (9.10) and (9.15) we can conclude that the derivative of a linear function is its constant slope.

$$\frac{d}{dx}\alpha = 0 \qquad \forall \alpha \in \mathbb{R} \tag{9.19}$$

$$\frac{d}{dx}\beta x = \beta \qquad \forall \beta \in \mathbb{R} \tag{9.20}$$

Additionally, by Equation (9.11), we have

$$\frac{d}{dx}(\beta x + \alpha) = \beta \qquad \forall \alpha, \beta \in \mathbb{R} \tag{9.21}$$

9.7 Polynomial Differentiation

In Section 9.5, we discussed a method for approximating the derivative of a Python function. In the case that the function is a polynomial whose coefficients are known, similar to Equation (9.21) we may compute the exact derivative of the function.

We can compute this derivative by combining the rules for exponents (Equation (9.18)) with the rules for linearity (Equations (9.10) and (9.11)).

Suppose a polynomial $p(x)$ is defined as follows, then we can compute its derivative exactly.

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ &= \sum_{k=0}^{n} a_k x^k \\ \frac{d}{dx} p(x) &= a_1 + 2a_2\ x + 3a_3\ x^2 + \cdots + n\ a_n\ x^{n-1} \\ &= \sum_{k=1}^{n} k a_k\ x^{k-1} \end{aligned} \tag{9.22}$$

Equation (9.22) means that we may compute the derivative of any polynomial simply by manipulating its exponents and coefficients. Moreover, In Python we represent a polynomial by a list of coefficients, $[a_0, a_1, a_2, \ldots, a_n]$, each of whose position in the list is the exponent of x.

Therefore, we can compute $\frac{d}{dx}p(x)$ as the list $[a_1,\ 2 \cdot a_2,\ 3 \cdot a_3,\ \ldots,\ n \cdot a_n]$. This action is implemented by the Python function `poly_derivative`. *I.e.*, we take the coefficient list, shift off the left-most element, to get the list $[a_1, a_2, \ldots, a_n]$, then multiply the shifted elements in sequence by 1, 2, 3, ... respectively.

Listing 9.7.1

```
def poly_derivative(coefs):
    ...
    return ???
```

`poly_derivative` operates on the same little-endian coefficient list representation introduced in Chapter 7: `coefs[i]` is the coefficient of x^i. Knowing the representation, the derivative formula $\sum k\, a_k\, x^{k-1}$ translates directly to a list shift followed by element-wise multiplication by $1, 2, 3, \ldots$

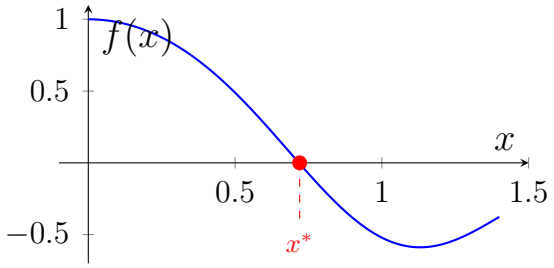


Figure 9.3: Binary search finding an x-intercept of $f(x) = \frac{\cos(2x^2)-x^2}{e^{x^2}}$

Example 9.7.2

Compute the derivative of $p(x) = x^4 - 3x^3 + 2x^2 - 7x + 5$.

$$\begin{aligned}\frac{d}{dx}p(x) &= 4x^{4-1} - 3 \cdot 3x^{3-1} + 2 \cdot 2x^{2-1} - 7x^{1-1} + 0 \times 5x^{0-1} \\ &= 4x^3 - 9x^2 + 4x - 7\end{aligned}$$

The Python function `poly_derivative` effectively computes $\frac{d}{dx}p(x) = 4x^3 - 9x^2 + 4x - 7$ using the following steps:

1. We represent $p(x)$ as the Python list `[5, -7, 2, -3, 1]`.
2. Next, we shift the list to obtain `[-7, 2, -3, 1]`.
3. Finally, we multiply the successive elements by 1, 2, 3, 4 to obtain 1×-7, 2×2, 3×-3, and 4×1, or `[-7, 4, -9, 4]`.

As an *important reminder*, recall the coefficients are in order of increasing power of x. *I.e.*, we represent $p(x) = x^4 - 3x^3 + 2x^2 - 7x + 5$ as the list `[5, -7, 2, -3, 1]` rather than the list `[1, -3, 2, -7 5]`.

9.8 Binary Search with Functions

Assuming the function `f` corresponds to a continuous function which crosses the x-axis somewhere between the values of `left` and `right`, we can find the value where the function crosses the x-axis; at least we can find a function with ε distance from that crossing point. The function `find_x_intercept` does just that.

Listing 9.8.1

```
def find_x_intercept(f, left, right, epsilon):
    ...
    return ???
```

`find_x_intercept` is a binary search (Section 5.4) applied to a continuous function instead of a list: the search space is the interval [`left`, `right`], the midpoint is tested each iteration, and the interval that must contain the answer is chosen for the next iteration. The interval halves in size with each step, so $\log_2\left(\frac{\texttt{right} - \texttt{left}}{\varepsilon}\right)$ iterations suffice.

Consider the function shown in Figure 9.3. The function crosses the x-axis several places. If we know some interval where we are sure there is an x-intercept, we can use a binary search to find it. *E.g.*, we know there is an x-intercept between $x = -1.0$ and $x = 0.0$, because the function is negative at $x = -1.0$ and positive at $x = 0.0$

The binary search strategy is to measure the function at the halfway point in the interval. *E.g.*, let's use the interval $(-1.0, 0.0)$. The midpoint is -0.5. So if we look at $f(-0.5)$, since $f(-0.5) > 0$, then we can recursively search the interval $(-1.0, -0.5)$. With each iteration the size of the interval is divided in half. We can continue the iteration until the size of the interval is ε or less. Figure 9.4 shows how the interval halves in size with each iteration, and how the value of $f(x)$ approaches 0.

The function `solve_for_x` is a simple extension of `find_x_intercept` which rather than finding x such that $g(x) = 0$, it finds x for which $g(x) = y$ for some given y. It does this simply with a call to `find_x_intercept`, but with a function $x \mapsto g(x) - y$.

Listing 9.8.2

```
def solve_for_x(g, a, left, right, epsilon):
    ...
    return ???
```

`solve_for_x` reuses `find_x_intercept`: rather than implementing a new search, it shifts the problem by passing the function $x \mapsto g(x) - y$

left	**right**	$f(left)$	$f(right)$
-1.0	0.0	-1.15309186	1.0
-1.0	-0.5	-1.15309186	0.43346198
-0.75	-0.5	-0.31682302	0.43346198
-0.75	-0.625	-0.31682302	0.08980795
-0.6875	-0.625	-0.10769465	0.08980795
-0.65625	-0.625	-0.00718398	0.08980795
-0.65625	-0.640625	-0.00718398	0.04178609
-0.65625	-0.6484375	-0.00718398	0.01741548
-0.65625	-0.65234375	-0.00718398	0.00514383
-0.654296875	-0.65234375	-0.00101312	0.00514383
-0.654296875	-0.6533203125	-0.00101312	0.00206710

Figure 9.4: Intervals obtained during binary search finding an x-intercept of $f(x) = \frac{\cos\left(2x^2\right) - x^2}{e^{x^2}}$. The x-intercept $\approx -0.654296875 \pm 0.0009765625$.

so that finding a zero of this shifted function is equivalent to solving $g(x) = y$. Building complex operations by composing simpler ones is a recurring theme throughout this course.

9.9 Programming Challenges

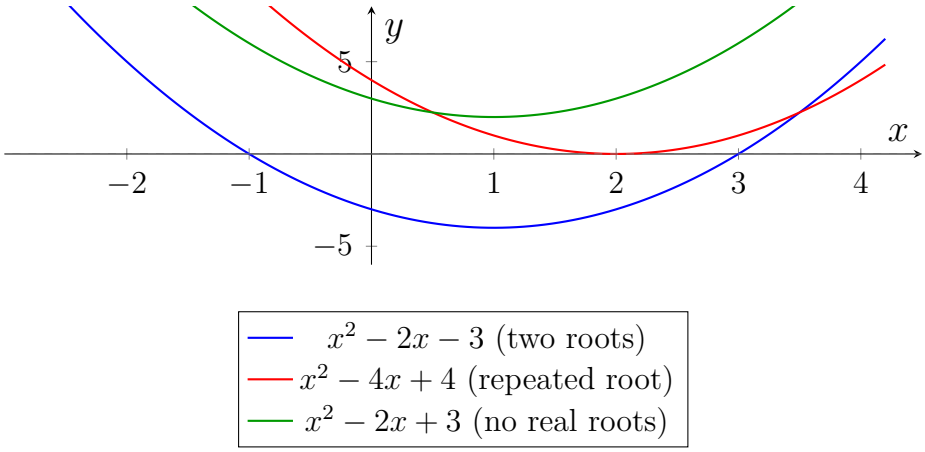
Complete the exercises in the file `algebra/limit.py` and test with the tests in the file `tests/limit_test.py`.

Chapter 10

Finding Roots

Chapter Objectives

- Implement `roots_quadratic` using the quadratic formula, handling positive, zero, and negative discriminants correctly.
- Implement `search_root` by expanding the search interval when `f(left)` and `f(right)` have the same sign, then calling `find_x_intercept` once a sign change is found.
- Implement `poly_roots` by composing `poly_to_function`, `search_root`, `divide_out_root`, and `poly_derivative` to find all real roots of a polynomial iteratively.
- Recognize that `poly_roots` integrates functions from six earlier chapters: a capstone exercise in function composition.
- Implement the `roots` programming assignment.



In this chapter we will apply what we've learned in the previous chapters. We will construct the Python function `poly_roots` in several steps.

Listing 10.0.1

```
def poly_roots(coefs, epsilon):
    ...
    return ???
```

1. In Section 10.3 we will use the quadratic formula to implement `roots_quadratic`.

2. In Section 10.6 we will use a binary search to find one root of a cubic polynomial. Once that root, r, is found, we will use synthetic division to factor $(x - r)$ out of the cubic to form a quadratic polynomial, using `roots_quadratic`.

3. In Section 10.8 we will find one or more roots of a quartic polynomial by binary search for roots between inflection points. We will find the inflection points by finding the roots of the cubic polynomial corresponding to the derivative of the quartic. If such roots r_1, r_2 are detected, then $(x - r_1)$ or $(x - r_2)$ can be factored out of the quartic to produce a cubic or quadratic polynomial, whose roots can further be found using the previously developed functions.

4. In Section 10.9, we will observe that the function used for finding a root of a cubic polynomial can be used to find a root of any odd degree polynomial. Furthermore the method for finding a (or

multiple) roots of a quartic polynomial can be used to search for roots of any even degree polynomial.

5. In Section 10.10, we will discuss three special cases for finding roots.

6. The function `poly_roots` can then be constructed by combining the above steps, along with a trivial case for find the root of a degree-1 polynomial $a_0 + a_1 x$.

10.1 Motivation

Given the roots, $r_1, r_2, \ldots r_n$, we can always generate a polynomial $P(x)$ which has those roots. We can do this simply by computing the product of n-many degree-1 polynomials (n-many monomials):

$$P(x) = (x - r_1)(x - r_2) \cdots (x - r_n) \tag{10.1}$$

We are careful to say, given the roots $r_1, r_2, \ldots r_n$ rather than the *set* of roots $\{r_1, r_2, \ldots r_n\}$ because the roots might not be unique, it makes no sense to say the set of roots is $\{1, 1, 2\}$, because this is a set of size two $\{1, 1, 2\} = \{1, 2\}$.

Here are a few examples of expansions of the form of Equation (10.1).

$$\begin{aligned}
(x-2)(x-1) &= x^2 - 3x + 2 \\
(x-2)(x-1)(x+1) &= x^3 - 2x^2 - x + 2 \\
(x-2)(x-1)(x+1)(x+2) &= x^4 - 5x^2 + 4 \\
(x-2)(x-1)(x+1)(x+2)(x+3) &= x^5 + 3x^4 - 5x^3 - 15x^2 + 4x + 12
\end{aligned}$$

You have already written Python code in the previous chapters which will perform this computation, thus generating the coefficients of such a polynomial, as shown in Listing 10.1.1.

Listing 10.1.1 (*Polynomial Expansion*)

```
>>> print(polys_mult([[-2,1], [-1,1]]))
[2, -3, 1]

>>> print(poly_by_roots([2, 1]))
[2, -3, 1]

>>> print(poly_by_roots([2, 1, -1]))
[2, -1, -2, 1]

>>> print(poly_by_roots([2, 1, -1, -2]))
[4, 0, -5, 0, 1]

>>> print(poly_by_roots([2, 1, -1, -2, -3]))
[12, 4, -15, -5, 3, 1]
```

One might ask whether the polynomial generated by multiplying the roots as in Equation (10.1) is unique. *I.e.*, are there other polynomials of the same degree which have the same roots? It turns out the answer is yes, because if $P(x) = 0$ then $\alpha P(x) = 0$. *I.e.*, every (non-trivial) scalar multiple of an n-degree polynomial has the same n-many roots. Attention: if $\alpha = 0$, then it is true that $\alpha P(x) = 0P(x) = 0$; however, $0P(x)$ is the constant 0, and thus not a polynomial of degree n.

So, the next question one might as is whether an n-degree polynomial with roots, $r_1, r_2, \ldots r_n$, is necessarily of the form

$$\alpha(x - r_1)(x - r_2)\cdots(x - r_n)\,.$$

The answer is: yes.

Theorem 10.1.2 (*Unique Factorization*)

If $P(x)$ is a polynomial of degree n with roots, $r_1, r_2, \ldots r_n$, then there exists α such that

$$P(x) = \alpha(x - r_1)(x - r_2)\cdots(x - r_n)\,.$$

Given the roots of an unknown polynomial, and given α, we can easily generate the coefficients of the polynomial by tediously multiplying the polynomials. The next question is given the coefficients, can we generate the roots. The answer is yes sometimes, and no sometimes. In general this is impossible in the sense that there is no formula which has the sequence of roots input and the sequence of coefficients as output.

The student might say, "Wait! What about the quadratic formula?". Yes, for particular degrees, there are formulas. In particular if the polynomial is of the form $P(x) = ax^2 + bx + c$, then we know the roots are given by the quadratic formula:

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

There is also a formula for explicitly computing the roots of a cubic polynomial: $P(x) = ax^3 + bx^2 + cx + d$. And finally, there is a formula (which fills several pages), which computes the roots of a quartic polynomial: $P(x) = ax^4 + bx^3 + cx^2 + dx + e$. But the Norwegian mathemati-

cian, Nils Abel, [1802 – 1829] proved that no such formula exists for the quintic of the form $P(x) = ax^5 + bx^4 + cx^4 + dx^2 + ex + f$.

The fact that no formula exists to compute the roots of the quintic polynomials is surprising, because there are numerical techniques for approximating them. You will implement some of these techniques in this chapter. In Chapter 11, you'll read about Evariste Galois, who explained that even though there is no general equation for the roots of the quintic; nevertheless, certain classes of quintics (and higher degrees) are solvable, and certain are not. Moreover, Galois was able to characterize these different classes.

10.2 Review Existing Functions

Before diving into the implementation of `poly_roots`, let's review the functions we have developed so far which will be useful in this chapter. One of the goals in implementing `poly_roots` is to re-use as much as possible the functions which have already been implemented. Don't re-invent the wheel.

The student should review each of the following functions which should have been implemented in previous assignments. Without working implementations of these functions, you will not be able to finish `poly_roots`.

10.2.1 `poly_scale`

The function, `poly_scale`, from Section 7.5, can be used to produce a polynomial which has a positive highest order coefficient. For example to convert $1 + x - x^2 - 3x^3$ to $-1 - x + x^2 + 3x^3$. If we know the highest order coefficient is positive, the certain other logic is easier.

Listing 10.2.1

```
1  def poly_scale(s, coefs):
2    ...
3    return ???
```

10.2.2 `poly_to_function`

The function, `poly_to_function` from Section 7.7, will produce a function which is compatible with other functions like `find_x_intercept` and `search_root`.

Listing 10.2.2

```
def poly_to_function(coefs):
  ...
  def f(x):
    ...
    return ???

  return f
```

10.2.3 `divide_out_root`

The function `divide_out_root` (or `divide_out_roots`) from Section 8.2 is used after identifying a root to produce the quotient polynomial, reducing the degree by one each time. In terms of the Recursion Recipe (Pattern 4.2.1), this function provides the *reduction step*: each call yields a strictly simpler polynomial, guaranteeing termination.

Listing 10.2.3

```
def divide_out_root(r, coefs):
  ...
  return ???
```

10.2.4 `poly_derivative`

The function, `poly_derivative` from Section 9.7, is used to produce a new polynomial whose roots specify the *inflection points* of the original polynomial.

Listing 10.2.4

```
def poly_derivative(coefs):
  ...
  return ???
```

10.2.5 `find_x_intercept`

The function, `find_x_intercept` from Section 9.8, along with `poly_to_function` is used to identifying one root in a range for which we know a root exists. For example, we are certain that $p(x) = 5x^3 + x^2 - x + 3$ has a root because

$$\lim_{x \to \infty} 5x^3 + x^2 - x + 3 = \infty$$

and

$$\lim_{x \to -\infty} 5x^3 + x^2 - x + 3 = -\infty\,.$$

The same reasoning applies not only to polynomials of degree 3, but also to any polynomial of odd degree.

Listing 10.2.5

```
def find_x_intercept(f, left, right, epsilon):
  ...
  return ???
```

10.3 Implement the Quadratic Formula

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

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

then

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

Implement the function `roots_quadratic`. 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 $b^2 \geq 4ac$.

Listing 10.3.1

```
def roots_quadratic(a, b, c):
  from math import sqrt
  # warning, this is a poor implementation,
  # the student should improve this function by refactoring
  return [(-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.

We would like to repair the function `roots_quadratic` by refactoring the code above.

- Enhance `roots_quadratic`: 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

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

- It might be that the Δ as computed by b^2-4ac is very close to zero but negative because of floating point round off error. Add an ε to the function parameters, and extend the cases to test $\Delta < 0 - \varepsilon$. What should happen in this case to $\sqrt{\Delta}$?

Example 10.3.2

Consider the following values. If we compute the discriminant ex-

actly, we get 0.

$$
\begin{aligned}
a &= \frac{64}{53} \\
b &= \frac{16}{7} \\
c &= \frac{53}{49} \\
\Delta &= b^2 - 4ac \\
&= \left(\frac{16}{7}\right)^2 - 4\left(\frac{64}{\cancel{53}}\right)\left(\frac{\cancel{53}}{49}\right) = \frac{256}{49} - 4\left(\frac{64}{49}\right) \\
&= \frac{256}{49} - \frac{256}{49} = 0
\end{aligned}
$$

However, if we compute the discriminant from Example 10.3.2 using Python, we get a negative number very close to zero.

Listing 10.3.3

```
>>> a = 64/53
>>> print(a)
1.2075471698113207

>>> b = 16/7
>>> print(b)
2.2857142857142856

>>> c = 53/49
>>> print(c)
1.0816326530612246

>>> b*b - 4 * a * c
-8.881784197001252e-16

>>> sqrt(b*b - 4 * a * c)
Traceback (most recent call last):
  File "...pydevconsole.py", line 364, in runcode
    coro = func()
  File "<input>", line 1, in <module>
ValueError: math domain error
```

Then there are two cases, if $\Delta < 0 - \varepsilon$ (as in Example 10.3.2), and otherwise. Refactor the function so that if there are no real roots it returns an empty list; otherwise return a list of two roots (possibly equal)

in increasing order. *I.e.*, if the function returns $[r_0, r_1]$, assure that $r_0 \leq r_1$.

10.4 Binary Search on Polynomial

Suppose we know that some *event* occurs at a real 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.

$$f(x) = 0 \qquad \text{for some } x_{left} < x \leq x_{mid} \tag{10.2}$$

$$f(x) = 0 \qquad \text{for some } x_{mid} < x < x_{right} \tag{10.3}$$

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 similar in nature to `binary_search_on_list` in Section 5.4, but rather than dividing the list in half, 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 $x_{right} - x_{left} < \varepsilon$.

If $g(x)$ is a polynomial for which we are guaranteed it has a root between `left` and `right`, then we can use the function `find_x_intercept`. However, there is a problem. What if we know there is a root, but we don't know values of `left` and `right`? We need to implement the function `search_root` for this purpose.

The function `search_root` will return the x value where the $p(x) = 0$ or at least where $g(x)$ is 0 for some value between $x - \frac{\varepsilon}{2}$ and $x + \frac{\varepsilon}{2}$. However, what distinguishes from `search_root` from `find_x_intercept` is that if the given polynomial has the same sign at `left` and `right`, then `search_root` expands the search space by recursively searching on `left - d` and `right + d` where `d = abs(right - left) / 2.0`, thus doubling the size of the search window on each recursive call.

Listing 10.4.1

```
def search_root(g, left, right, epsilon):
 assert left < right

 # what if g(left) or g(right) is 0?
 ...

 if g(left) * g(right) > 0:
   # expand the search window
   d = abs(right - left) / 2.0
   return search_root(g, left - d, right + d, epsilon)
 else
   return find_x_intercept(g, left, right, epsilon)
```

10.5 Roots by Synthetic Division

Recall the synthetic division algorithm we presented in Section 8.1.

Given an monomial of the form $x - r$, we can write polynomial $N(x)$ as

$$N(x) = (x - r)\ Q(x) + R(x)\,.$$

If r is a root of $N(x)$ then $R(x) = 0$.

Theorem 10.5.1 (*Factorization induced by roots*)

If r is a root of the degree-n polynomial, $P(x)$, then $(x - r)$ is a factor, and there exists a polynomial, $Q(x)$ of degree $n-1$ such that $P(x) = (x - r)Q(x)$.

Proof. Suppose $P(x)$ is a polynomial of degree n, and suppose that $P(r) = 0$. Now consider the polynomial $P'(x) = P(x + r)$. Therefore $P'(x)$ is of the form

$$P'(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n\,.$$

$P'(0) = P(0 + r) = P(r) = 0$, so 0 is a root of $P'(x)$. Since $P'(0) =$

0, we have

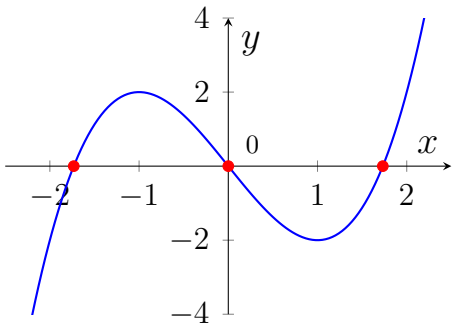
$$\begin{aligned} 0 &= P'(0) \\ &= c_0 + c_1 \times 0 + c_2 \times 0^2 + \cdots + c_n \times 0^n \\ &= c_0. \end{aligned}$$

I.e., $c_0 = 0$. So:

$$\begin{aligned} P'(x) &= 0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \\ &= x(c_1 + c_2 x^1 + \cdots + c_n x^{n-1}) \\ P(x + r) &= P'(x) \\ P(x) &= P'(x - r) \\ &= (x - r) \underbrace{(c_1 + c_2 (x - r)^1 + \cdots + c_n (x - r)^{n-1})}_{\text{a polynomial of degree } n-1} \end{aligned}$$

We may use a technique called synthetic division to compute $Q(x)$ and $R(x)$ given $N(x)$ and a root r.

10.6 Roots of a Cubic Polynomial



Can we combine the techniques we already know 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?
Now we can find the roots of a cubic by

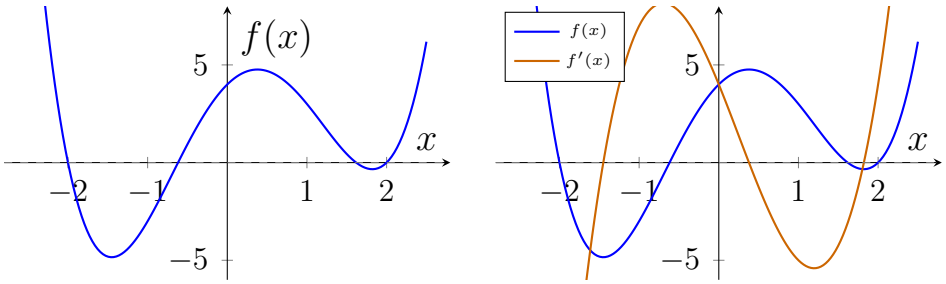


Figure 10.1: Graph of $f(x) = x^4 - x^3 - 5x^2 + 4x + 4$ (left), overlaid with $\frac{d}{dx}f(x) = 4x^3 - 3x^2 - 10x + 4$ (right).

1. use binary search to find a root of the given cubic
2. use synthetic division to *factor out* the monomial leaving a binomial, $Q(x)$
3. use the quadratic formula to find the roots of $Q(x)$.

10.7 Roots of a Quartic Polynomial

The derivative of a quartic (4th degree) polynomial is cubic (3rd degree) polynomial and is given as follows.

$$\frac{d}{dx}\left(x^4 - x^3 - 5x^2 + 4x + 4\right) = 4ax^3 + 3bx^2 + 2cx + d$$

We may find the roots of this cubic polynomial to find the extrema and inflection points of the quartic polynomial.

We may find a root of a quartic by testing the original polynomial at each of the extrema. If we find two extrema e_1 and e_2 where $N(e_1) < 0$ and $N(e_2) > 0$ or where $N(e_1) > 0$ and $N(e_2) < 0$, then we can use a binary search between e_1 and e_2 to find a root r. Once we have a root we can use synthetic division to factor out $(x - r)$ leaving a cubic. We may find the roots of the cubic as in Section 10.6.

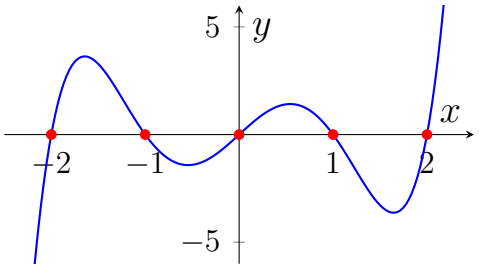
Figure 10.1 shows the graph of a quartic polynomial (left) and overlaid with its derivative (right). Notice that when $\frac{d}{dx}f(x) = 0$ then $f(x)$ has a minimum or maximum. It is between these minima and maxima that $f(x)$ crosses the x-axis. However, it is not guaranteed that $f(x)$ has a

root between every two consecutive extrema. As we see in Figure 10.1, $\frac{d}{dx}f(x)$ has three real roots, while $f(x)$ has only two. If we order the roots of $\frac{d}{dx}f(x)$ as r_0, r_1, r_2, and we perform a binary search between r_0 and r_1, we'll find a root of $f(x)$ because $f(r_0) < 0$ and $f(r_1) > 0$. However, since $f(r_1) > 0$ and $f(r_2) > 0$ we cannot find a root of $f(x)$ between r_1 and r_2.

Listing 10.7.1

```
def find_roots_by_inflection_points(coefs, epsilon):
    return ...
```

10.8 Roots of a Quintic Polynomial



A quintic (5th degree) polynomial is guaranteed to have a root. We can find it (approximately) with a binary search similar to the technique in Section 10.6. If we factor out $(x - r)$ we are left with a quartic, in which case we can use the technique shown in Section 10.7.

10.9 Roots of Higher Degree Polynomials

Any odd degree polynomial has at least one root. The method for finding a root of a cubic or quintic polynomial can be generalized to find such a root. After factoring out $(x - r)$ of a polynomial of degree $2n + 1$ we arrive at a polynomial of degree $2n$.

Furthermore, any even degree polynomial, $(2n > 2)$, can be examined using the technique described in Section 10.7. This method is not guaranteed to find a root because not all even-degree polynomials have a real root. Nevertheless, if the technique is successful, and one or more roots are found, $r_1, r_2, \ldots$, then synthetic division can be used to factor out $(x - r_1), (x - r_2), \ldots$, to arrive at polynomials of lower degree, until

either a first or second degree polynomial is found, which can be solved by a fixed formula.

Listing 10.9.1

```
def poly_roots(coefs, epsilon):
    degree = poly_degree(coefs)
    coefs = poly_chop(coefs)
    if degree == 0:
        return []
    if coefs[-1] == 0:
        return poly_roots(coefs[0:-1], epsilon)
    if coefs[-1] < 0:
        return poly_roots(poly_scale(-1, coefs), epsilon)

    if degree == 1:
        ...

    if coefs[0] == 0:
        ...

    mid_zero_roots = poly_mid_zero_roots(coefs, epsilon)
    if mid_zero_roots:
        ...

    if degree == 2:
        ...

    if degree % 2 != 0:  # odd degree
        ...

    # even degree
    rs = find_roots_by_inflection_points(coefs, epsilon)
    if rs:
        ...

    # otherwise we are unable to find any roots
    return []
```

10.10 Special Cases of Roots

This Sections 10.10.1, 10.10.2, and 10.10.3 deal with some special cases which are straightforward to implement.

10.10.1 Zero as Constant Term

A polynomial which as zero as its constant term necessarily has 0 as a root. Consider the example $x + 3x^2 - 7x^3 - x^4$. If $x = 0$, then all of the terms of the polynomial are zero, so the sum is zero. In Python we represent $x + 3x^2 - 7x^3 - x^4$ as `[0, 1, 3, -7, -1]`. Therefore, the roots can be computed by appending `[0]` (a known root) to the roots polynomial formed when the x is divided out.

$$\frac{x + 3x^2 - 7x^3 - x^4}{x} = 1 + 3x - 7x^2 - x^3 \,.$$

In Python we can obtain $1 + 3x - 7x^2 - x^3$ simply by left-shifting `[0, 1, 3, -7, -1]` to obtain `[1, 3, -7, -1]`.

10.10.2 Rational Roots Test

There is an *easy* way to find all rational roots if the coefficients of the polynomial are all integers. If the polynomial is of the form

$$\alpha_0 + \alpha_1 x^1 + \cdots + \alpha_n x^n \,,$$

and if a root is of the form $r = \pm\frac{a}{b}$ where a and b are both integers, then we know that a is a factor of α_0 and b is a factor of α_n.

Example 10.10.1 (*Rational Roots Test*)

Suppose

$$p(x) = 4 + \cdots + 9x^n \,,$$

then we compute the factors of 4 which are $\{1, 2, 4\}$ and the factors of 9 which are $\{1, 3, 9\}$, and then the only possible rational roots are:

$$\left\{\pm\frac{1}{1}, \pm\frac{2}{1}, \pm\frac{4}{1}, \pm\frac{1}{3}, \pm\frac{2}{3}, \pm\frac{4}{3}, \pm\frac{1}{9}, \pm\frac{2}{9}, \pm\frac{4}{9}\right\}$$

To figure out which one, we simply evaluate the polynomial at each possibility, $p\left(\frac{1}{1}\right)$, $p\left(-\frac{1}{1}\right)$, $p\left(\frac{2}{1}\right)$, $p\left(-\frac{2}{1}\right)$, $p\left(\frac{4}{1}\right)$, $p\left(-\frac{4}{1}\right)$, $p\left(\frac{1}{3}\right)$, $p\left(-\frac{1}{3}\right)$, $p\left(\frac{2}{3}\right)$ ..., and select the ones for which $p(x) = 0$.

Example 10.10.2 (*WARNING: Rational Roots Test*)

Suppose

$$p(x) = 4 + \cdots + 10x^n\,,$$

then we compute the factors of 4 which are $\{1, 2, 4\}$ and the factors of 10 which are $\{1, 2, 5, 10\}$, and then the only possible rational roots are:

$$\left\{\pm\frac{1}{1}, \pm\frac{2}{1}, \pm\frac{4}{1}, \pm\frac{1}{2}, \pm\frac{2}{2}, \pm\frac{4}{2}, \pm\frac{1}{5}, \pm\frac{2}{5}, \pm\frac{4}{5}, \pm\frac{1}{10}, \pm\frac{2}{10}, \pm\frac{4}{10}\right\}$$

But be careful, $\frac{2}{1} = \frac{4}{2}$. So we can't just blindly try them all and collect the ratios for which $p(x) = 0$ as we did in Example 10.10.1. Rather, we must assure that we eliminate redundant ratios.

How do we know in python whether $\frac{a_1}{b_1} = \frac{a_2}{b_2}$ considering that floating point division might not be exact? Instead of checking $\frac{a_1}{b_1} = \frac{a_2}{b_2}$, we should check whether $a_1 b_2 = a_2 b_1$, because if a_1, a_2, b_1, and b_2 are integers then we rely on the fact that integer multiplication is exact.

A function implementing this test would return the list of rational roots found, or an empty list `[]` if the coefficients are not all integers or no rational roots exist.

Another thing to be aware of is that a root might appear with multiplicity greater than 1. For example

$$(x-5)(x-5)(x-\frac{2}{5}) = \frac{1}{5}(x-5)(x-5)(5x-2)$$

has root 5 with multiplicity 2, so such a function must test the multiplicity of each candidate root.

Example 10.10.3 (*Rational Roots Test*)

Find a rational root of $p(x) = 15 + 7x - 2x^2 + 6x^3$.

The factors of $c_0 = 15$ are $b \in \{1, 3, 5, 15\}$, and the factors of $c_3 = 6$ are $a \in \{1, 2, 3, 6\}$.

The possible positive rational roots $(\frac{b}{a})$ are

$$\left\{\frac{1}{1}, \frac{3}{1}, \frac{5}{1}, \frac{15}{1}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{15}{2}, \frac{1}{3}, \cancel{\frac{3}{3}}, \frac{5}{3}, \cancel{\frac{15}{3}}, \frac{1}{6}, \cancel{\frac{3}{6}}, \frac{5}{6}, \cancel{\frac{15}{6}}\right\}$$

So we need to test all these ratios and their negatives until we

find $p(x) = 0$.

$$\begin{aligned} p(1) &= 15 + 7(1) - 2(1)^2 + 6(1)^3 = 26 \\ p(-1) &= 15 + 7(-1) - 2(-1)^2 + 6(-1)^3 \\ &= 15 - 7 - 2 - 6 = 0 \end{aligned}$$

So now we can divide $p(x)$ by $x + 1$. We could use synthetic division as shown in Section 8.2.

$$\begin{array}{r|rrrr} & 6 & -2 & 7 & 15 \\ -1 & & -6 & 8 & -15 \\ \hline & 6 & -8 & 15 & 0 \end{array}$$

So we have a remainder of zero, which verifies that indeed -1 is a root.

$$\frac{15 + 7x - 2x^2 + 6x^3}{x + 1} = 6x^2 - 8x + 15$$

At this point we may continue with the remaining potential rational roots using the updated polynomial.

There is a small simplification/optimization. We know a polynomial of degree n has at most n real roots, so once we have found n roots we can abandon the search.

10.10.3 Mid Zero Roots

Consider the two polynomials: $p_1(x) = x^{12} - 1$ and $p_2(x) = x^{13} + 1$. By inspection we see that $p_1(1) = p_1(-1) = 0$, and that $p_2(-1) = 0$.

In general there are two cases when we can trivially find one or two roots of $ax^n + b$. Both cases involve computing $\sqrt[n]{\frac{b}{a}}$.

In Python, we can compute $\sqrt[n]{\left|\frac{b}{a}\right|}$ as `abs(b/a) ** (1/n)`.

If you attempt to compute $\sqrt[n]{\frac{b}{a}}$ in Python in a case where $\frac{b}{a} < 0$, you'll get a complex number. For our purposes, we would like to avoid complex numbers and search only for real roots.

Listing 10.10.4

```
>>> -5.3 ** (1/6)
-1.3204216755510825

>>> (-5.3) ** (1/6)
(1.1435187147348513+0.6602108377755411j)
```

If the polynomial $p(x) = ax^n + b$ has even degree, and a and b have opposite sign (one positive and one negative), then by inspection we know that two roots are $x = \pm\sqrt[n]{\frac{-b}{a}}$. Moreover, these are the only real roots.

We can only compute $\sqrt[2k]{\frac{-b}{a}}$ in the case that $\frac{-b}{a} \geq 0$. Thus a and b must have opposite signs, in which case, $\frac{-b}{a} = \left|\frac{b}{a}\right|$.

Theorem 10.10.5 (*Roots of trivial even degree polynomial*)

If n is even, and a and b have opposite sign, then the polynomial $ax^n + b$ has two real roots of the form

$$x = \pm\sqrt[n]{\left|\frac{b}{a}\right|}.$$

Example 10.10.6 (*Trivial even degree roots*)

$13x^8 - 5$ has roots $\pm\sqrt[8]{\frac{5}{13}}$. Because

$$\begin{aligned} 13x^8 - 5 &= 13\left(\sqrt[8]{\frac{5}{13}}\right)^8 - 5 \\ &= 13\left(\frac{5}{13}\right) - 5 \\ &= 5 - 5 = 0 \end{aligned}$$

And also,

$$\begin{aligned} 13x^8 - 5 &= 13\left(-\sqrt[8]{\frac{5}{13}}\right)^8 - 5 \\ 13x^8 - 5 &= 13\left(\sqrt[8]{\frac{5}{13}}\right)^8 - 5 \\ &= 13\left(\frac{5}{13}\right) - 5 \\ &= 5 - 5 = 0 \end{aligned}$$

We can compute $\sqrt[2k+1]{\frac{-b}{a}}$ in both cases $\frac{-b}{a} > 0$ and also $\frac{-b}{a} < 0$; however, the sign of the result will differ in the two cases.

Theorem 10.10.7 (*Roots of trivial odd degree polynomial*)

If n is odd, then the polynomial $ax^n + b$ has a root of the form

$$x = \begin{cases} -\sqrt[n]{\left|\frac{b}{a}\right|} & \text{if } \frac{a}{b} < 0 \\ \sqrt[n]{\left|\frac{b}{a}\right|} & \text{if } \frac{a}{b} > 0 \end{cases}$$

Example 10.10.8 (*Trivial odd degree root*)

$13x^7 + 5$ has root $-\sqrt[7]{\frac{5}{13}}$. Because

$$\begin{aligned} 13x^7 + 5 &= 13\left(-\sqrt[7]{\frac{5}{13}}\right)^7 + 5 \\ &= 13\left(-\frac{5}{13}\right) + 5 \\ &= -5 + 5 = 0 \end{aligned}$$

Listing 10.10.9

```
def poly_mid_zero_roots(coefs, epsilon):
  ...
  return ...
```

10.11 Programming Challenges

Complete the exercises in the file `algebra/roots.py` and test with the tests in the file `tests/roots_test.py`.

Chapter 11

Abstract Algebra

Chapter Objectives

- Define semigroup, monoid, and group in terms of closure, associativity, identity element, and inverses, and give examples and non-examples from familiar sets.
- Implement `isClosed`, `isAssociative`, `isAbelian`, `findIdentity`, and `isMonoid` in Python using `all` / `any` / `next`.
- Explain why the `power` function from Chapter 5 generalizes to any monoid, and apply it to strings, matrices, and integers.
- Describe the inductively constructed (free) monoid over an alphabet and identify concrete examples such as string concatenation.
- Implement the `abstract_algebra` programming assignment.

In Section 10.3, 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.

11.1 Motivation

11.1.1 Video: Evariste Galois a Documentary

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

The video was filmed in a single day in Paris — at Pont Neuf, Lycée Louis-le-Grand, the École Normale Supérieure, and other places from Galois's life that you can walk to today. You are living in one of the most historically and culturally extraordinary cities in the world. Take advantage of it. See the places, learn the history, talk to the people. Galois lived and died within a few kilometres of where you are sitting right now.

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.

Terms to look up before watching the video

Napoleon	Alexander Dumas
Victor Hugo	Joseph Fourier
Monarchist	Republican
Democratic	Pont Neuf
Lycée Louis Le Grand	École Normale Supérieure
Maths	Continued fractions
Apocryphal	Seminal concept

Figure 11.1: Terms to look up before watching *Evariste Galois a documentary* (`https://youtu.be/J6dsanpnpt0`)

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

11.1.2 Questions about the Video

We'll watch the video *Evariste Galois a documentary.* `https://youtu.be/J6dsanpnpt0?si=knrIqqoSmYSVp02Y` Our short tribute to a genius who lived an incredible life. We filmed in Paris for a single crazy day, missed our plane home, then edited into the footage some clips from 'Evariste Galois', a short period-drama about his last amazing night.

The story of Galois is not only a story of genius cut short by tragedy. It is also a story of character and consequence. Some of what happened to him was beyond his control: the indifference of powerful examiners, the political turmoil of post-revolutionary France, the suicide of his father one week before his entrance examination at the École Polytechnique. But some of his difficulties were of his own making. He held his teachers in contempt and made no effort to hide it. He defied the courts. His mathematical writing was so disorganised that even sympathetic readers could not follow it. As you discuss the questions below, pay attention to which adversities were imposed upon him and which he brought upon himself.

1. Which nationality was Evariste Galois?
2. What were Galois's important contributions?
3. Why was Galois misunderstood?
4. Why did he *think* he was begin constantly rejected by the best schools?
5. What are some of the disappointments he faced?
6. Which of the events were beyond his control? And which were his own fault?
7. Why was Galois arrested the first time? Second time?
8. How did he die?
9. What were his last words?
10. Before his death he wrote three letters? What were their contents?

11.1.3 The Central Idea

What was the mathematical idea at the heart of Galois's work? It was the concept of a *group* — a set equipped with an associative operation, an identity element, and inverses for every element. Galois did not announce this as a definition or give it any formal treatment. He mentioned it almost in passing, in a parenthetical remark, buried in work that barely circulated during his lifetime. Yet it is one of the most far-reaching ideas in the history of mathematics.

We will study groups in this chapter. But we approach them by way of a simpler and computationally more immediate structure: the *monoid*. A monoid is a group without the requirement for inverses — and that turns out to be exactly the structure that appears everywhere in computer science, from string concatenation to parallel computation. Once the monoid is understood, the group follows naturally.

11.2 Sets with Structure

In Sections 11.3 and 11.4 and again in Chapter 12, we will study some 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 many mathematical structures defined by axioms. We will thereafter write programs to manipulate these structures.

Before looking at monoids in Section 11.3, we first look at a simpler example.

Example 11.2.1 (*Commutative but not associative*)

Let $M = \{\text{rock}, \text{paper}, \text{scissors}\}$ with the multiplication table:

$\times$	rock	paper	scissors
rock	rock	paper	rock
paper	paper	paper	scissors
scissors	rock	scissors	scissors

First, notice that the multiplication defined in Example 11.2.1 exhibits the *closure* property, meaning that whenever we have $a, b \in M$, then we have $a \times b \in M$.

Since the operation may not be referred to as *multiplication*, we often use the term *Cayley table* to refer to the look up table for the monoidal operation.

You will implement a function `isClosed` which detects whether a given set is closed under the given operation.

Listing 11.2.2 (*Python `isClosed` function*)

```
def isClosed(q, op):
    ...
    return ???
```

The multiplication defined in Example 11.2.1 is commutative, which means that $a \times b = b \times a$ for all $a, b \in M$. This can be seen from the multiplication table which is symmetric about the top-left to bottom-right diagonal.

You will implement a function `isAbelian` which detects whether a given operation is commutative on a given set.

Listing 11.2.3 (*Python `isAbelian` function*)

```
def isAbelian(q, op):
    ...
    return ???
```

Even though the multiplication is commutative, it is not associative. For example

$$(\text{rock} \times \text{paper}) \times \text{scissors} = \text{scissors}$$
$$\text{rock} \times (\text{paper} \times \text{scissors}) = \text{rock}$$

Thanks to `https://math.stackexchange.com/users/122131/ennar` for this stackexchange example `https://math.stackexchange.com/q/1549811`.

You will write a function, `isAssociative`, which determines whether a given operator is associative on a given set.

Listing 11.2.4 (*Python* `isAssociative` *function*)

```
def isAssociative(q, op):
    ...
    return ???
```

The set M (from Example 11.2.1) has no identity element with respect to the operation. *I.e.*, there is no element e such that rock$\times e = e \times$rock = rock (similarly for paper and scissors). We can, however, arbitrarily add such an element, and update the Cayley table.

×	rock	paper	scissors	e
rock	rock	paper	rock	rock
paper	paper	paper	scissors	paper
scissors	rock	scissors	scissors	scissors
e	rock	paper	scissors	e

You will implement a function `findIdentity` which searches a given set for an element which behaves like the identity, *i.e.*, $e \circ x = x \circ e = x$ for all x in the set.

Listing 11.2.5 (*Python* `findIdentity` *function*)

```
def findIdentity(q, op):
    ...
    return ???
```

Can an operator have two identity elements? What happens in the Cayley table if we try to add a second identity, say e'?

×	rock	paper	scissors	e	e'
rock	rock	paper	rock	rock	rock
paper	paper	paper	scissors	paper	paper
scissors	rock	scissors	scissors	scissors	scissors
e	rock	paper	scissors	e	???
e'	rock	paper	scissors	???	e'

What should the values of $e \times e'$ and $e' \times e$ be? Since e is an identity, then $e \times e'$ should be e'. However, since e' is also an identity, then $e \times e'$ should be e. This is impossible unless $e = e'$. From this simple argument, we see that an operation cannot have two distinct identities. We revisit this question in Theorem 11.3.3.

11.3 Monoids

We begin our exploration of sets with structure with the study of the so-called monoid. Monoids play an important role in Computer Science and in computer programming, because they capture some simple axioms which occur often in day to day programming.

Definition 11.3.1 (*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 $a \in S \implies a \circ e = e \circ a = a$

You will implement the `isMonoid` function which determines whether the given set is a monoid under the given operation and with the given (optionally) identity element.

Listing 11.3.2 (*Python `isMonoid` function*)

```
def isMonoid(q, op, e):
    ...
    return ???
```

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 11.3.3 (*Unique Identity*)

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

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

Note that we do not assume that the monoidal operation is commutative. *I.e.*, it is not necessarily the case that $a \circ b = b \circ a$. One might wonder, however, whether commutativity implies associativity. *I.e.*, if we know the operation is commutative, do we still need to prove it is associative? The answer is no, commutative does not imply associativity. We refer the reader to Example 11.2.1

11.3.1 Monoid of Integers

You have already seen several monoids in your mathematical life. The first monoid which might come to mind is the set of natural numbers $\mathbb{Z}$.

Theorem 11.3.4 (*The Multiplicative Integer Monoid*)

The set of integers is a monoid under multiplication.

Proof. To show $\mathbb{Z}$ is a monoid, we must show (1) closure, (2) associativity, and (3) identity.

1. **Closure:** The set of integers is known to be closed under multiplication.
2. **Associativity:** multiplication of integers is known to be associative.
3. **Identity:** 1 is known to be the identity for multiplication within the integers.

Theorem 11.3.5 (*The Additive Inteber Monoid*)

The set of integers is a monoid under addition.

Proof. The proof is left as an exercise for the student. 🎓

Is the set of integers a monoid under the operation of subtraction? No. $\mathbb{Z}$ is certainly closed under subtraction, $a, b \in \mathbb{Z} \implies a - b \in \mathbb{Z}$.

However, the subtraction operator is not associative over the integers;

$$\exists a, b, c \in \mathbb{Z} \text{ for which } (a - b) - c \neq a - (b - c) .$$

Additionally, $\mathbb{Z}$ fails to have an identity element for subtraction. While $a - 0 = a$ for all $a \in \mathbb{Z}$ there is no element, e, for which $e - a = a$.

11.3.2 Clock Arithmetic

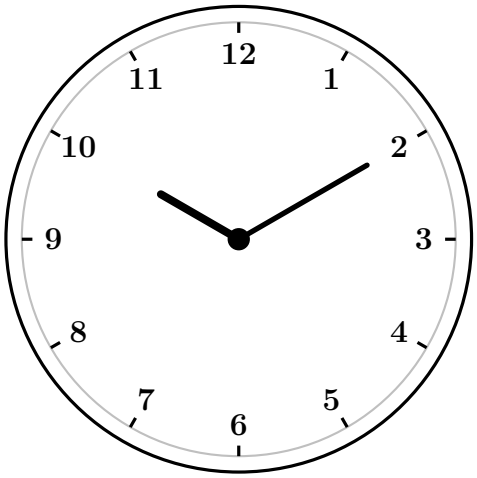


Figure 11.2: A clock face: the 12 hour labels form a group under addition modulo 12

The integers on the face of the clock form a commutative 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 monoidal operation, $(+)$, represent the forward movement of the hands of the clock. For example if the current time is 7 o'clock, $7+6=1$ computes the time 6 hours later. Notice that $6+7=1$ as well because if it is currently 6 o'clock, then 7 hours later it will be 1 o'clock. So clock addition is commutative.

We could verify that clock addition is associative. To do so we'd have to verify $a+(b+c)=(a+b)+c$ for all 12 choices of a, b, and c; thus we'd need to verify $12^3=1728$ equivalences. Such a task should motivate the student to write a program to test associativity. This is the `isAssociative` function in the homework, Section 11.5.

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

We may also define multiplication on the clock face.

The integers on the face of the clock also form a commutative monoid under multiplication with the following Cayley table. The meaning of $a \times b$ is the following. Start at 12 o'clock, and move forward a hours b times. For example, $7 \times 3 = 9$ means that if we start at 12 o'clock and move forward 7 hours, and then move forward 7 hours twice more, the hour hand will be at 9 o'clock.

*	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

Again we might verify associativity of multiplication by verifying all $12^3=1728$ possible equalities of the form $a \times (b \times c) = (a \times b) \times c$. Again, such verification would be easier programmatically than by hand.

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

11.3.3 Semigroups

An algebraic structure slightly simpler than a monoid is a *semigroup*. The term *semigroup* comes from the study of *group* which we will discuss in Section 11.4. As the name suggests, mathematicians have historically been more interested in the study of groups than the study of monoids. Computer scientists are often interested in monoids, because of their applications in exponentiation (Section 11.3.6), finite languages, concurrent and parallel processing (Section 11.3.11), optimzation of computation, and many other applications.

Definition 11.3.6 (*semigroup*)

A *semigroup* is a non-empty set which is closed under an associative operator. More explicitly a set S is a semigroup with operation, $\circ$ if the following hold.

1. Closure: $a, b \in S \implies a \circ b \in S$.
2. Associative: $a, b, c \in S \implies (a \circ b) \circ c = a \circ (b \circ c)$

From Definition 11.3.6, we see that an alternative definition of monoid is a semigroup with identity. Moreover, any semigroup without identity can be extended to a monoid, simply by adding a distinguishable element, e, and extending the Cayley table so that $e \circ x = x \circ e = x$.

Theorem 11.3.7 expresses a subtle but powerful property of semigroups. We will use Theorem 11.3.7 to prove Theorem 11.3.8.

Theorem 11.3.7 (*Associativity of Sub-Semigroups*)

If S is a semigroup with operation, $\circ$, and $T \subset S$, then T is a semigroup if and only if T is closed under the semigroup operation of S.

Proof. Suppose S is a semigroup and $T \subset S$. Suppose $T \subset S$, and suppose T is closed under the operation $\circ$.

To prove T is a monoid, we need only show, associativity, the second axiom of semigroups (Definition 11.3.6), as the first axiom, closure, holds by assumption.

Let $a, b, c \in T$. Since $T \subset S$ then we have $a, b, c \in S$. Since the operation is associative on S we know that

$$(a \circ b) \circ c = a \circ (b \circ c) .$$

Hence, we have $\circ$ associative on T.

Theorem 11.3.8 (*Sub-monoids*)

If M is a monoid with operation, $\circ$, and $T \subset M$, then T is a monoid with the same operation if and only if T is closed under the operation and T contains an identity element. Furthermore the identity element of T is necessarily the same as that of M.

Proof. To show T is a monoid, we must show (1) closure, (2) associativity, and (3) identity.

1. **Closure:** true by assumption
2. **Associativity:** by Theorem 11.3.7, T is a semigroup, and thus the operation is associativity on T.
3. **Identity:** true by assumption.

We now show the identity element of T is the same as that of M. Let e_T be the identity element of T (guaranteed by the hypothesis of the corollary), and let e_M be the identity of M. Let $a \in T$. Since $T \subset M$, then we have $a \in M$ and thus

$$a \circ e_M = e_M \circ a = a .$$

So e_M is an identity also on T. By Theorem 11.3.3, $e_M = e_T$.

Theorem 11.3.8 has powerful consequences. Recall in Section 11.3.2 we discussed that to verify the associativity of an operator in a monoid of n elements, we might have to verify n^3 equalities. An approach with is often easier to show associativity of an operator for a proposed monoid, is to show the set in question is a subset of a larger set which is known to be a monoid (or at least a semigroup). Thereafter, it need only be

shown that the operation is closed on the subset in question, and show that the identity is in the subset. Since Theorem 11.3.8, is *if-and-only-if* we can sometimes also disprove a claim that some set, S is a monoid, for example by showing that another set, M is a monoid, but the identity element of M fails to be an element of S. We use this *trick* to prove Corollary 11.3.9 using Theorem 11.3.4

Corollary 11.3.9

The set of whole numbers ($\mathbb{N} = \{0, 1, 2, \ldots\}$) is a monoid under multiplication.

Proof. By Theorem 11.3.8, we need only show $\mathbb{N}$ has a multiplicative identity and $\mathbb{N}$ is closed under multiplication. Associative comes for free.

Let $a, b \in \mathbb{N}$, then $a \times b \in \mathbb{N}$ so $\mathbb{N}$ is closed. Furthermore, 1 is the identity of multiplication for $\mathbb{N}$.

Corollary 11.3.10

The set of even whole numbers, $\mathbb{N}_{even}$ is not monoid under multiplication.

Proof. Since $\mathbb{N}_{even} \subset \mathbb{N} \subset \mathbb{Z}$ and $\mathbb{Z}$ is a monoid by Theorem 11.3.4. However, the identity, $1 \in \mathbb{Z}$ is not an element of $\mathbb{N}_{even}$ as 1 is not even. Therefore by Theorem 11.3.8, $\mathbb{N}_{even}$ is not a monoid under multiplication.

11.3.4 Proving Closure

When attempting to verify that a given set with a given operation forms a monoid, it may be challenging to show that the operation is closed. Consider the following example

Example 11.3.11 (*Non-zero complex numbers composed of integers*)

Consider the set C defined as follows:

$$C = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid (a, b) \neq (0, 0)\}.$$

Now consider the operation on C defines as

$$(a,b)\circ(c,d) = (ac-bd, bc+ad)\,.$$

Verify that C forms a monoid under this operation

Proof. Showing that $(1,0)$ is the identiy is easy.

$$\begin{aligned}(a,b)\circ(1,0) &= ((a)(1)-(b)(0),(b)(1)+(a)(0))\\ &= (a-0,b-0)\\ &= (a,b)\\ (1,0)\circ(a,b) &= ((1)(a)-(0)(b),(1)(b)+(0)(a))\\ &= (a-0,b-0)\\ &= (a,b)\end{aligned}$$

Showing the operation is associative is straightfoward but tedious.

$$\begin{aligned}(a,b)\circ((c,d)\circ(e,f)) &= (a,b)\circ(ce-df,cf+de)\\ &= ((a(ce-df)-b(cf+de)),a(cf+de)+b(ce-df))\\ &= (ace-adf-bcf-bde,acf+ade+bce-bdf)\\ &= (ace-bde-bcf-adf,acf-bdf+bce+ade)\\ &= ((ac-bd)e-(bc+ad)f,(ac-bd)f+(bc+ad)e)\\ &= (ac-bd,bc+ad)\circ(e,f)\\ &= ((a,b)\circ(c,d))\circ(e,f)\end{aligned}$$

Showing closure is tricky. Clearly the product $(ac-bd,bc+ad)\in\mathbb{Z}^2$. But we need to also show that the $(ac-bd,bc+ad)\neq(0,0)$ assuming that neither (a,b) nor (c,d) is $(0,0)$. To do this we remark that

$$(a,b)\neq(0,0) \iff a^2+b^2>0\,.$$

So it suffices to show that $(ac-bd)^2+(ad+bc)^2>0$.

$$\begin{aligned}(ac-bd)^2+(ad+bc)^2 &= (a^2c^2-2abcd+b^2d^2)+(a^2d^2+2abcd+b^2c^2)\\ &= a^2c^2+b^2d^2+a^2d^2+b^2c^2\\ &= a^2(c^2+d^2)+b^2(c^2+d^2)\\ &= \underbrace{(a^2+b^2)}_{>0}\underbrace{(c^2+d^2)}_{>0}\\ &> 0\end{aligned}$$

11.3.5 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 integers is a monoid under various definitions of various choices of operation. What are some such operations? Consider the following choices of operation:

   - $a \circ b = a + b$
   - $a \circ b = a + b + 1$
   - $a \circ b = a \times b$
   - $a \circ b = |a - b|$

   What is the identity element in each case?

4. The set of even integers is a monoid under addition, but not multiplication. Why?

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 polynomials in a single variable a monoid. What is the operation(s)? What is the identity element for each such operation?

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

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

10. The set, M, in Example 11.2.1 is not a monoid. Why?

11. The set $M = \{1, T, F\}$ using the Cayley table:

$\times$	1	T	F
1	1	T	F
T	T	F	T
F	F	T	T

    M is not a monoid. Why? Is the operation associative? Is there an identity element? Is the operation commutative?

12. The set $M = \{T, F\}$ using the Cayley table:

$\times$	T	F
T	F	T
F	T	T

    M is not a monoid. Why? Is the operation associative? Is there an identity element? Is the operation commutative?

13. Let M be any set of all possible Python lists. M is a monoid with list concatenation as operation. What is its identity element? Is it a commutative monoid?

14. Let M be any set of all possible Python strings. M is a monoid with string concatenation as operation. What is its identity element? Is it a commutative monoid?

11.3.6 Fast Exponentiation in a Monoid

Recall the definition of exponentiation in Equation (4.4). 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.

$$x^n = \begin{cases} e & ;\text{if } n = 0 \\ x & ;\text{if } n = 1 \\ x \circ x^{n-1} & ;\text{if } n \text{ is odd} \\ (x \circ x)^{\frac{n}{2}} & ;\text{if } n \text{ is even} \end{cases} \tag{11.1}$$

The student should recall the Python function `power` defined and discussed in Section 5.3. The `power` function was defined with the two arguments `mult` and `ident`, which are clearly the monoid operation and identity. In fact the `power` function will work not only for integers and strings as was seen in Section 5.3, but it will also work for any monoid which can be represented in Python, provided the elements can be represented as Python objects, and the operation can be represented as a binary Python function.

11.3.7 Inductive Construction

An *inductive construction* is a procedure for constructing a set from a set of *generators* and a binary operation, $\circ$, (or a set of binary operations) via an infinite sequence of intermediate sets.

Definition 11.3.12 (*general inductive construction*)

The *general inductive construction* is the following: Assume $\{o_1, o_2, \ldots o_n\}$ are binary functions ($o_i : U \times U \to U$ for $i = 1, 2, \ldots, n$), and $S_0 \subseteq U$ is a finite set of generators. Next, define S_1, S_2, ... as follows: for each $k = 1, 2, \ldots$, define

$$S_k = S_{k-1} \cup \left(\bigcup_{i=0}^{n} \{o_i(x, y) \mid x, y \in S_{k-1} \right)$$

Finally define $M = \cup_{k=1}^{\infty} S_k$.

The expression $M = \cup_{k=0}^{\infty} S_k$ is understood to mean

$$M = \{s \mid \exists k = 1, 2, \ldots; s \in S_k\}\,.$$

An example of the *general inductive construction* is given in Section 11.3.9. Definition 11.3.13 is a special case of Definition 11.3.12 for the case where $n = 1$; *i.e.*, in the case of a single binary operation.

Definition 11.3.13 (*simple inductive construction*)

The *simple inductive construction* is the following: Assume $S_0 \subseteq U$ is a finite set of so-called *generators*, and $\circ$ is a binary operator defined on $U \times U \to U$. Define

$$M = \bigcup_{k=0}^{\infty} S_k\,,$$

where

$$S_k = S_{k-1} \cup \left(\bigcup_{i=0}^{n} \{x \circ y \mid x, y \in S_{k-1}} \right).$$

Example 11.3.14 (*Inductive construction from even numbers*)

Let $S = \{0, 2\}$. Now we can construct a sequence of sets as follows.

$$\begin{aligned}
S_0 &= \{0, 2\} \\
S_1 &= S_0 \cup \{x + y \mid x, y \in S_0\} \\
&= S_0 \cup \{0 + 0, 0 + 2, 2 + 2\} \\
&= \{0, 2, 4\} \\
S_2 &= S_1 \cup \{x + y \mid x, y \in S_1\} \\
&= S_1 \cup \{0 + 0, 0 + 2, 0 + 4, 2 + 2, 2 + 4, 4 + 4\} \\
&= S_1 \cup \{0, 2, 4, 6, 8\} \\
&= \{0, 2, 4, 6, 8\} \\
S_3 &= S_2 \cup \{x + y \mid x, y \in S_2\} \\
&= \{0, 2, 4, 6, 8, 10, 12, 14, 16\} \\
&\ldots \\
S_k &= S_{k-1} \cup \{x + y \mid x, y \in S_{k-1}\}
\end{aligned}$$

Now, define $M = \cup_{k=1}^{\infty} S_k$. M is the set of positive even integers. Why? Because if you take any even integer, $2m$, there is some k for which $2m \in S_k$.

Sometimes an inductive construction defines a monoid. We claim that M from Example 11.3.14 is a monoid.

Proposition 11.3.15 (*Monoid by Construction*)

M in Example 11.3.14 is a monoid.

Proof. To show that M is a monoid, we must show:

1. **Closure:** To see that M is closed under addition: take $x, y \in M$. There exists $k \in \mathbb{N}$ for which $x, y \in S_k$, and thus $x + y \in S_{k+1} \subset M$.
2. **Associativity:** Clearly the addition operator is associative on whole numbers.
3. **Identity:** Finally, $0 \in S_0 \subset M$, and 0 is the identity.

Note that if the operation is associative, the simple inductive construction always defines a semigroup (recall Definition 11.3.6), but does not necessarily define a monoid. Take for example $S_0 = \{1\}$ as the set of generators, and use addition as the operator. The constructed set will be $M = \{1, 2, 3, \ldots\}$, but $0 \notin M$.

Definition 11.3.16 (*inductively constructed semigroup*)

A semigroup defined by the inductive construction is called an *inductively constructed semigroup*.

For example: a set inductively constructed using the operation of cross-product defined in Exercise 7 (page 247)

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

does not form a semigroup, except in some generate cases such as the generator set $S = \emptyset$, or $S = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\}$, because the cross-product is not an

associative operation.

Definition 11.3.17 (*inductively constructed monoid*)

A monoid defined by the inductive construction is called an *inductively constructed monoid*.

Example 11.3.18 (*Inductively constructed monoid of strings*)

Let S be the following set of Python strings: $S_0 = \{"", "a", "bc"\}$.

Using the operation of string concatenation, what is, M, the inductively constructed semigroup generated by S_0? Is this semigroup a monoid?

The set contains the following trivial elements, both because they are elements of S_0, and because they are results of addition of elements of S_0:

- `""+"" = ""`
- `"a"+"" = "a"`
- `""+"bc" = ""`
- `"a"+"" = "a"`
- `"bc"+"" = "bc"`

The set also contains elements not in S_0

- `"a"+"bc" = "abc"`
- `"bc"+"a" = "bca"`

Furthermore, since we have discovered that the M contains `"abc"` and `"bca"`, then M also contains all binary combinations of `"abc"` and `"bca"` with any of `""`, `"a"`, `"bc"`, `"abc"` or `"bca"`.

- `"abc"+"a" = "abca"`
- `"a"+"abc" = "aabc"`
- `"abc"+"bc" = "abcbc"`

- `"abc"+"abc" = "abcabc"`
- `"bca"+"bca" = "bcabca"`
- ...

In fact, each time we *discover* a new element of the set, x, we know that the set also contains $x + y$ and $y + x$ for all previously discovered values of y.

We'll discuss more about sets related to Example 11.3.18 in Section 11.3.9.

We cannot write a Python program to produce the monoid from Example 11.3.18 as a set, because the set would need to be infinite. However, we can write a program which produces the sequence of sets S_0, S_1, S_2, ..., as in Example 11.3.14.

Listing 11.3.19 (`next_set` *version 1*)

```
1  def next_set(s):
2    return {x + y
3            for x in s
4            for y in s}.union(s)
5
6  def nth_set(n,s0):
7    if n == 0:
8      return s0
9    else:
10     return next_set(nth_set(n-1, s0))
```

Listing 11.3.19 follows the Recursion Recipe (Pattern 4.2.1): termination at `n == 0` returning `s0`; reduction by passing `n-1` to the recursive call; and bookkeeping by applying `next_set` to the result.

The Python code in Listing 11.3.19 is able to generate any particular set in the sequence of inductively generated sets. Python also supports generating the infinite sequence of such sets, as long as at run-time, the only accesses finitely many elements of this infinite sequence. An example is shown in Listing 11.3.20. As is seen in the Listing, `next(...)` is used to get the next element generated. The function `infinite_sequence` uses `yield` rather than `return` to *produce* the next element of the sequence.

Listing 11.3.20 (*Generator of infinite sequence*)

```
def next_set(s):
    return {x + y
            for x in s
            for y in s}.union(s)

def infinite_sequence(s0):
    s = s0
    while True:
        yield s
        s = next_set(s)

ss = infinite_sequence({1,5, -3})
print([next(ss) for _ in range(3)])

# prints the following output
[{1, 5, -3},
 {1, 2, -6, 5, 6, 10, -3, -2},
 {0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 15, 16,
   20, -12, -1, -9, -8, -6, -5, -4, -3, -2}
]
```

11.3.8 Finite Inductive Construction

Recall in Examples 11.3.14 and 11.3.18 we defined a monoid with infinite size. One might think that all inductively constructed sets have infinite size, but that is not the case as shown in Example 11.3.21.

Example 11.3.21 (*An inductively constructed semigroup of sets*)

Let S_0 be the following set of sets: $S_0 = \{\emptyset, \{1,2\}, \{1,3\}\}$. If we now apply the inductive construction using set intersection as monoidal operation.

$$
\begin{aligned}
S_0 &= \{\emptyset, \{1,2\}, \{1,3\}\} \\
S_1 &= S_0 \cup \{x \cap y \mid x, y \in S_0\} \\
&= S_0 \cup \{\emptyset\cap\emptyset, \emptyset\cap\{1,2\}, \{1,2\}\cap\{1,2\}, \{1,2\}\cap\{1,3\}, \{1,3\}\cap\{1,3\}\} \\
&= \{\emptyset, \{1\}, \{1,2\}, \{1,3\}\} \\
S_2 &= S_1 \cup \{x \cap y \mid x, y \in S_1\} \\
&= \{\emptyset, \{1\}, \{1,2\}, \{1,3\}\} \\
&= S_1
\end{aligned}
$$

The process terminates, because $S_2 = S_1$, $S_3 = S_2$, ..., $S_{k+1} = S_k$.

Note that the set constructed in Example 11.3.21 is a semigroup but not a monoid as the set has no identity element as seen in its Cayley table.

$\cap$	$\emptyset$	{1}	{1,2}	{1,3}
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
{1}	$\emptyset$	{1}	{1}	{1}
{1,2}	$\emptyset$	{1}	{1,2}	{1}
{1,3}	$\emptyset$	{1}	{1}	{1,3}

If we perform a similar inductive construction using set union $\cup$ instead of intersection, we will have a monoid.

Example 11.3.22 (*An inductively constructed monoid of sets*)

Let S_0 be the following set of sets: $S_0 = \{\emptyset, \{1,2\}, \{1,3\}\}$. If we now apply the inductive construction using set union as monoidal operation as can be observed from the Cayley table shown below.

$$
\begin{aligned}
S_0 &= \{\emptyset, \{1,2\}, \{1,3\}\} \\
S_1 &= S_0 \cup \{x \cup y \mid x, y \in S_0\} \\
&= S_0 \cup \{\emptyset\cup\emptyset, \emptyset\cup\{1,2\}, \{1,2\}\cup\{1,2\}, \{1,2\}\cup\{1,3\}, \{1,3\}\cup\{1,3\}\} \\
&= \{\emptyset, \{1,2\}, \{1,2,3\}, \{1,3\}\} \\
S_2 &= S_1 \cup \{x \cup y \mid x, y \in S_1\} \\
&= \{\emptyset, \{1,2\}, \{1,2,3\}, \{1,3\}\} \\
&= S_1
\end{aligned}
$$

The process terminates, because $S_2 = S_1$, $S_3 = S_2$, ..., $S_{k+1} = S_k$.

$\cup$	$\emptyset$	{1}	{1,2}	{1,3}	{1,2,3}
$\emptyset$	$\emptyset$	{1}	{1,2}	{1,3}	{1,2,3}
{1}	{1}	{1}	{1,2}	{1,3}	{1,2,3}
{1,2}	{1,2}	{1,2}	{1,2}	{1,2,3}	{1,2,3}
{1,3}	{1,3}	{1,3}	{1,2,3}	{1,3}	{1,2,3}
{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}

The Python code in listing 11.3.19 has the monoidal operation (+)

hard coded. Because of operator overloading in Python, the same code would work for integer addition. However, if the monoidal operation is integer multiplication or set intersection, the same code would not work. However, with a small change, we can generalize the code to accept the monoidal operation as an argument.

Listing 11.3.23 (`next_set` *version 2*)

```
  def next_set(op,s):
    return {op(x, y)
            for x in s
            for y in s}.union(s)

  def nth_set(n,op,s0):
    if n == 0:
      return s0
    else:
      return next_set(op,
                      nth_set(n-1, op, s0))
```

If we attempt to use the Python code in Listing 11.3.23 to generate the sets in Example 11.3.21, we will find that it doesn't work, because Python does not support sets of sets. However, if we could use the `lset` implementation in `algebra/lset.py` from Section 2.4, to represent sets of sets. We'd also need to replace the call to `union` with `lset_union` as in Listing 11.3.24.

Listing 11.3.24 (`next_set` *version 3*)

```
from algebra.lset import lset_union, distinct


def next_set(op, s):
    return lset_union(s, distinct([op(x, y)
                                   for x in s
                                   for y in s]))


def nth_set(n, op, s0):
    if n == 0:
        return s0
    else:
        return next_set(op,
                        nth_set(n - 1, op, s0))
```

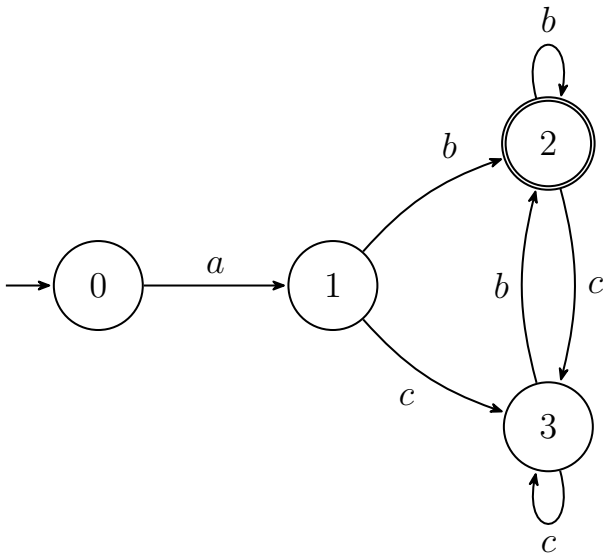


Figure 11.3: Finite Automaton recognizing RE `"a(c*b*)*b"`

11.3.9 Regular Languages

The details of this section are not of critical importance for this course, but the section serves as a non-trivial example of the use of and inductive construction in the field of compute science. Definition 11.3.25 uses the *general inductive construction* explained in Section 11.3.7. We use two two binary operators: language-concatenation and language-union. Additionally, we define the set of generators in two steps rather than one.

Regular expression libraries are common for many programming languages. As an extension of the properties of sets such as discussed in Example 11.3.18, the theory of *regular languages* (also called the theory of *finite automata*) which forms the theoretical basis of regular expressions. A *regular language* is not defined directly; rather the set of all *regular languages* for a given alphabet is defined using an inductive construction. Thereafter, a *regular language* is any element of the set of all regular languages.

An *alphabet*, Σ, is a finite set of distinguishable elements.

A *word* is an ordered finite sequence (possibly empty) of letters from the alphabet. If $\ell_1, \ell_2, \ldots, \ell_n \in \Sigma$, we denote the corresponding word as $[\ell_1, \ell_2, \ldots, \ell_n]$. The empty word is denoted as [].

A word, ℓ_1, of length n may be concatenated to a word, ℓ_2, of length

m to produce a word, $\ell_1 \cdot \ell_2$, of length $n + m$. *E.g.*, if $\{a, b, c, x, y\} \subset \Sigma$, then $[x, y] \cdot [a, b, c] = [x, y, a, b, c]$.

A *language* is a possibly empty set of words.

A language, L_1, may be concatenated to a language L_2 by forming the set of all concatenations of words from L_1 with words from L_2. *I.e.*,

$$L_1 \cdot L_2 = \{\ell_1 \cdot \ell_2 \mid (\ell_1 \in L_1) \wedge (\ell_2 \in L_2)\} \in \mathcal{L}_\Sigma \,.$$

Definition 11.3.25 (*Regular languages*)

The set, $\mathcal{L}_\Sigma$, of *regular languages on* Σ is defined by the following axioms.

1. $\{\emptyset, \{[]\}\} \subset \mathcal{L}_\Sigma$.
2. If $\ell \in \Sigma$, then $\{[\ell]\} \in \mathcal{L}_\Sigma$.
3. If $L_1 \in \mathcal{L}_\Sigma$ and $L_2 \in \mathcal{L}_\Sigma$, then $L_1 \cdot L_2 \in \mathcal{L}_\Sigma$.
4. If $L_1 \in \mathcal{L}_\Sigma$ and $L_2 \in \mathcal{L}_\Sigma$, then $L_1 \cup L_2 \in \mathcal{L}_\Sigma$.

In the theory of regular languages we discover that every *regular expression* corresponds to some uniquely determined regular language. However, each regular language can be expressed by many different regular expressions. Furthermore, we may derive a (not unique) finite automata (as in Figure 11.3) from a regular expression which *accepts* exactly the words from the regular language, and likewise any regular language corresponds to many possible finite automata.

The finite automaton in Figure 11.3 serves as a computation schema to recognize strings which match regular expression `"a(c*b*)*b"`. *E.g.*,

- `"a"`,
- `"ab"`,
- `"abcb"`,
- `"abccccb"`,
- `"abcccbbbcbbbccbb"`,
- etc.

11.3.10 Isomorphisms

Two monoids, M_1 and M_2, are said to be isomorphic if there is a bijection between them. This means there is an invertible function $f : M_1 \to M_2$ such that $f^{-1} : M_2 \to M_1$. In such a case the only thing that distinguishes the two monoids is notation.

Consider the two monoids $M_1 = \{0, 1\}$ and $M_2 = \{T, F\}$ whose Cayley tables are as follows:

M_1

×	1	0
1	1	0
0	0	0

M_2

×	T	F
T	T	F
F	F	F

It is clear that we can construct M_2 from M_1 simply by replacing T with 1 and replacing F with 0. Thus M_1 and M_2 are isomorphic. We may also say that M_1 and M_2 are the same *up to isomorphism*.

It is not always easy to decide whether two monoids are the same (up to isomorphism). For example Is M_2 isomorphic to $M_3 = \{53, 97\}$ with the following multiplication?

M_3

×	97	53
97	97	53
53	53	53

To show M_2 and M_3 are isomorphic we must prove the existence of a bijection between them which also transforms one multiplication table into the other. However, the Cayley tables might not be the same as written. In this example, we have to transform $1 \leftrightarrow 53$ and $0 \leftrightarrow 97$ to obtain the Cayley table M_4:

M_3

×	97	53
97	97	53
53	53	53

$\leftrightarrow$ M_4

×	0	1
0	0	1
1	1	1

$\leftrightarrow$ M_2

×	1	0
1	1	0
0	0	0

The Cayley table M_4 is not exactly like the Cayley table for M_2. To obtain the same table we must swap the rows and columns.

If there are n elements in a monoid, then there are $n!$ different orderings for the rows (and columns). Thus in general to show two Cayley tables are the same, we may need to examine $n!$ different orderings.

Consider the following two Cayley tables. The corresponding monoids are isomorphic. Can you find the isomorphism? Can you find the ordering of the rows and columns that makes the Cayley tables identical?

$\circ$	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

$\circ$	a	b	c	d
a	a	b	c	d
b	b	a	d	c
c	c	d	b	a
d	d	c	a	b

Let's look again at the Cayley table in Example 11.3.22. We repeat the Cayely table here.

M_5

$\circ$	$\emptyset$	{1}	{1,2}	{1,3}	{1,2,3}
$\emptyset$	$\emptyset$	{1}	{1,2}	{1,3}	{1,2,3}
{1}	{1}	{1}	{1,2}	{1,3}	{1,2,3}
{1,2}	{1,2}	{1,2}	{1,2}	{1,2,3}	{1,2,3}
{1,3}	{1,3}	{1,3}	{1,2,3}	{1,3}	{1,2,3}
{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}	{1,2,3}

Now consider a bijection which maps

$$\begin{aligned} \emptyset &\to c \\ \{1\} &\to a \\ \{1,2\} &\to d \\ \{1,3\} &\to b \\ \{1,2,3\} &\to e \end{aligned}$$

This bijection maps M_5 to M_6

$\circ$	c	a	d	b	e
c	c	a	d	b	e
a	a	a	d	b	e
d	d	d	d	e	e
b	b	b	e	b	e
e	e	e	e	e	e

Now sort the Cayley table into alphabetical order.

M_6

∘	a	b	c	d	e
a	a	b	a	d	e
b	b	b	b	e	e
c	a	b	c	d	e
d	d	e	d	d	e
e	e	e	e	e	e

Finally, it is not immediately clear that M_5 and M_6 corresponding to the same monoid, but they do.

11.3.11 Parallelization and Concurrency

There are at least two direct applications of monoids to programming. The first is exponentiation as discussed in Section 11.3.6; if an operation is monoidal, then repeated applications can be optimized in the form of exponentiation.

The second application of a monoid is in parallelization. Let's first look at an example.

Example 11.3.26 (*Summing a list in parallel*)

Suppose you wish to sum the integers in a list. You could of course add them sequentially. For n elements, you'd need $n-1$ additions; thus the time needed would be proportional to the length of the list.[a]

For simplicity let's suppose you wish to sum the list of integers $[0, 1, 2, \ldots, 99]$.

Since $1+2+\cdots+98+99 = (0+\cdots+49)+(50+\cdots+99)$, we can compute $(0+\cdots+49) = 1225$ in one thread and $(50+\cdots+99) = 3725$ in another thread. If our computer supports parallel threads, then those two computations will occur at the same time; thus dividing the wall-time in half.

Once we have the results, we can simply add them to determine the sum from 0 to 99 is $1225 + 3725 = 4950$.

[a] Actually proportional to $n-1$, but we assume $n \approx n+1$ for large lists.

The approach in Example 11.3.26 allows the computation of an associative operation in half the time plus the overhead of the threading mechanism. This overhead is significant for operations as simple as addition of integers, especially short lists. But if the lists are enormous or if

the operation is compute intensive, the threading overhead can become negligible.

The Python code in Listing 11.3.27 performs such summing on a given list, but recursively splitting the list into half (or almost half if the length is odd), until the length becomes 1.

In such an algorithm it is important to never accidentally reverse the arguments of the operation, because in general the operation is not commutative. Line 14, `return op(sums[0], sums[1])` enforces the arguments in the correct order. If the line were changed to `return op(sums[1], sums[0])`, the operations such as integer addition would still work, but operations such as string concatenation or matrix multiplication would compute the wrong result.

The Python code uses feature `async` and `await` which are beyond the scope of this course. The function simply recursively breaks the list in halves and adds the two sums together.

Listing 11.3.27 (*sum_list*)

```
1  import asyncio
2
3  async def sum_list(data, op, identity):
4    if len(data) == 1:
5      print(f"singleton: {data=}")
6      return data[0]
7    elif len(data) == 0:
8      return identity
9    else:
10     print(f"summing: {len(data)} {data=}")
11     mid = len(data) // 2
12     sums = await asyncio.gather(sum_list(data[0:mid],
13                                          op,
14                                          identity),
15                                 sum_list(data[mid:],
16                                          op,
17                                          identity))
18     return op(sums[0], sums[1])
19
20 print(asyncio.run(sum_list(range(100),
21                            lambda a,b: a+b,
22                            0)))
```

If we run this function on `range(12)` rather than `range(100)`, we see the following output—demonstrating that the `range(0,12)` gets re-

cursively bisected until the base case of singletons are reached.

Listing 11.3.28 (*Output of* `sum_list`)

```
    summing: 12 data=range(0, 12)
    summing: 6 data=range(0, 6)
    summing: 6 data=range(6, 12)
    summing: 3 data=range(0, 3)
    summing: 3 data=range(3, 6)
    summing: 3 data=range(6, 9)
    summing: 3 data=range(9, 12)
    singleton: data[0]=0
    summing: 2 data=range(1, 3)
    singleton: data[0]=3
    summing: 2 data=range(4, 6)
    singleton: data[0]=6
    summing: 2 data=range(7, 9)
    singleton: data[0]=9
    summing: 2 data=range(10, 12)
    singleton: data[0]=1
    singleton: data[0]=2
    singleton: data[0]=4
    singleton: data[0]=5
    singleton: data[0]=7
    singleton: data[0]=8
    singleton: data[0]=10
    singleton: data[0]=11
    66

```

This Python code is somewhat dubious, because the `asyncio` package does not actually support parallel execution. However, the same or similar technique in other computer languages do support threads running in parallel on multi-core machines.

Notice that the code has three `if` branches: singleton list, empty list, and other list. The singleton list case takes advantage of the fact that $x \circ e = x$; thus there is no need to explicitly mention the identity element in this case. However we need to handle the empty list case, if we wish to implement a feature such as:

- The sum of an empty list of integers should be 0.
- The product of an empty list of integers should be 1.
- The union of and empty list of sets should be $\emptyset$.

- The minimum of an empty list of whole numbers should be 0.

Notice that the monoidal identity is only explicitly used in the empty list case. If your application can guarantee that the list is never empty, then the empty list can be eliminated and thus the need for specifying the identity element goes away. This allows the parallelization algorithm to work for a semigroup.

11.3.12 Parallel List Sum in Clojure

As mentioned in Section 11.3.11, the Python `asyncio` package does not support parallel threads. Here is a very similar implementation of the algorithm in the Clojure programming language which does support parallel threads. It is not important to understand the details of the Clojure language to get a general idea of the code for the function `sum-list`, which is similar structurally (but not syntactically) to the Python code for `sum_list`

Listing 11.3.29 (*Clojure implementation of* `sum-list`)

```
(defn sum-list [data]
  (if (= (count data) 1)
    (do
      (println [:singleton (first data)])
      (first data))
    (do
      (println [:summing (count data)
                :data data])
      (let [mid (/ (count data) 2)
            [head tail] (split-at mid data)
            s1 (future (sum-list head))
            s2 (future (sum-list tail))]
        (+ @s1 @s2)))))
```

When we run this program we see a very different output compared to the output of the Python program.

Listing 11.3.30 (*Output of* `sum-list`)

```
[:summing 12 :data (0 1 2 3 4 5 6 7 8 9 10 11)]
[:summing 6 :data[ :summing 6 :data ((6 07  18  29  10 3 114)
   ]5)
]
[:summing 3 :data (9 10 11)]
[:summing [3 :data :summing (3 6:data  7 8()[]3
:summing4 3 :data  (0 51) ]2)]

[[[:summing[:singleton:singleton  :singleton 5]8[]
[
:summing 2 [:data:summing  2  (26:data   :data7)( ]
[0( :summing1 )]2:singleton3   :data 2]4
)(]11]9


[:singleton[ 3:singleton] 10
4)][:singleton[ 1]

:singleton]
[ 0[[]:singleton:singleton
106]]:singleton 7]
[:singleton 9]


66
```

Whereas the output from the Python program contains one print output per line, the Clojure output is much messier. We can see that the `println` calls are writing over each other, because they are running in many different parallel threads. Several of the threads are trying to print at the same time.

11.3.13 Challenges with Monoids

1. We have seen examples of 2-element monoids. Give an example of a 3-element monoid.

2. How many 2-element monoids exist, up to isomorphism?

3. Write a function to compute b^p where $p \in \mathbb{N}$ and $1 \leq b < 12$ which works using clock arithmetic. Hint, use Section 11.3.6.

4. Given three distinct elements, one of which is the designated iden-

tity element, count the Cayley tables of the 3-element monoids. Count the Cayley tables of the 3-element commutative monoids.

∘	e	a	b
e	?	?	?
a	?	?	?
b	?	?	?

5. Given two sets, H, and G, and a bijection `bij`, write a Python function, `isIsomorphism`, which determines whether `bij`, is an isomorphism from H to G, *i.e.*, does it transform one Cayley table of H to the Cayley table of G?

6. Write a Python function to determine whether two monoids are isomorphic. Hint, iterate over all possible bijections from H to G stopping if such a function is an isomorphism according to `isIsomorphism`.

7. Let the 3-dimensional *cross-product* is a binary operation on R^3; *i.e.* $x, y \in \mathbb{R}^3 \implies x \times y \in R^3$. The *cross-product* is defined as follows:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

   Is the cross-product commutative? associative? Is R^3 a monoid under the cross-product operation?

8. The set $\mathcal{L}_\Sigma$ as defined in Section 11.3.25 has two operations defined: $L_1 \cdot L_2$ and $L_1 \cup L_2$. $\mathcal{L}_\Sigma$ is a monoid under both of these operations. What are the identites for each?

11.4 Groups

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 $7 \times c = 11$.

Consulting the 12-clock Cayley table on page 223, we see that there is no solution of $a \times 3 = 7$, because 7 is not found in row 3 of the table. Furthermore, we see that $4 \times b = 8$ as multiple solutions: $4 \times 2 = 8$, $4 \times 5 = 8$, $4 \times 8 = 8$, and $4 \times 11 = 8$. Finally, $7 \times c = 11$ has a unique solution, $7 \times 5 = 11$.

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.

You will implement the function `findInverse` which searches a set for an inverse of a given element, using a given operation and given (optional) identity element. The function should return `None` if no inverse exists for the given element `x`.

Listing 11.4.1 (*Python* `findInverse` *function*)

```
def findInverse(x, q, op, e):
    ...
    return ???
```

Definition 11.4.2 (*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 $a \circ a^{-1} = a^{-1} \circ a = e$

You will implement the `isGroup` function which determines whether the given set is a monoid under the given operation and with the given (optionally) identity element.

Listing 11.4.3 (*Python* `isGroup` *function*)

```
def isGroup(q, op, e):
    ...
    return ???
```

To implement `isGroup` you'll need to first implement the `hasInverses` function which calls `findInverse` repeatedly to determine whether all elements of the given set have an inverse.

Listing 11.4.4 (*Python* `hasInverses` *function*)

```
def hasInverses(q, op, e):
    ...
    return ???
```

Definition 11.4.5 (*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 11.3.3. But we can ask a similar question about uniqueness of the inverse. Can an element $a \in S$ have two different inverses? No, but why?

Theorem 11.4.6 (*Unique Inverse*)

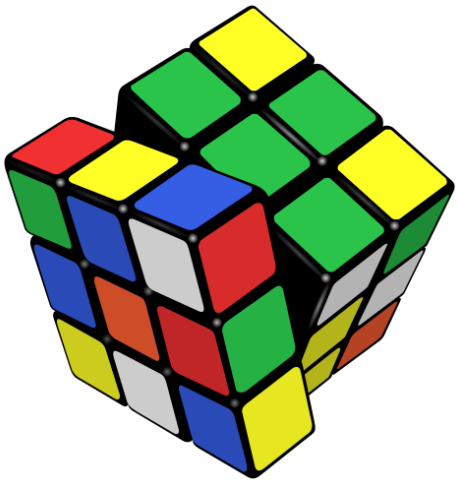
Each element of a group has exactly one inverse.

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

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

11.4.1 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 group 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 polynomials of a single variable is an additive group. What are the inverses and what is the identity? But this set is not a multiplicative group. Why?

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. Consider again the 11-clock from the previous example. We can tediously verify the existence of a unique inverse for each non-11 element.

   The set of non-11 elements of the 11-clock is a multiplicative group with the following Cayley 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 Example 11.3.18, we defined a set of strings with concatenation as the monoid operation. This set is not a group. Why?

11. The set $\mathcal{L}_\Sigma$ as defined in Section 11.3.25 has two operations defined: $L_1 \cdot L_2$ and $L_1 \cup L_2$. $\mathcal{L}_\Sigma$ is a group under neither of these operations. Why?

11.4.2 Fast Exponentiation in a Group

In Section 11.3.6 we talked about fast exponentiation in a monoid. 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 $|n|$.

$$x^n = \begin{cases} e & ;\text{if } n = 0 \\ x & ;\text{if } n = 1 \\ x \circ x^{n-1} & ;\text{if } n > 0 \text{ is odd} \\ (x \circ x)^{\frac{n}{2}} & ;\text{if } n > 0 \text{ is even} \\ (x^{-1})^{|n|} & ;\text{if } n < 0 \end{cases} \tag{11.2}$$

11.4.3 Challenges

1. Under some operation, is the empty set a monoid? semigroup? group?

2. Which of the familiar numerical sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ are groups, and under which operations (addition, subtraction, division, exponentiation).

3. Which subsets of $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ are groups with multiplication as the operation?

4. The set of Python strings under string concatenation is not a group? Why? Of course the string concatenation operation is not closed if memory is finite (why?). But assume memory is infinite, is the set a group under that assumption?

5. What is the subset relation between the set of groups, the set of semigroups, and the set of monoids? *I.e.*, is every monoid a group; is every group a semigroup; is every group a monoid; etc?

6. Which of the following Cayley tables (if any) define groups? If not, why not?

∘	e	a	b
e	e	e	e
a	e	e	e
b	e	e	e

∘	e	a	b
e	e	a	b
a	a	e	e
b	b	e	e

∘	e	a	b
e	e	a	b
a	a	a	b
b	b	a	b

∘	e	a	b
e	e	a	b
a	a	a	b
b	b	b	b

7. How many possible Cayley tables are there if a group has exactly 3 elements: a, b, and e? Remember that every element must have an inverse, and the operation must be associative.

∘	e	a	b
e	?	?	?
a	?	?	?
b	?	?	?

8. The set $\mathbb{Z}$ with operation, $x \circ y = |x - y|$, is not a group. Why not?

9. Can you find a 3-element non-Abelian group? Hint, use the result of the previous exercise.

10. How many 2-element groups are there, up to isomorphism?

11. 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?

12. Determine (experimentally or otherwise) 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?

13. Theorem 11.4.6 shows that every group element has exactly one inverse. However, can a finite group contain two elements, a and b, which are not inverses but nevertheless $a \circ b = e$ while $b \circ a \neq e$?

14. Use a simple inductive construction (Definition 11.3.13) to construct the Cayley table for the set whose generators are $\{R90, MX\}$ where $R90$ is a 90° counter-clockwise rotation about the origin, and MX is a reflection about the x-axis.

    The binary operation is concatenation of transforms, for example $MX \circ MX$ means reflect twice about the x-axis; $MX \circ R90$ means first reflect about the x-axis, then rotate 90° about the origin. Is $MX \circ R90 = R90 \circ MX$?

    This set is finite so the Cayley table has finitely many rows and columns. How many elements does this set have? You are free to invent names for newly discovered elements; *e.g.* $R90 \circ R90 = R180$; use whatever names you like to name all the elements.

$\circ$	MX	$R90$	$\cdots$
MX	?	?	$\cdots$
$R90$	?	?	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

    Is the operation commutative? Is the operation associative? Does the set contain an identity element?

    Is this set a group?

11.5 Programming Challenges

Complete the exercises in the files `algebra/monoid.py` and `algebra/group.py`. Test with the tests in file `tests/monoid_test.py`.

Chapter 12

Rings and Fields

Chapter Objectives

- State the ring axioms and verify that $(S, +, \times)$ is a ring for examples including $\mathbb{Z}$, $n \times n$ matrices, and $\mathbb{Z}[x]$.
- Explain why the polynomial operations `poly_add` and `poly_mult` from Chapter 7 are the $+$ and $\times$ of the ring $\mathbb{Z}[x]$.
- State the field axioms and identify examples ($\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}/p$ for prime p) and non-examples ($\mathbb{Z}$, $\mathbb{Z}/12$).
- Implement modular arithmetic (addition, subtraction, multiplication, division, and exponentiation) for $\mathbb{Z}/n$ using fast exponentiation from Chapter 5.
- Implement the `ring` and `field` programming assignments.

In Chapter 11 we discussed algebraic structures having a single operation. In this chapter, we'll discuss algebraic structures having two operations. The two most important such structures are the ring (Section 12.1) and the field (Section 12.2).

12.1 Rings

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 12.1.1 (*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:
   $a, b, c \in S \implies (b + c) \times a = (b \times a) + (c \times a)$.

Some authors prefer to define a ring such that $(S, \times)$ is a semigroup rather than a monoid. In those treatments a ring may lack a multiplicative identity. Such authors may then to refer to a *ring with unity*. For our treatment, we will consider rings to have both additive and multiplicative unit.

Definition 12.1.2 (*Commutative Ring*)

If the $\times$ operation of a ring is commutative, *i.e.*, if $(S, \times)$ is a commutative monoid, $(S, +, \times)$ is called an *commutative* 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 12.1.3. Several other familiar connections between multiplication and addition also hold for rings, in general. In particular $-1 \cdot -1 = 1$ (Theorem 12.1.4), and $-1 \cdot a = -a$ (Theorem 12.1.5).

Theorem 12.1.3 (*Annihilation*)

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 12.1.4 (*Additive inverse of identity*)

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.

$$\begin{aligned}
\overline{e} \cdot \overline{e} &= \overline{e} \cdot \overline{e} + z \\
&= \overline{e} \cdot \overline{e} + (\overline{e} + e) \\
&= \overline{e} \cdot \overline{e} + (\overline{e} \cdot e + e) \\
&= (\overline{e} \cdot \overline{e} + \overline{e} \cdot e) + e \\
&= \overline{e} \cdot (\overline{e} + e) + e \\
&= \overline{e} z + e \\
&= z + e && \text{By Theorem 12.1.3} \\
&= e
\end{aligned}$$

Theorem 12.1.5 (*Negative elements*)

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.

$$\begin{aligned}
\overline{e} \cdot a &= \overline{e} \cdot a + z \\
&= \overline{e} \cdot a + (a + \overline{a}) \\
&= (\overline{e} \cdot a + a) + \overline{a} \\
&= (\overline{e} \cdot a + e \cdot a) + \overline{a} \\
&= (\overline{e} + e) \cdot a + \overline{a} \\
&= z \cdot a + \overline{a} \\
&= z + \overline{a} && \text{By Theorem 12.1.3} \\
&= \overline{a}
\end{aligned}$$

12.1.1 Ring of Polynomials

Recall in Chapter 7 we talked extensively about polynomials. The set of polynomials in a single variable is a ring, if the coefficients come from

a ring. For example, the set of integers, $\mathbb{Z}$, form a ring, so the set of polynomials with integer coefficients also form a ring. This ring is sometimes denoted $\mathbb{S}[x]$ where S is the ring of coefficients: for example $\mathbb{R}[x]$, $\mathbb{Q}[x]$, $\mathbb{R}[x]$, $\mathbb{C}[x]$, etc. Furthermore, $\mathbb{S}[x]$ is commutative whenever S is commutative.

Let's look a bit more closely at the ring $\mathbb{Z}[x]$, knowing that $\mathbb{Z}$ is a ring. Why is $\mathbb{Z}[x]$ a ring? Let's look at the ring axioms. Notice that the exact same reasoning works for any ring S rather than explicitly $\mathbb{Z}$.

- $(\mathbb{Z}[x], +)$ is an Abelian group.
  - **Closure:** Clearly, the sum of two polynomials is a polynomial. And if the given polynomials have integer coefficients, then the sum certainly has integer coefficients.
  - **Associativity:** Clearly, polynomial addition is associative. Take three polynomials: $p(x), q(x), r(x) \in \mathbb{Z}[x]$. The fact that $(p(x) + q(x)) + r(x) = p(x) + (q(x) + r(x))$ follows from the fact that addition is associative in $\mathbb{Z}$.
  - **Identity:** The zero polynomial, 0, is the group additive identity. For every $p(x) \in \mathbb{Z}[x]$ we have $0+p(x) = p(x)+0 = p(x)$.
  - **Inverse:** For every $p(x) \in \mathbb{Z}[x]$, we have $-p(x) \in \mathbb{Z}[x]$ with $p(x) + -p(x) = 0$.
  - **Commutativity:** for every $p(x), q(x) \in \mathbb{Z}[x]$ we have $p(x) + q(x) = q(x) + p(x)$.
- $(\mathbb{Z}[x], \times)$ is a monoid.
  - **Closure:** Clearly, the product of two polynomials is a polynomial. And if the given polynomials have integer coefficients, then the product certainly has integer coefficients.
  - **Associativity:** See Theorem 12.1.6.
  - **Identity:** The unit polynomial, 1, is the group multiplicative identity. For every $p(x) \in \mathbb{Z}[x]$ we have

$$1 \times p(x) = p(x) \times 1 = p(x)\,.$$

- Left-to-right distribution. See Theorem 12.1.7
- Right-to-left distribution. This follows from left-to-right distribution and the fact that $\mathbb{Z}[x]$ multiplication is commutative.

The Python functions `poly_add` and `poly_mult` from Chapter 7 are precisely the $+$ and $\times$ operations of the ring $\mathbb{Z}[x]$. The ring axioms verified above are the abstract guarantee that the concrete implementations behave correctly under composition.

Theorem 12.1.6 (*Polynomial multiplication is associative*)

$f(x), g(x), h(x) \in \mathbb{Z}[x] \implies (f(x)g(x))h(x) = f(x)(g(x)h(x))$.

This proof was aided by ChatGPT `https://chatgpt.com/share/69aac2b0-5f8c-800d-825e-81a5ee2238d4`.

Proof. Let m is the maximum degree of the three polynomials. Choose: $a_0, \ldots, a_m \in \mathbb{Z}$, $b_0, \ldots, b_m \in \mathbb{Z}$, and $c_0, \ldots, c_m \in \mathbb{Z}$ such that $f(x) = \sum_{i=0}^{m} a_i x^i$, $g(x) = \sum_{j=0}^{m} a_j x^j$, and $h(x) = \sum_{k=0}^{m} a_k x^k$.

Note that

$$f(x)g(x) = \sum_{i=0}^{m}\sum_{j=0}^{m} a_i b_j x^{i+j}$$

$$g(x)h(x) = \sum_{j=0}^{m}\sum_{k=0}^{m} b_j c_k x^{j+k}$$

So that

$$\begin{aligned}(f(x)g(x))h(x) &= \left(\sum_{i=0}^{m}\sum_{j=0}^{m} a_i b_j x^{i+j}\right)\sum_{k=0}^{m} a_k x^k \\ &= \sum_{i=0}^{m}\sum_{j=0}^{m}\sum_{k=0}^{m} (a_i b_j) c_k x^{i+j+k} \\ f(x)(g(x)h(x)) &= \sum_{i=0}^{m} a_i x^i \left(\sum_{j=0}^{m}\sum_{k=0}^{m} b_j c_k x^{j+k}\right) \\ &= \sum_{i=0}^{m}\sum_{j=0}^{m}\sum_{k=0}^{m} a_i (b_j c_k) x^{i+j+k}\end{aligned}$$

We know that for all i, j, k, we have $(a_i b_j)c_k = a_i(b_j c_k)$, because multiplication is associative in $\mathbb{Z}$. Thus

$$(f(x)g(x))h(x) = f(x)(g(x)h(x))\,.$$

Theorem 12.1.7 (*Polynomial left-to-right distributive*)

$f(x), g(x), h(x) \in \mathbb{Z}[x] \implies f(x)(g(x) + h(x)) = f(x)g(x) + f(x)h(x)$.

Proof. As in Theorem 12.1.6, let

$$f(x) = \sum_{i=0}^{m} a_i x^i$$
$$g(x) = \sum_{j=0}^{m} a_j x^j$$
$$h(x) = \sum_{k=0}^{m} a_k x^k$$

We need three intermediate values to facilitate the development below.

$$g(x) + h(x) = \sum_{j=0}^{m} (b_j + c_j)x^j$$
$$f(x)g(x) = \sum_{i=0}^{m} \sum_{j=0}^{m} a_i b_j x^{i+j}$$
$$f(x)h(x) = \sum_{i=0}^{m} \sum_{j=0}^{m} a_i c_j x^{i+j}$$

So now we have

$$\begin{aligned} f(x)(g(x)+h(x)) &= \left(\sum_{i=0}^{m} a_i x^i\right)\left(\sum_{j=0}^{m}(b_j+c_j)x^j\right) \\ &= \sum_{i=0}^{m}\sum_{j=0}^{m} a_i(b_j+c_j)x^{i+j} \\ &= \sum_{i=0}^{m}\sum_{j=0}^{m}(a_i b_j + a_i c_j)x^{i+j} \\ &= \sum_{i=0}^{m}\sum_{j=0}^{m} a_i b_i x^{i+j} + \sum_{i=0}^{m}\sum_{j=0}^{m} a_i c_i x^{i+j} \\ &= f(x)g(x) + f(x)h(x) \end{aligned}$$

12.1.2 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, Section 12.1.1
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

12.2 Fields

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 12.2.1 (*Field*)

$(F, +, \times)$ is called a *field* if

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

Note that in Definition 12.2.1, $(F \setminus \{0\}, \times)$ must be a group. But since $(F, +, \times)$, then whenever $(F \setminus \{0\}, \times)$ is a group then it is necessarily an Abelian group.

Theorem 12.2.2 (*Alternate Definition of Field*)

$(F, +, \times)$ is a field if and only if

1. $(F, +, \times)$ is an Abelian ring,
2. the identity of $+$ is distinct from the identity of $\times$, and
3. all non-zero elements have a multiplicative inverse.

12.2.1 Examples of Fields

1. The rational numbers, $\mathbb{Q}$, is a field.
2. The rational numbers, $\mathbb{R}$, is a field.
3. The complex numbers, $\mathbb{C}$, is a field.
4. The set of polynomials with integer coefficients, $\mathbb{R}(x)$, is not a field. Why?
5. The set of $n \times n$ matrices is not a field. Why?
6. The set of 2×2 matrices of the form $\begin{bmatrix} a & -b \\ b & a \end{bmatrix}$ where $a, b \in \mathbb{Q}$, is a field.
7. The integers modulo 12 is not a field. Why?
8. The integers modulo any prime such as 7 is a field.

+	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

×	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.

9. 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 $a + b\sqrt{2} \neq 0$?

12.2.2 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 $a^b \mod p$? Efficient modular exponentiation is fast exponentiation (Section 5.2) applied in $(\mathbb{Z}/p, \times)$: the same repeated-squaring strategy that computes a^b in $O(\log b)$ multiplications over any ring.

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.

12.3 Programming Challenges

Complete the exercises in the file `algebra/ring.py` and `algebra/field.py`. Test with the tests in file `tests/ring_test.py`.

Appendix A

Course Setup Check-List

You will need to set up a work area, install several tools, and download code and templates. The following is a check-list to assure that each student has a valid setup to proceed in this course.

- Generate Public Keys
  - ☐ Verify `git` and `ssh` B.1.2
  - ☐ Generate public key, Section B.2.
  - ☐ Register public key with Forge, Section B.3.
  - ☐ Register public key with GitLab, Section B.4.
- Setup Repository
  - ☐ Download `checkout-workarea-algebra-1.py` Script, SectionC.1
  - ☐ Checkout grader repository, Section C.1.7.
- Submitting your homework
  - ☐ Open IDE, Section D.1.
  - ☐ Test interactively, Section D.1.
  - ☐ Submit from menu, Section D.2.1.
  - ☐ View results, Section D.3.
  - ☐ (VS Code only) Install **Black Formatter**, Section D.4.2.
  - ☐ Format code, Section D.4.2.

Figure A.1: Check List

Appendix B

Public Keys

B.1 Before Arriving in Class

YOU MUST MAKE SURE YOU HAVE `git` and `ssh-keygen` ALREADY INSTALLED ONTO YOUR LAPTOP BEFORE COMING TO CLASS.

Section B.1.1 suggests how to install these tools. However, you may also do your own web search or ask ChatGPT for installation instructions.

B.1.1 Verify `ssh` and `git`

First, verify that `ssh` and `git` are installed. If they are installed, skip to Section B.2.

To verify, enter the following in the shell or PowerShell.

Listing B.1.1 (*Shell or Power Shell*)

```
cmd> git --version
git version 2.32.0

cmd> ssh -V
OpenSSH_9.9p2, LibreSSL 3.3.6
```

It is not important that you have exactly the most recent versions of these two tools. It is important however, that these two command do not trigger an error message.

If you see an error such as **not recognized**, then proceed to Section B.1.2 and install the required tools. If the two tools are already installed skip to Section B.2.

B.1.2 Installing `ssh` and `git`

Most systems come with these tools already installed. However, if you don't already have `git` and `openssh`, you need to install them on your laptop. You will have to install: `git` and `openssh` (at least the ssh client).

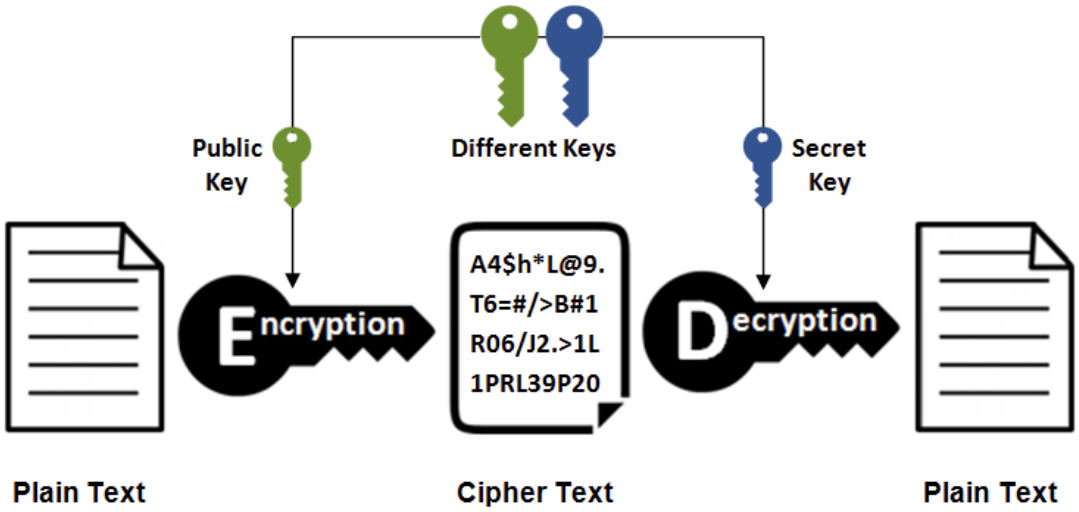
To install `git` on Windows, you can use the `git-for-windows` installer `https://gitforwindows.org/`.

On Linux, it depends of the package manager but `apt-get install git` on debian-based, `pacman -S git` on Arch-based is enough.

On MacOS, typing `git` will trigger the installation if is it not available.

You can find more information here: `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

The SSH-client is it already in embedded in MacOS and Linux. To enable it on Windows, you can followed `https://learn.microsoft.com/fr-fr/windows/terminal/tutorials/ssh`



B.1.3 What is `ssh-keygen`

According to `https://www.techtarget.com`, the `ssh-keygen` command is a component of most SSH implementations used to generate a public key pair for use when authenticating with a remote server. In the typical use case, users generate a new public key and then copy their public key to the server using SSH and their login credentials for the remote server.

By default, `ssh-keygen` creates an RSA key pair and stores the public key in a public key file named `.ssh/id_rsa.pub` and a private key file named `.ssh/id_rsa`.

B.2 Create a Public Key

To use the *intra* service you'll need to create an ssa public key and register that key in two places, once on `https://cri.epita.fr` (Section B.3) and once on `https://gitlab.cri.epita.fr` (Section B.4).

The procedure for creating an ssa public key is different from Windows, Linux, and MacOS.

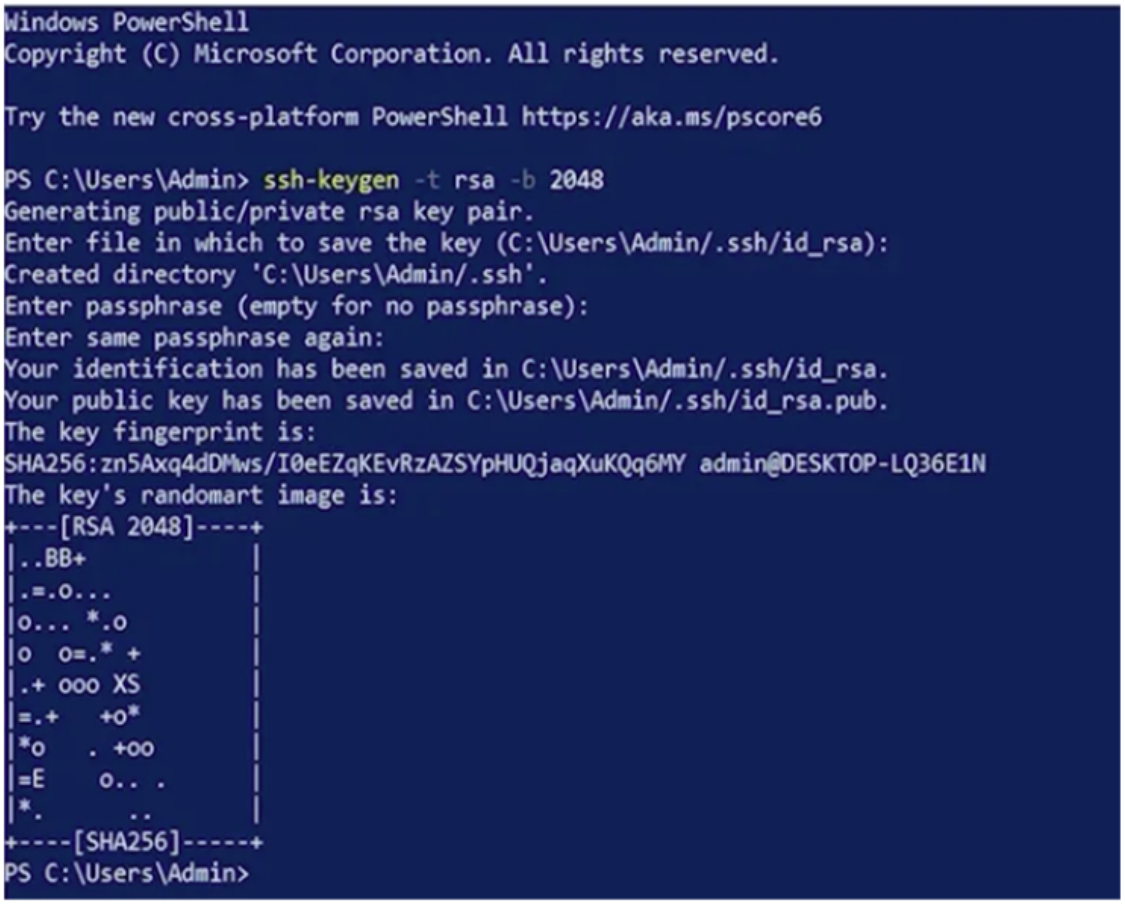
B.2.1 From Windows

If you are using Windows, then you'll need to open the `powershell`.

From the windows search bar, type powershell and click on the top result. A powershell window will open.

Next type `cd .ssh`. Use the command `ls` to discover the files in the `.ssh` directory. If you already have a file named `id_rsa.pub`, then you already have a public key. Otherwise enter the command `ssh-keygen`; you will be prompted for several questions—press ENTER to accept the defaults for each prompt. Don't enter a passphrase; just press ENTER. In the following image, the user has give some command line options to `ssh-keygen`; these are not necessary.

If the command `ssh-keygen` is not found, you may need to install some missing tools. See Section B.1.2.



To see your public key, enter `cat id_rsa.pub`.

```
> cd .ssh
> cat id_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQC4+iMUZvB6VObdDAiOOOaAg16ZMtlJEYDTFite+QSpIF5/5aMcoExJrVLOWa
eAHufDIelwinMHylUAFzdXh+8aIu88aV1dP4RAlTDN+7bwpoE0bQ7cOjURIkUCvuQHD5DY/ToPcrUkAILZCwhAkMjTbiDcvcKa
81OND1Xc7slHeSazy7pwrxiJSGL6CwfdmXjZhZdy01p8gH2+mPha/OTeYaukU/c1vvbuo3GwT1Kx99ruFhY76qbwuRjJlHbb4a
YyCc99v1RIARVUKmAnptAczpcm4zykg9U86PDpUpO8Koh8P7SvzgS9rq3NFWYkgwsF3Kwdn3brwWXvnhN7V5vLmq8bnFWNU7Ka
OvZA43/BbkTbJmol65jTgWjt8UDDJDj7AhFQwXayDG0ksfjfJrbVa/qKsogTvqYgUUpNhd2mWCydzg0MptG4Au2JPxtDS7dLFa
jvOgNVaLSegSXUbtFDqj4gi4/8FuiEN9PqzAazardHqLYAhkThwJnkqCo9H/W4oLeU= john.smith@epita.fr
```

B.2.2 From Linux and MacOS

Open a shell (or `xterm`), and type `cd .ssh ; ls`. If you already have a file named `id_rsa.pub`, then you already have a public key. Otherwise you'll need to create one with the command `ssh-keygen`. Press ENTER to accept the default value for all prompts. Finally, display the content of the file using `cat id_rsa.pub`. You should see something like the following.

```
> cd .ssh
> cat id_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQC4+iMUZvB6VObdDAiOOOaAg16ZMtlJEYDTFite+QSpIF5/5aMcoExJrVLOWa
eAHufDIelwinMHylUAFzdXh+8aIu88aV1dP4RAlTDN+7bwpoE0bQ7cOjURIkUCvuQHD5DY/ToPcrUkAILZCwhAkMjTbiDcvcKa
810ND1Xc7slHeSazy7pwrxiJSGL6CwfdmXjZhZdy01p8gH2+mPha/OTeYaukU/c1vvbuo3GwT1Kx99ruFhY76qbwuRjJlHbb4a
YyCc99v1RIARVUKmAnptAczpcm4zykg9U86PDpUpO8Koh8P7SvzgS9rq3NFWYkgwsF3Kwdn3brwWXvnhN7V5vLmq8bnFWNU7Ka
OvZA43/BbkTbJmol65jTgWjt8UDDJDj7AhFQwXayDG0ksfjfJrbVa/qKsogTvqYgUUpNhd2mWCydzg0MptG4Au2JPxtDS7dLFa
jvOgNVaLSegSXUbtFDqj4gi4/8FuiEN9PqzAazardHqLYAhkThwJnkqCo9H/W4oLeU= john.smith@epita.fr
```

B.2.3 Copy Public Key to Mouse Clipboard

Once the public key has been generated you'll need to copy the key to the mouse clipboard so you can paste it in Section B.3 and in Section B.4.

You may simply use `cat` to output the content of the file then select and copy with the mouse.

Listing B.2.1 (*Output Key and Copy with Mouse*)

```
cmd> cat ~/.ssh/id_rsa.pub
```

MacOS users may select the text with the mouse and press **CMD-C**. Linux and Windows users may select with CTL-C.

You may also automate the copy-to-clipboard operation, but it is done differently on each operating system.

Listing B.2.2 (*Windows*)

```
cmd> cat ~/.ssh/id_rsa.pub | clip
```

Listing B.2.3 (*Macos*)

```
cmd> cat ~/.ssh/id_rsa.pub | tr -d '\n' | pbcopy
```

Listing B.2.4 (*Linux*)

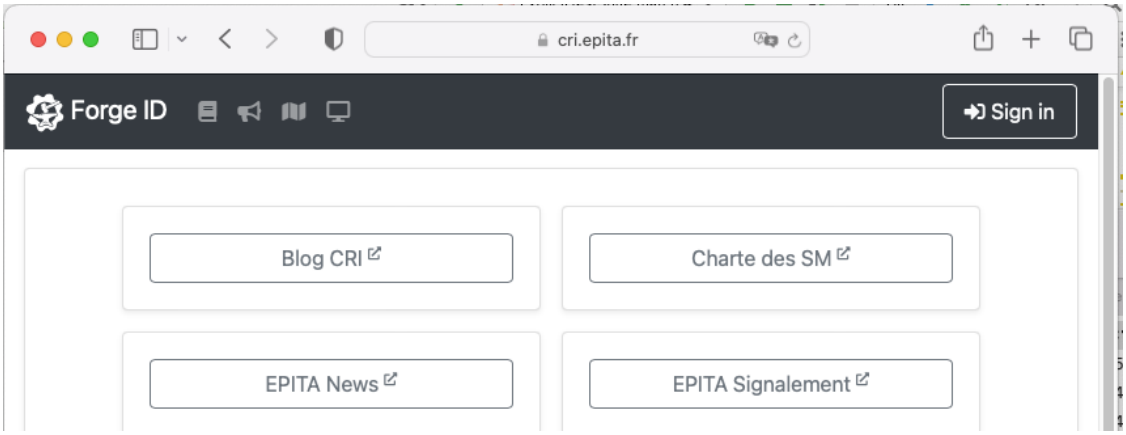
```
cmd> cat ~/.ssh/id_rsa.pub | tr -d '\n' | xclip -selection
    clipboard
```

B.3 Register Public Key with Forge

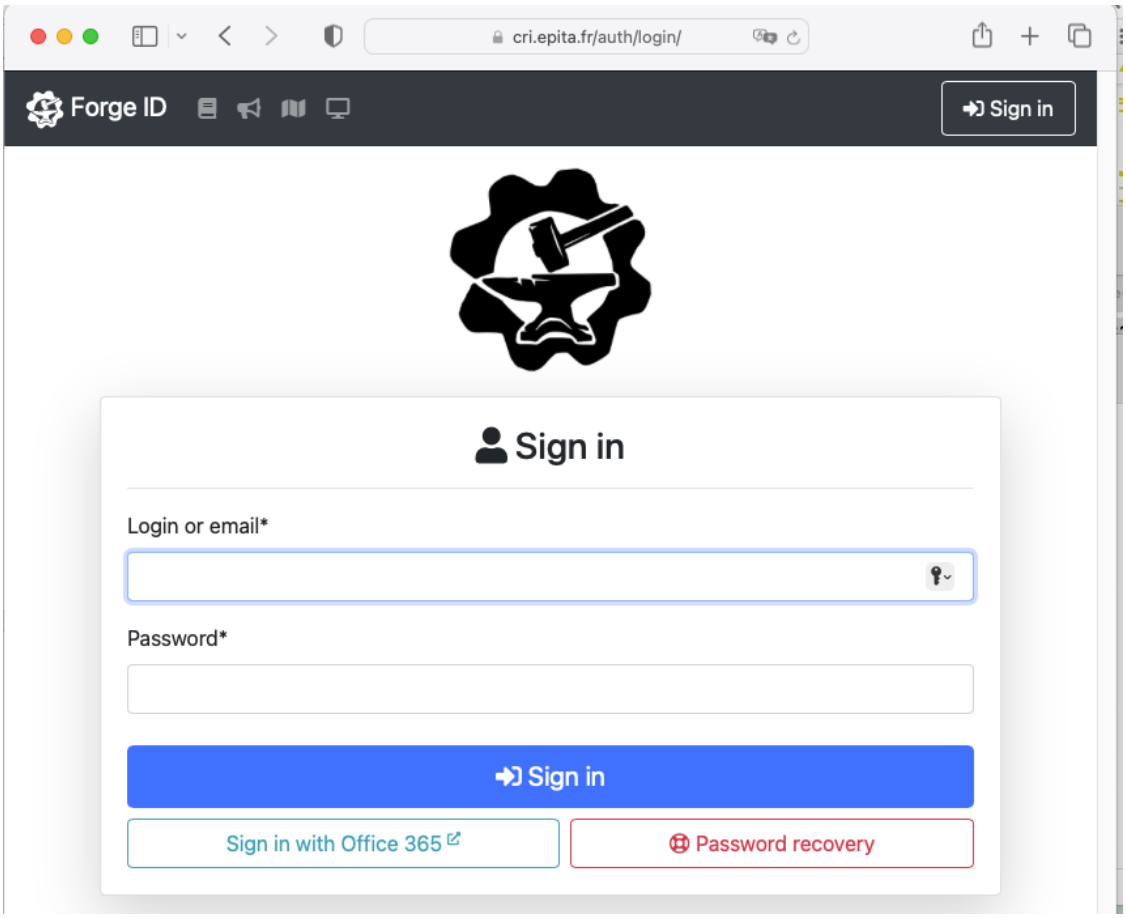
To submit your homework assignments for grading, you'll use the Forge/Intranet services. EPITA students often refer to this service simply as *intra*.

Once you have copied your ssa public key into the mouse clipboard as shown in Section B.2.3, you must register the key with Forge.

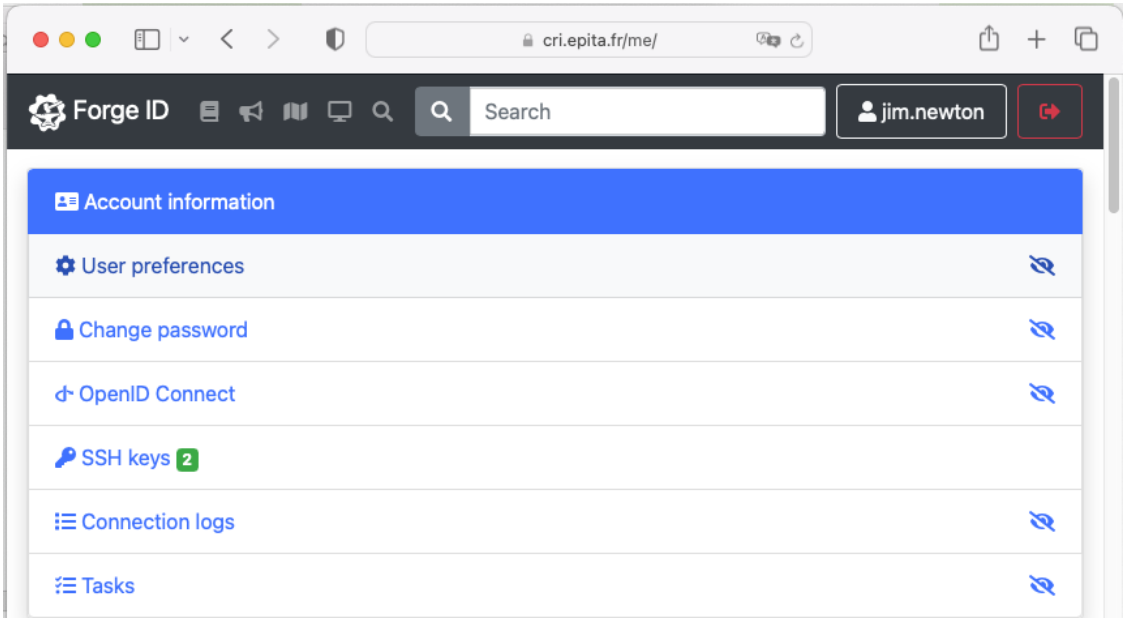
1. Use the URL `https://cri.epita.fr` to sign in to the Forge services.



2. Click the **Sign in** link.



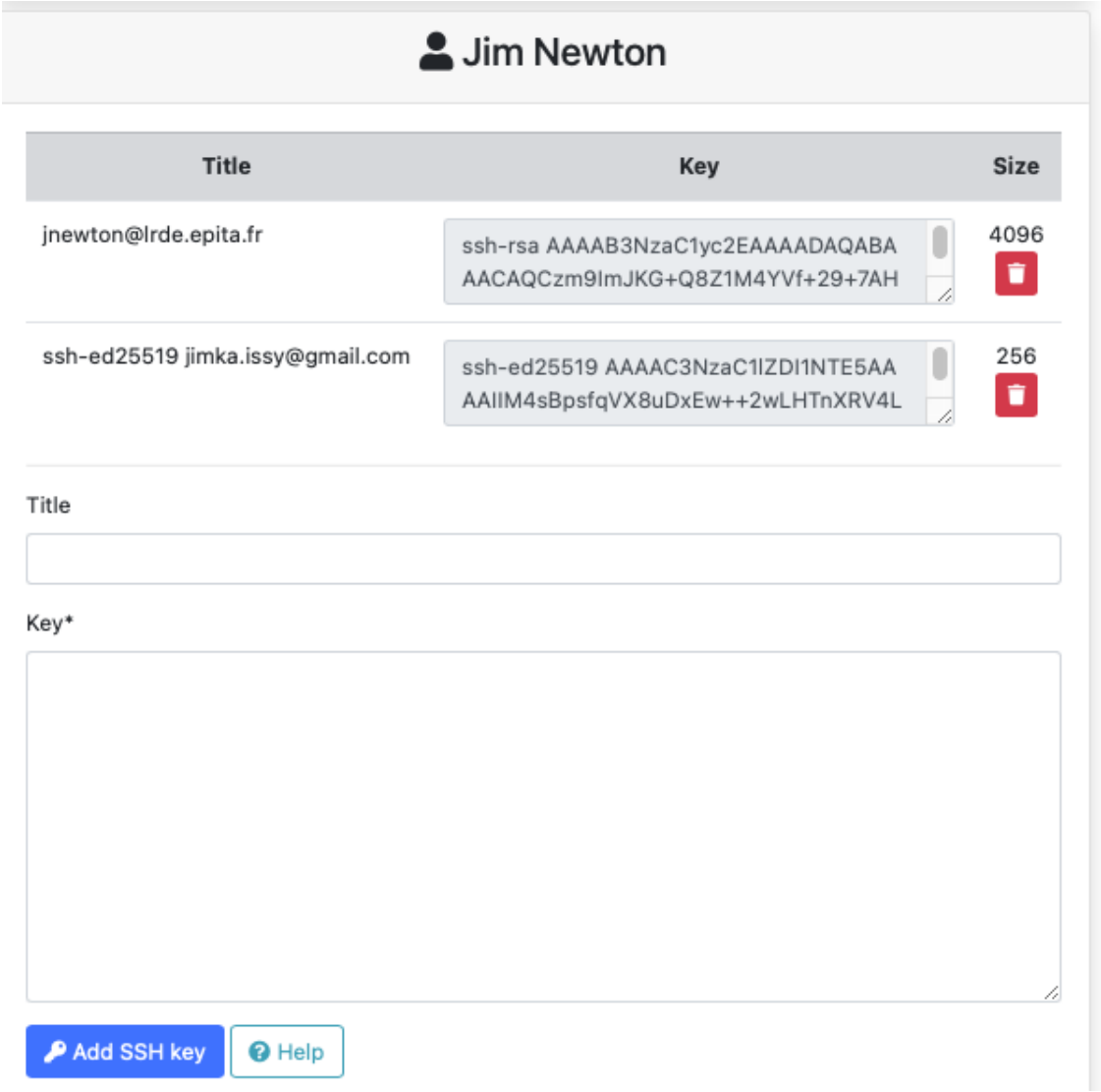
3. Enter your EPITA credentials. Your EPITA credentials are usually your first and last name separated by a dot such as `pierre.dupont` NOT `pierre.dupont@epita.fr`.



4. In the Account information form, select **SSH keys** 

.

5. You should now see the following form (with your name) at the bottom of the web page.

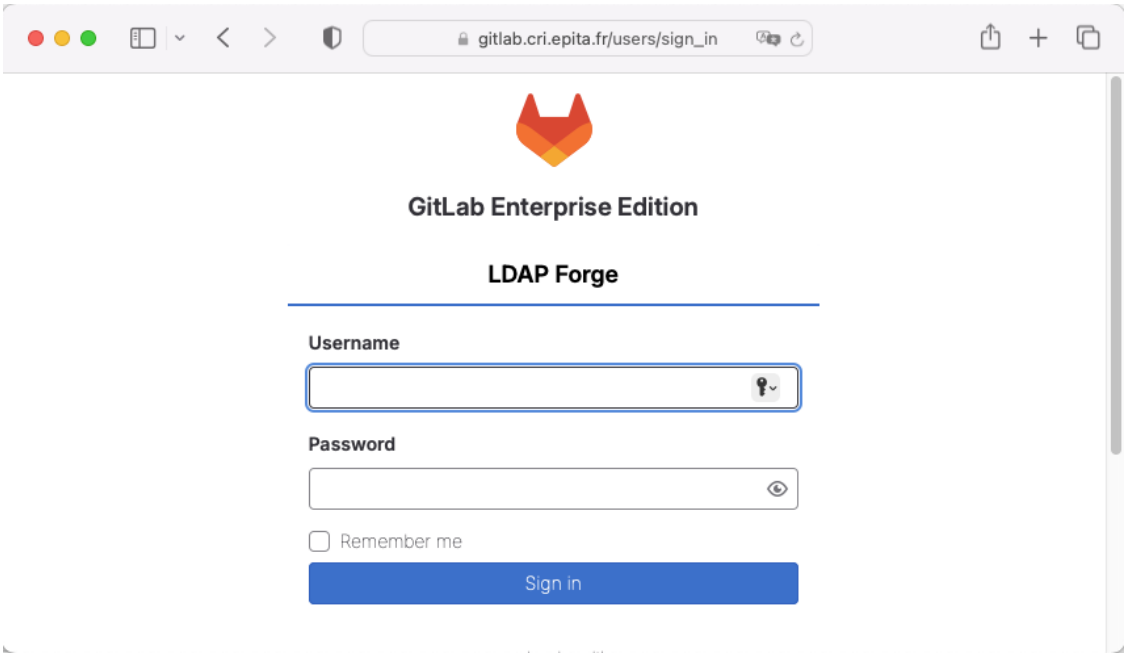


6. Find the type-in field marked **Key*** and paste your public key from the mouse clipboard (by pressing **CTL-V** (or **CMD-V** for MacOS). Recall that you copied the public key to the clip board in Section B.2.3, it is the content of the file `.ssh/id_rsa.pub`. You may be able to see the public key by entering the following command in the shell (`powershell` for Windows).

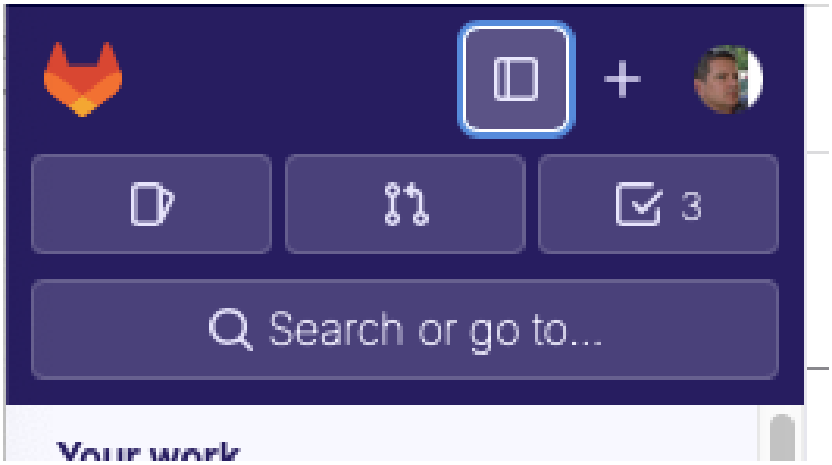
7. Then press the link **Add SSH key** once: 

. Sometimes this is slow—don't press twice. There is no need to enter a **Title**; just leave it blank.

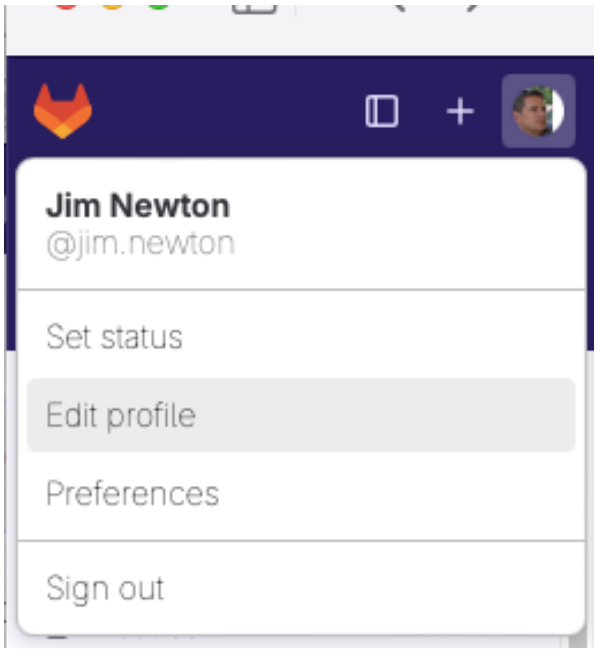
B.4 Register Public Key with GitLab



1. Log into GitLab `https://gitlab.cri.epita.fr` using your EPITA login credentials.

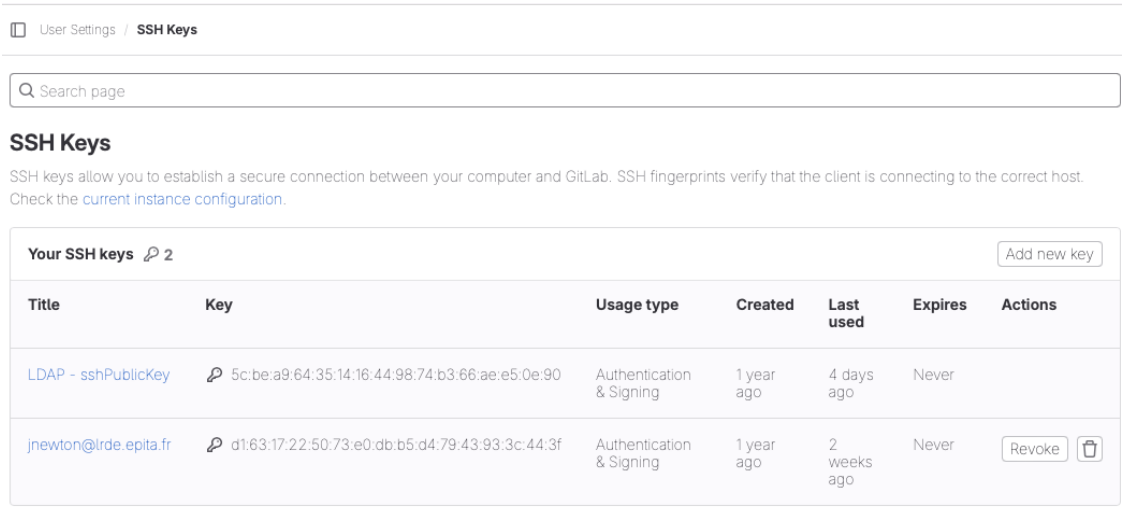


2. Click on your avatar and select **Edit Profile**.



3. Now in the left-hand side, find 

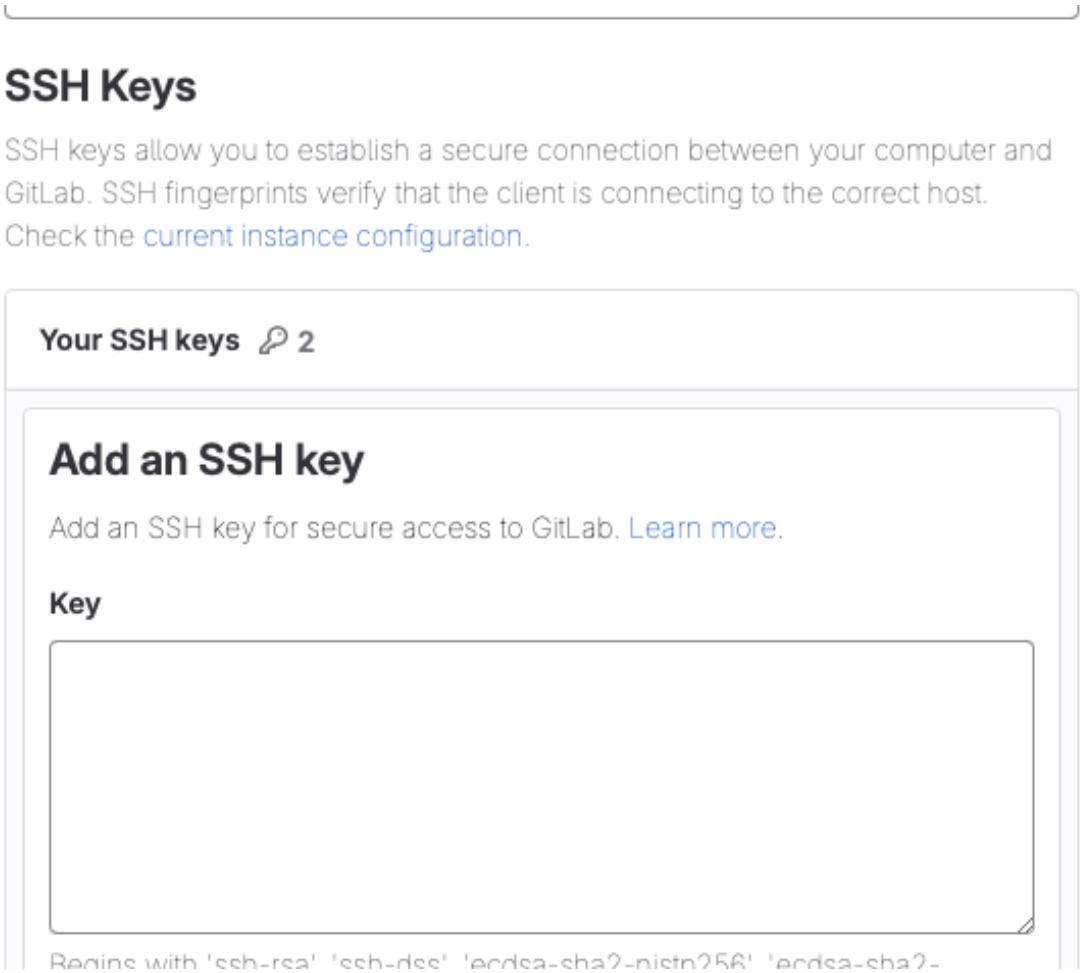
SSH Keys, and link the link, to see a page such as this. You may not yet have any keys listed.



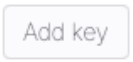
4. Click 

Add new key.

5. You will see a field labeled **Key**. Paste the public key from clipboard into the **Key** field.



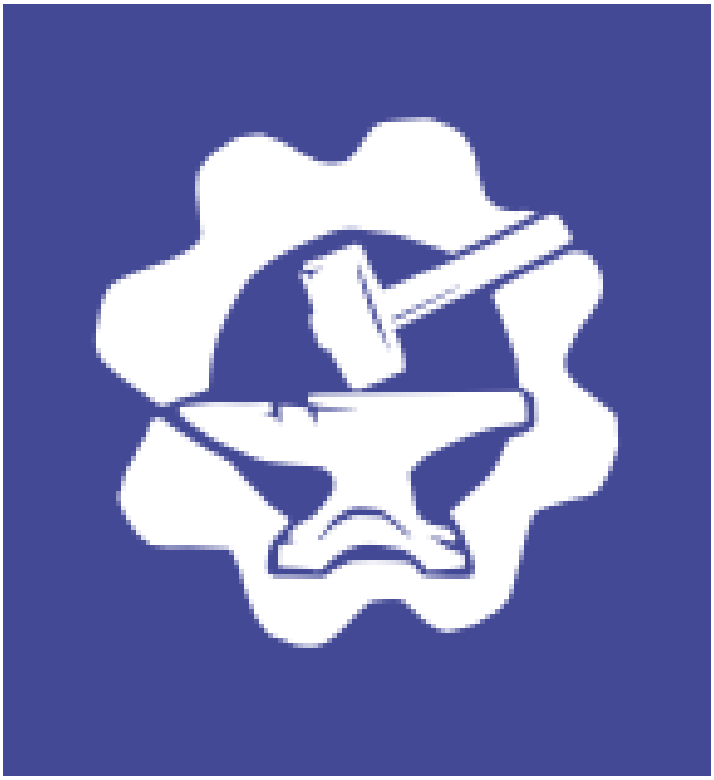
Recall that you copied the public key to the clip board in Section B.2.3, it is the content of the file `.ssh/id_rsa.pub`.

6. Finally, scroll down and find and click the 

Add key button.

Appendix C

Work Area and Code Templates



Homework assignments for this course will be submitted via Forge/Intranet. Forge/Intra is a service provided by EPITA which among other things allows students to securely submit programming assignments for automatic grading.

C.1 Running the `checkout-workarea-algebra-1.py` Script

You will download a Python script named `checkout-workarea-algebra-1.py` and run it from the command line. Follow the instructions for your operating system.

C.1.1 Step 1 — Download the Script

`https://gitlab.cri.epita.fr/jim.newton/bcs-algebra-1-student/-/raw/main/algebra/bin/checkout-workarea-algebra-1.py`

Use the URL `https://...` or the QR code above to download the file named:

`checkout-workarea-algebra-1.py`

If asked where to save it, choose **Downloads**.

C.1.2 Step 2 — Find the Downloaded File

- **Windows:** Open *File Explorer* and go to **Downloads**.
- **macOS:** Open *Finder* and go to **Downloads**.
- **Linux:** Open your file manager and go to **Downloads**.

C.1.3 Step 3 — Open the Terminal

- **Windows:** Press **Start**, type `PowerShell`, press **Enter**.
- **macOS:** Press **Cmd + Space**, type `Terminal`, press **Enter**.
- **Linux:** Open `Terminal` from applications.

C.1.4 Step 4 — Go to the Downloads Folder

Type the command below for your system and press **Enter**:

- **Windows (PowerShell):**

  ```
  cd $HOME\Downloads
  ```

- **macOS:**

  ```
  cd ~/Downloads
  ```

- **Linux:**

  ```
  cd ~/Downloads
  ```

C.1.5 Step 5 — Run the Script

Run the script by typing:

```
python checkout-workarea-algebra-1.py
```

If that does not work, try:

```
python3 checkout-workarea-algebra-1.py
```

C.1.6 Troubleshooting

- **Error:** `python: command not found`
  Try: `python3`

- **Error:** `No such file or directory`
  Make sure you changed directory to **Downloads** first.

C.1.7 Checkout the Grader Repository

At this point you have followed the above instructions and are running the script `checkout-workarea-algebra-1.py`

The script will prompt you for several pieces of input:

1. User Real Name
2. GitLab Login
3. User EPITA Email Address

Listing C.1.1

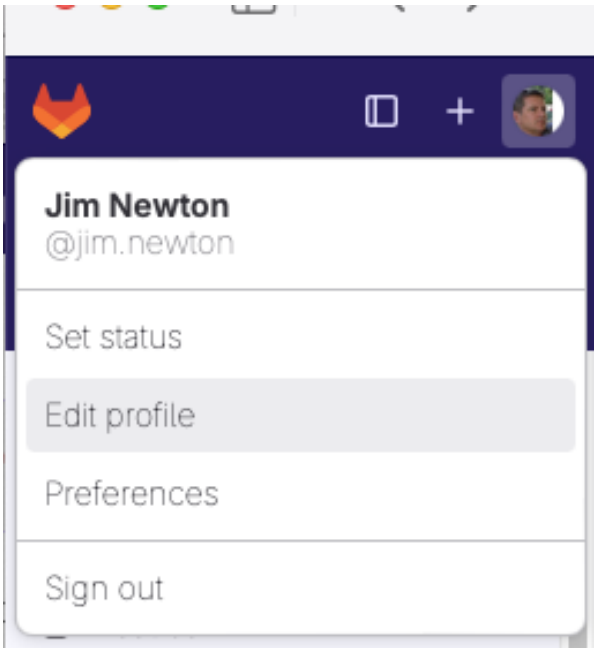
```
User Real Name[Joan Crawford]:
```

If the default in brackets is correct simply press Enter/Return to accept it, otherwise type your real name. Real names are names like `Joan Crawford`, or `Aretha Franklin`. A real name is not something like `joan.crawford` nor `aretha.franklin`.

Listing C.1.2

```
GitLab Login[joan.crawford]:
```

This is your EPITA name used to login to `gitlab.cri.epita.fr`. If you are unsure, you may verify; see Section B.4. Your GitLab Login is the text after (but not including) the `@` character in your GitLab profile.



Listing C.1.3

```
User EPITA Email Address[joan.crawford@epita.fr]:
```

This is your EPITA email address, not your gmail address nor any other private email address. It should end with `@epita.fr`.

C.2 Repository Overview

In the directory `doc/` you will find the file:

Listing C.2.1

```
algebra-1-handout.pdf
```

In the directory `algebra/algebra/` you will find the code template files. These are the files that you will complete one by one, over the semester.

Listing C.2.2

```
__init__.py     function.py   lset.py          relation.py
choose.py       group.py      monoid.py        ring.py
division.py     hello.py      polynomial.py
fibonacci.py    limit.py      roots.py
field.py        looping.py    recursion.py
```

Likewise, in the directory `algebra/tests/` you will find the files used for testing:

Listing C.2.3

```
__init__.py          hello_test.py         recursion_test.py
algtest.py           limit_test.py         relation_test.py
choose_test.py       looping_test.py       ring_test.py
division_test.py     lset_test.py          roots_test.py
fibonacci_test.py    monoid_test.py
function_test.py     polynomial_test.py
```

Finally, in the directory `algebra/bin/` you will find the script used for submitting to the grader, Section D.2

Listing C.2.4

```
submit-to-grader.py
```

Each chapter of the handout requires the student to complete one or more Python. The name of the file must match exactly what is expected; *e.g.*, `hello.py` or `lset.py`.

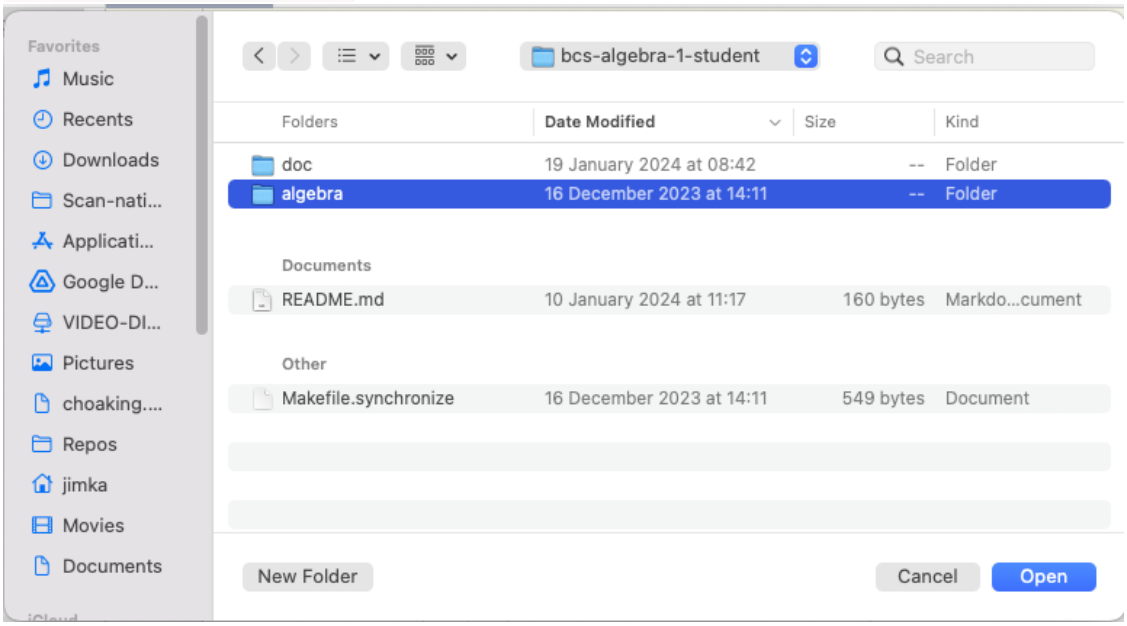
To complete an assignment, find the correct file in the `algebra/algebra` directory, *e.g.*, `algebra/algebra/hello.py`; find all occurrences in the file of `raise NotImplementedError`, and replace with code which passes the corresponding tests. To test `hello.py` you must run the tests in `algebra/tests/hello_test.py`. When all the tests pass, you are ready to submit.

Appendix D

Running the IDE

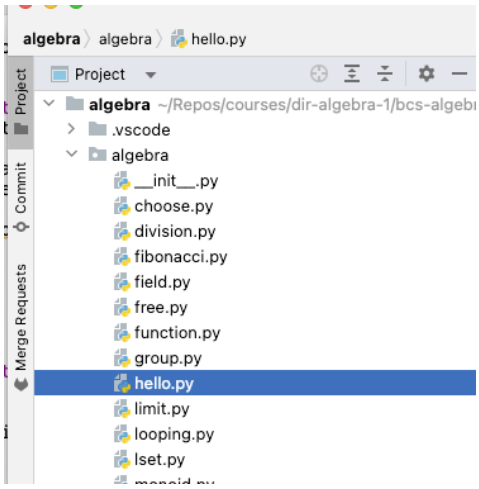
D.1 Implement Hello World

Start your IDE (PyCharm or VSCode) using `algebra-1-grader/algebra/` as the top level project directory. CAREFUL, be sure to open the `algebra/` directory, not the `algebra-1-grader/` directory.



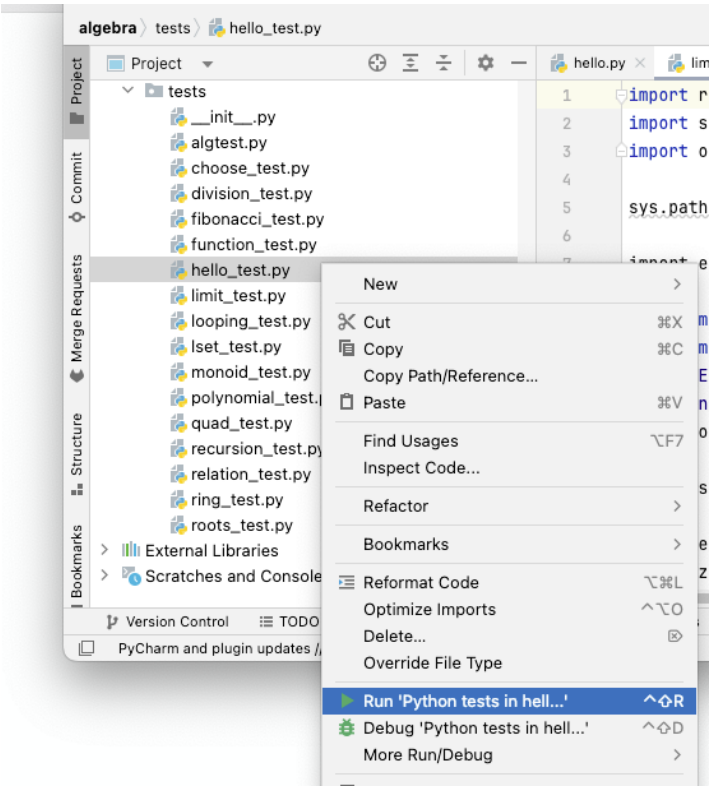
Use your knowledge of the Python language to complete the `hello.py` assignment.

1. Start with the file `algebra/hello.py`.



2. Find all occurrences of `raise NotImplementedError`. Replace `raise NotImplementedError` with the correct Python code.

3. Test the code with `tests/hello_test.py`.



4. Debug the code and repeat the process until the tests pass.

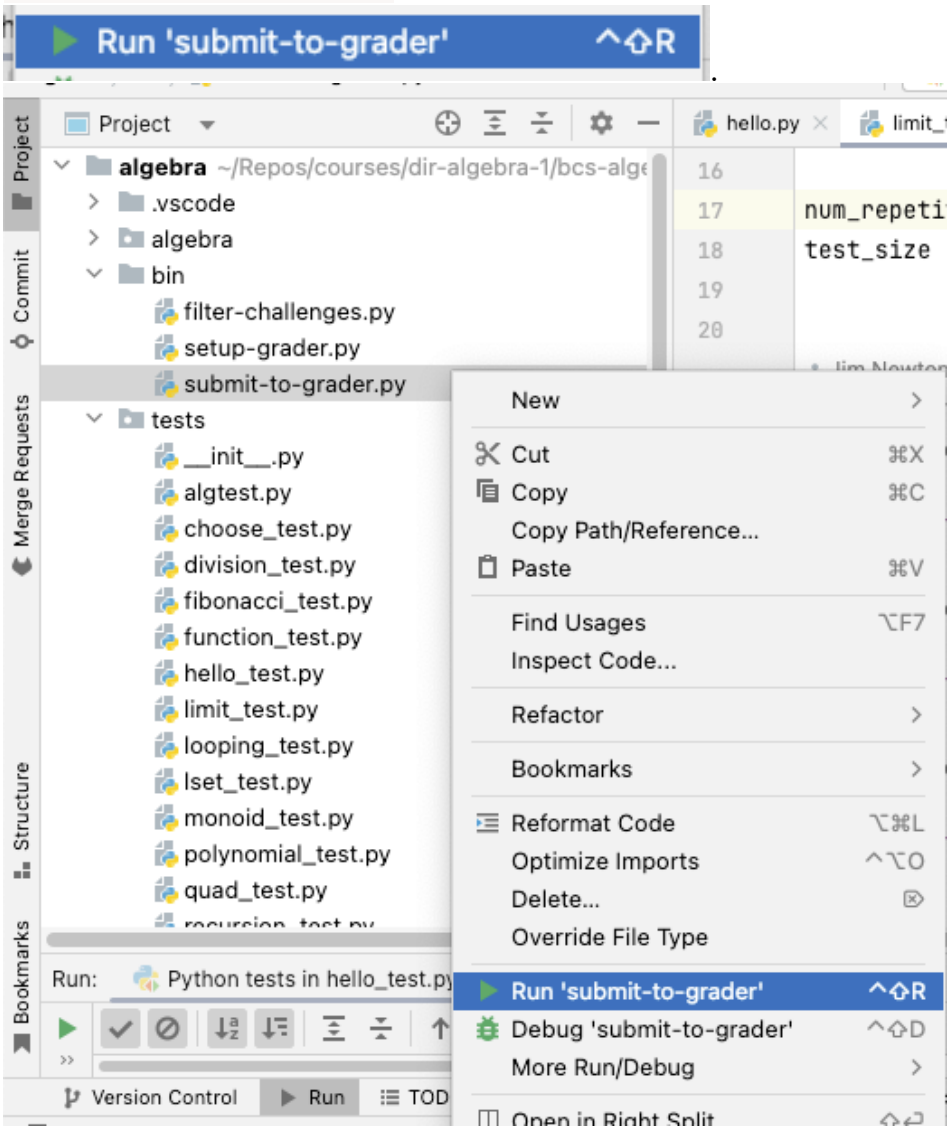
D.2 Submit Hello World

Once the `hello.py` code is working, you'll need to submit it to the autograder called *intra*. Note, you may also submit before ALL the tests pass. You may submit as many times as you like. If you have trouble getting ALL tests to pass, then submit what you have and receive partial credit. You can perfect the code later.

D.2.1 PyCharm Submit from Menu

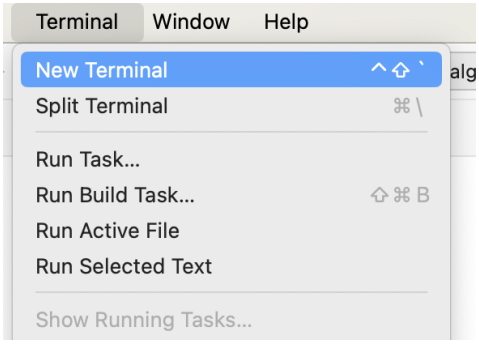
There are several ways to submit. First the easy way which will work most of the time.

If you are using PyCharm, find the file in the `submit-to-grader.py` file browser tree and select the menu item



D.2.2 VS Code Submit from Terminal

In `VS Code`, open **Terminal** using the menu item.



In the interactive terminal that opens, enter the following command

Listing D.2.1

```
cmd> python bin/submit-to-grader.py
```

You will be prompted to enter a prefix: Enter `Hello` (or any other prefix for a future homework assignment).

You should see output such as the following:

Listing D.2.2

```
python bin/submit-to-grader.py
Enter a tag prefix: Hello
Deleted tag 'Hello-2025-08-14T10-20-25-0187aa2' (was 0187aa2)
Deleted tag 'Hello-2025-08-14T10-24-08-0187aa2' (was 0187aa2)
Deleted tag 'Hello-2025-08-14T16-25-35-87c8c9b' (was 87c8c9b)
cmd = ['git', 'add', '--all']
cmd = ['git', 'commit', '-am', 'committing for tag Hello-']
[master 0fe36ae] committing for tag Hello-
 1 file changed, 1 insertion(+)
cmd = ['git', 'tag', 'Hello-2025-08-14T16-36-54.fe36ae']
cmd = ['git', 'push']
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 507 bytes | 507.00 KiB/s, done.
Total 5 (delta 3), reused 0 (delta 0), pack-reused 0
To file:///Users/jnewton/Repos/courses/dir-algebra-1/empty
   87c8c9b..0fe36ae  master -> master
cmd = ['git', 'push', '--tags']
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To file:///Users/jnewton/Repos/courses/dir-algebra-1/empty
 * [new tag]         Hello-2025-08-14T16-36-54-0fe36ae ->
   Hello-2025-08-14T16-36-54-0fe36ae
```

D.2.3 Submit from Shell

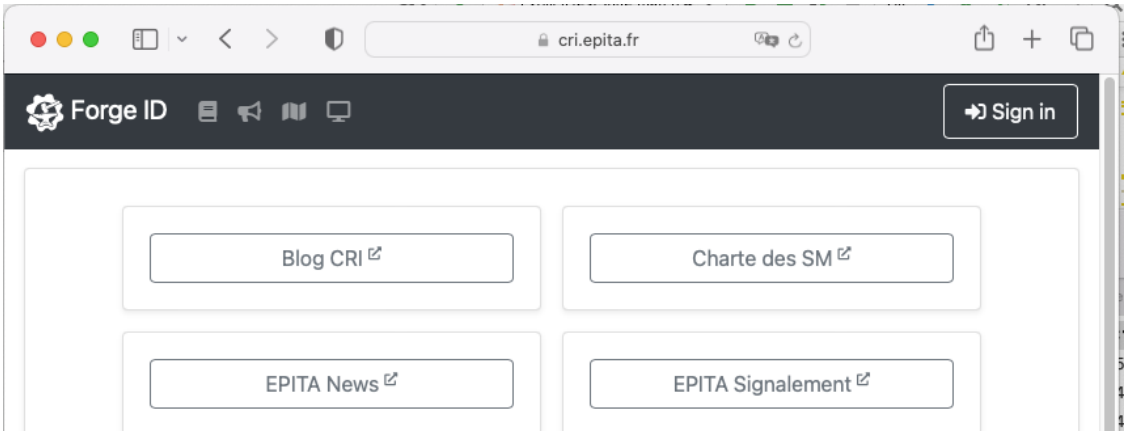
If the `submit-to-grader.py` fails for some reason, there may be something unusual with your work area. You'll have to debug it from the shell (or `powershell`) using `git` commands.

The normal way to submit from the shell is with the following commands. You'll need to substitute a unique suffix for the `XXX`, such as `HELLO-1` or `HELLO-24-09-2025` or any suffix which has not already been used. You may find all existing tags using `git tag` and `git ls-remote --tags`.

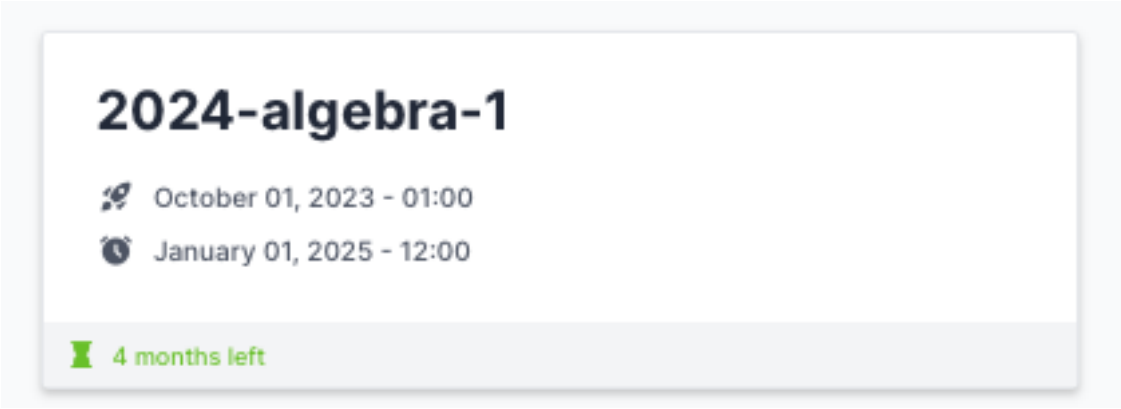
Listing D.2.3

```
cmd> git add -A
cmd> git commit -m'commit for grading'
cmd> git tag HELLO-XXX
cmd> git push
cmd> git push --tags
```

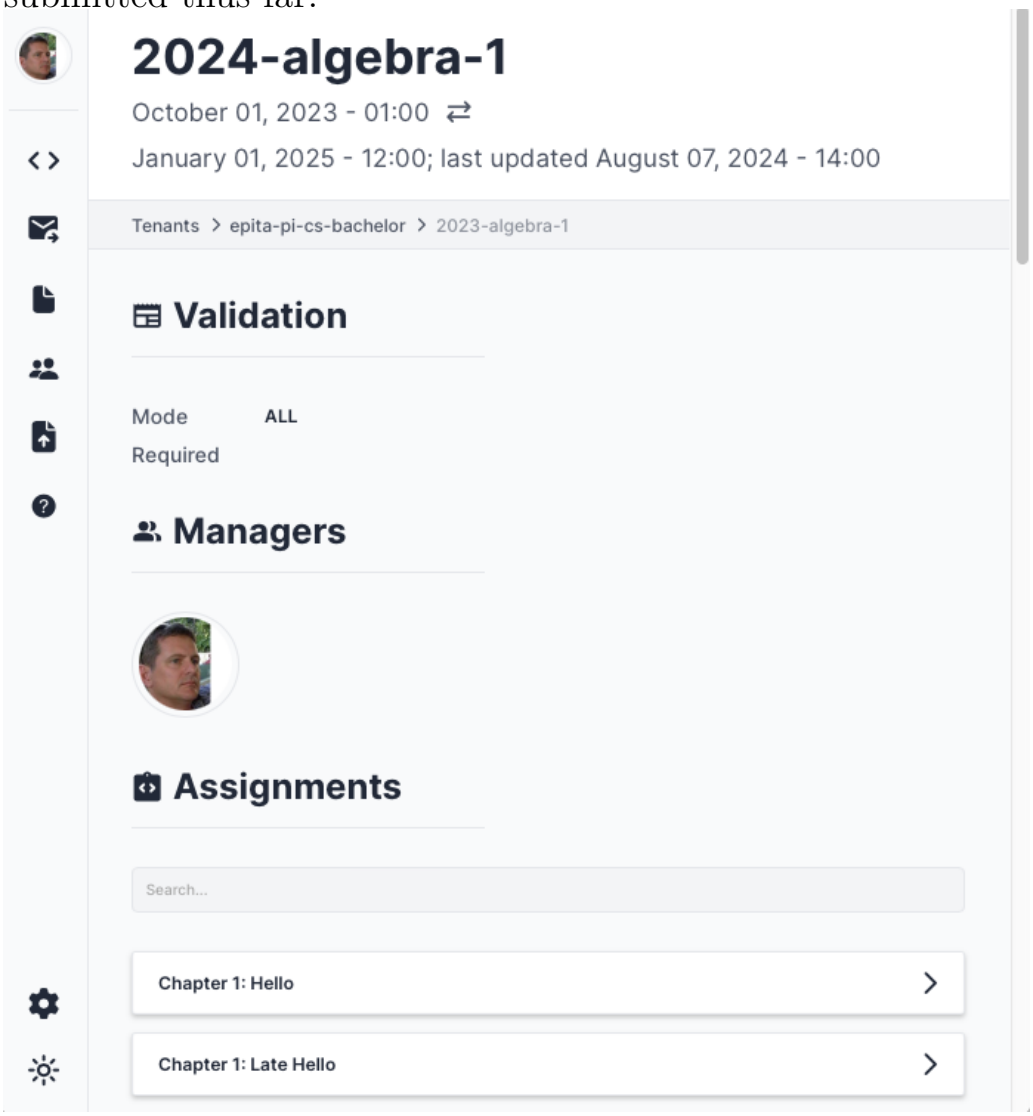
D.3 Viewing Results on Forge/Intranet

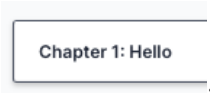


From the page `https://cri.epita.fr` find and click the link Intranet Forge, or use `https://intra.forge.epita.fr` and click the link:



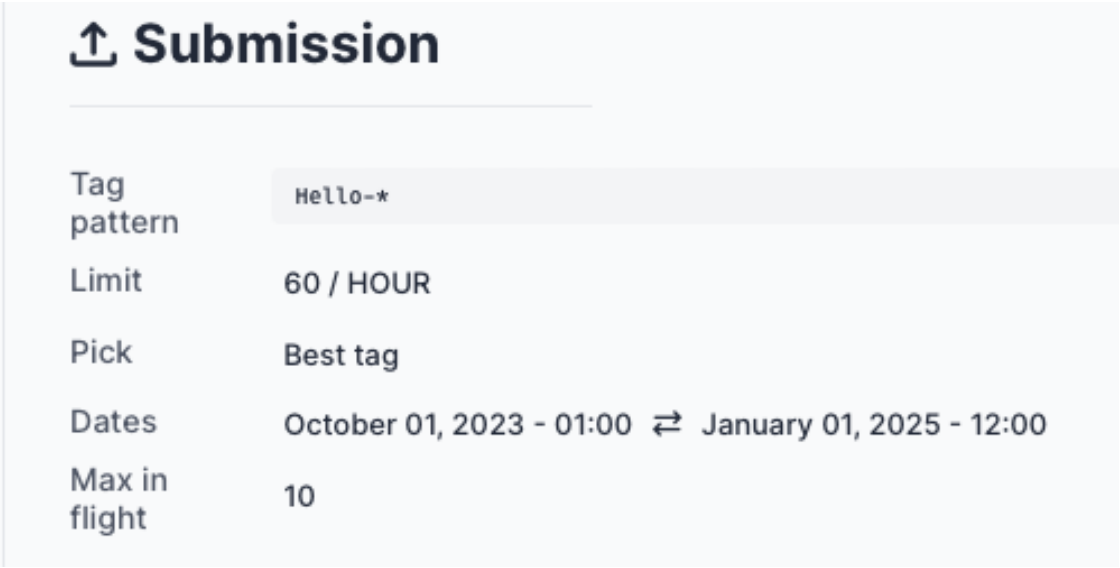
This will lead you to a page which lists all the assignments you have submitted thus far.



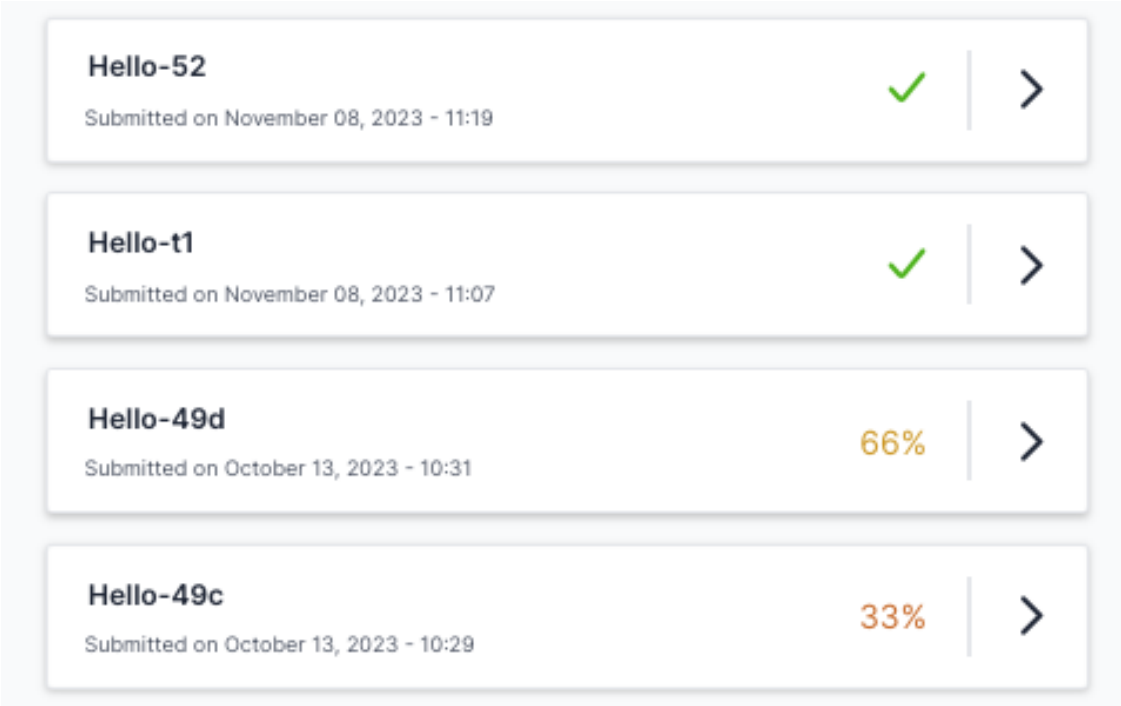
Click on 

.

If you have not yet submitted anything for this assignment, you can find the **Submission** section. This section announces what the tag prefix is for submitting an assignment to this category. In this case it is `Hello`, but other assignments have different prefixes. Intra uses this prefix to determine which assignment to grade and display in the web page.

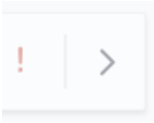
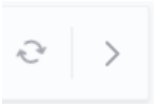
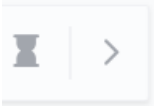


If you have already submitted at least one tag with the correct prefix, the link will take you to a page where you can see all the tags you have submitted.

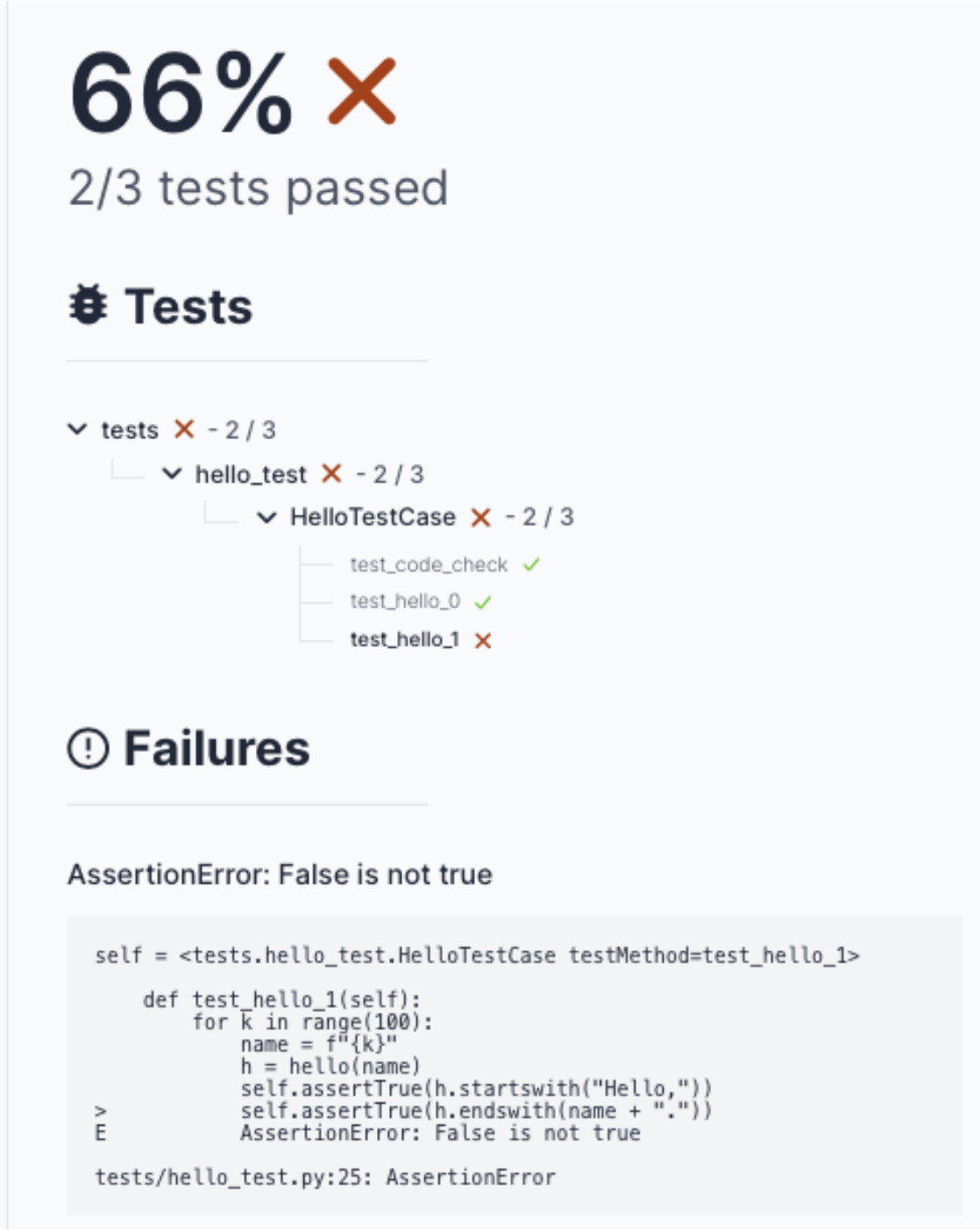


You'll see a  if you have a perfect score; this check means the assignment has been *validated*. You'll see a percentage indicating your score otherwise. If you have less than a perfect score, you may modify your code and submit again.

Other status icons you may see are

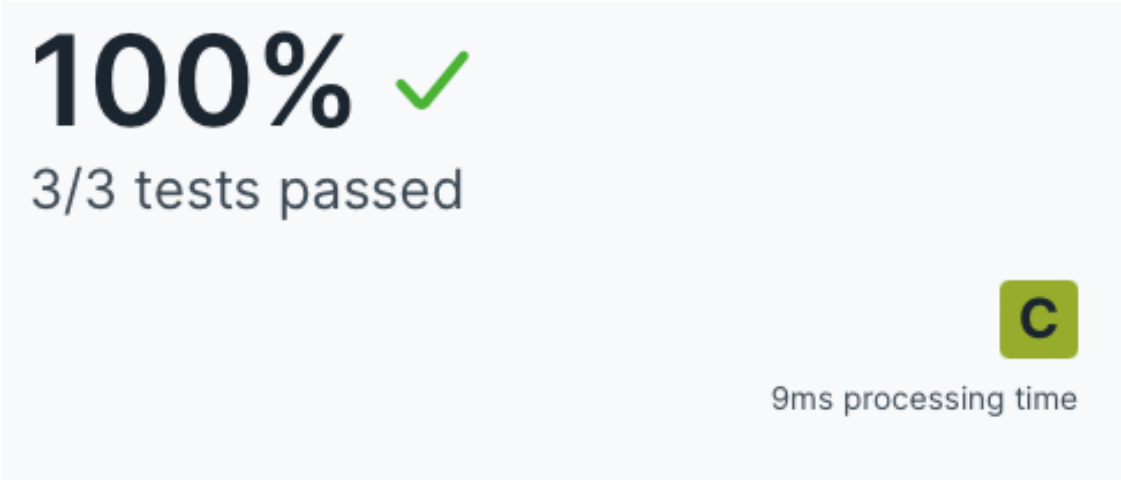
-  Job error
-  Job still running
-  Job waiting to run

Click on the  to see which tests passed and which failed and what the error message of the failure was.



The error message tells exactly which test failed. It is the student's responsibility to find the testing file and figure out exactly which assertion failed, and figure out how to fix it.

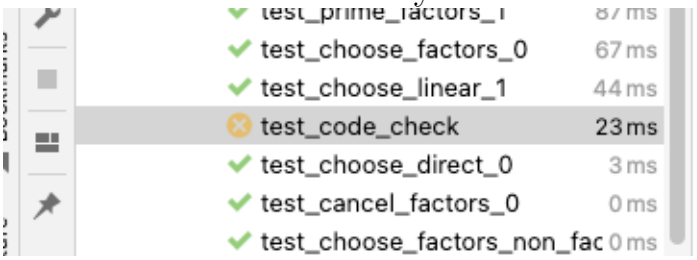
See the page `https://intra.forge.epita.fr/help` for a description the possible states (or states) of a *trace*.



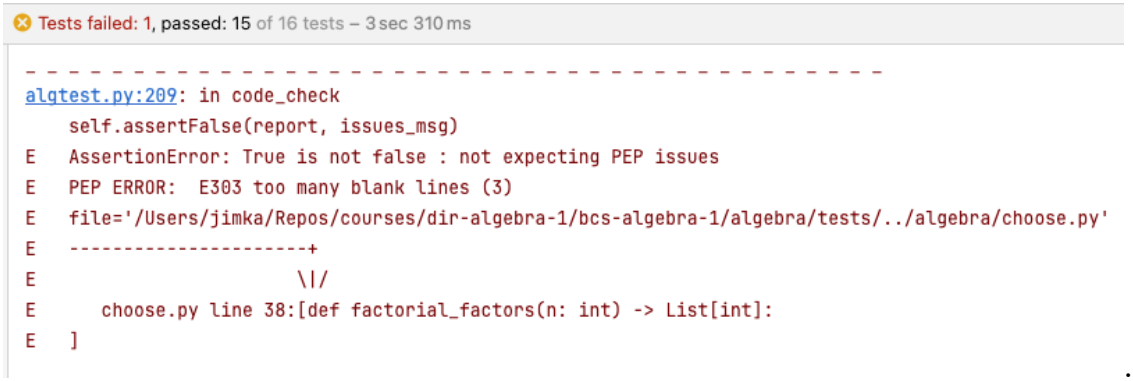
You may see a label with a letter such as A, B, C, etc labeled `processing time`. You can ignore this. It has no meaning for these exercises. It is measuring how much time is taken to run your code together with all the testing framework. Your job is NOT to write code that is fast, but rather is correct. Additionally, there is a very large overhead for the testing framework which is completely out of the control of the student.

D.4 Code Check

Special attention must be given to the test called `test_code_check` which you will see on every homework assignment. This check verifies that the submitted code obeys certain standard formatting rules.

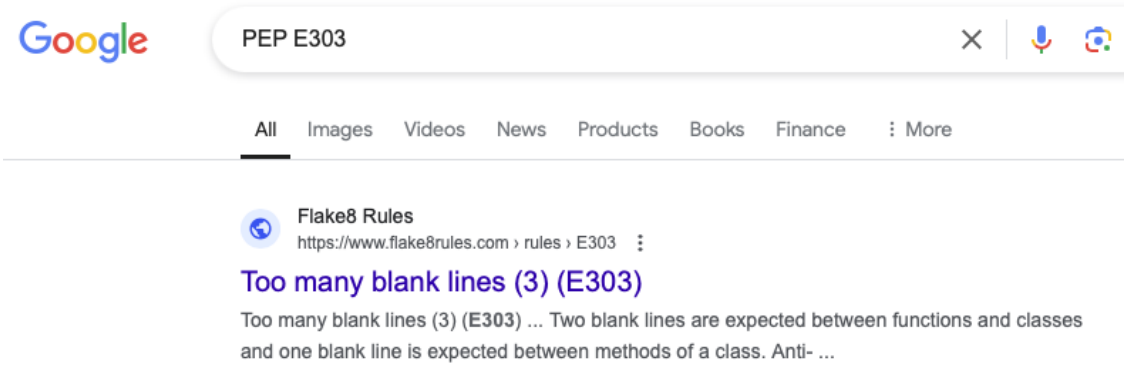


The error message explains at least the first formatting error discovered. The error message indicates the file name, line number, and column where the formatting error is discovered. Also the error message usually indicates a standard name for the error such as `PEP ERROR: PEP303`.



D.4.1 PEP Errors

To find out more about the description and perhaps motivation of the standard, you may easily search the Internet for the explanation.



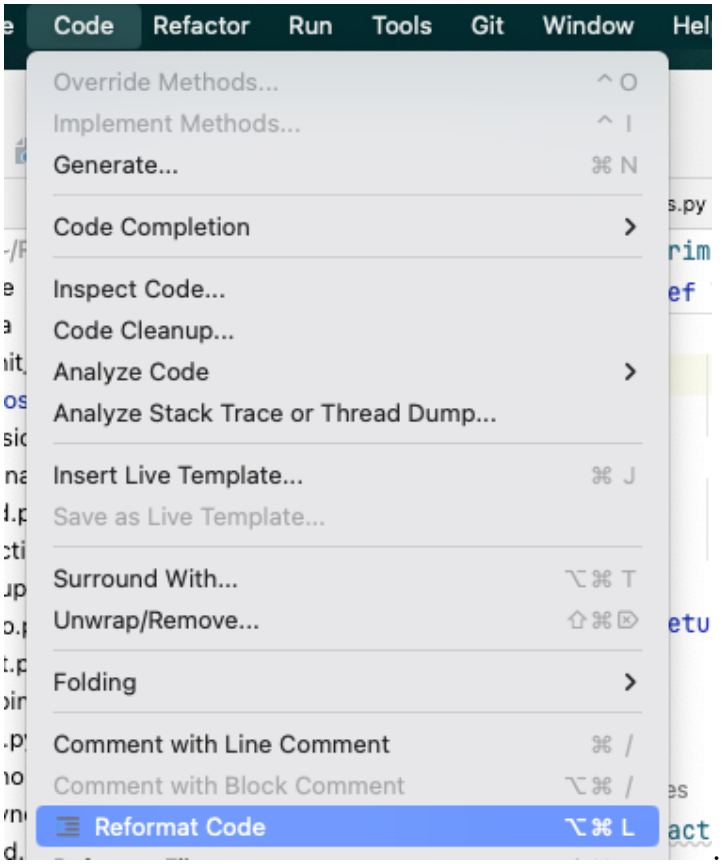
D.4.2 How to Fix a Code Check Error

There are several ways to fix a Code Check error.

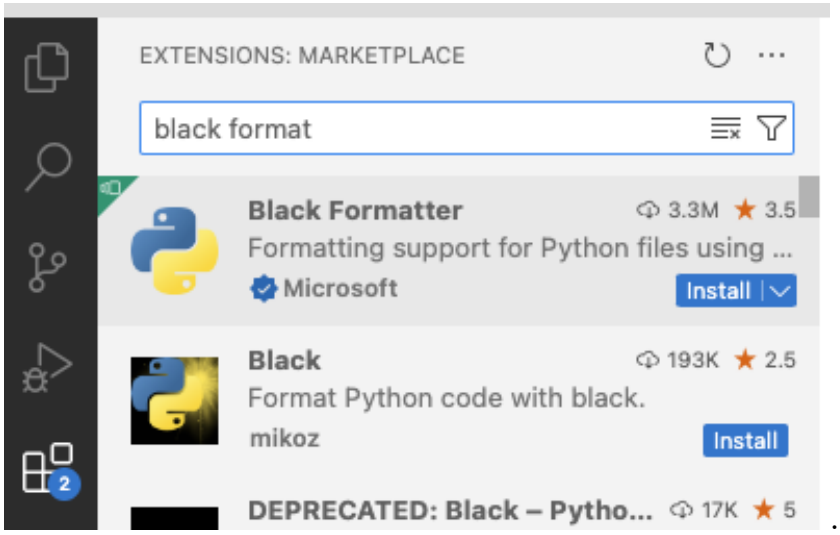
1. Manually modify the code at the indicated line number. This approach tends to be tedious because you may have many such error to fix, one by one.

2. Fix using the PyCharm **Reformat the file...** pop-up menu item.

3. Fix using the PyCharm pull-down menu item `Code -> Reformat File...`



4. Fix using the VS Code plug-in **Black Formatter**. You may need to install **Black Formatter**.



After **Black Formatter** has been installed, you can use the pop-up menu to reformat the entire file. This will fix *most* but perhaps not all of the PEP errors. The remaining formatting errors must be resolved manually if you are using VS Code as your IDE.