Omar Hijab

Discrete Math With Python

A Primer

August 24, 2020

Preface

This book is a discrete math introduction targeted at undergraduates in mathematics, computer science, and other science or engineering fields.

Typically, these students are exposed to this material early in their undergraduate career, after calculus, and before diving into more abstract mathematics or computer science courses. At this point, students face two difficulties, the gap between mathematics and coding, and the gap between technique and perspective.

One of the lessons of this text is that approaching both coding and mathematics with thoughtfulness and logic can open up new vistas and modes of thinking, a key goal of any transitions course.

To bridge the gap between the two cultures, we include coding as an integral part of the text. Python is a natural choice, given its mathematical nature and smooth syntax. This inclusion is made, on the one hand, to address the needs of students in data science who may need a review of discrete math in a succinct manner, and, on the other hand, for students who are new to coding.

Whether for machine learning, cryptography, error-correcting codes, or big data, scientists at all levels often need to learn abstract algebra or number theory. These subjects may seem bewildering to the uninitiated. For this audience, this text may serve as a useful introduction.

Six discrete math core areas are *functions and sets*, *logic*, *proofs*, *counting*, *graphs*, and *probability*. We leave out graphs and probability, and we present the other topics *in vivo*, rather than *in vitro*, by choosing arithmetic, both standard and modular, as our main vehicle.

Students may not appreciate the need for, or the power of, pure thought, unless they are presented with striking concrete results. We do not discuss proof techniques explicitly; instead proofs and derivations are carefully constructed, starting from first principles, as needed.

High points we touch on are square roots in modular arithmetic, and square roots in continued fractions. Since this is a discrete math text, we handle square roots algebraically, and avoid real numbers. We also include RSA encryption, given its pervasive presence in our lives. To the extent possible, corresponding Python coding is included either in the text or in the exercises.

Instructors can choose to emphasize mathematics more, or coding more. Some of the exercises may be fleshed out into end-of-semester projects. The uniqueness of $\mathbf{Z}$, $\mathbf{Z}_n$, and $\mathbf{Q}$, and the details of $\mathbf{Q}(\sqrt{D})$ are the most technical topics; these are presented in Appendix A. We do this so the reader is not distracted as they study the main text.

I thank my friends and colleagues Eric Grinberg, Fayez Al Hammadi, Salam Al-Rawi, Munther Hindi, and Kei Lutalo for their help and support, and Yevgeniya Rivers for getting me to start this project. Finally, I am most grateful to Rawa for their hospitality and generosity during my stay there while writing this book.

New Haven, Spring 2020 *Omar Hijab*

Contents

List of Figures

A Note to the Reader

This book is meant to be read side-by-side with a laptop running Python. It is essential to check *everything* by running it in your Python interpreter. *Everything can be checked.* This is a feature, not a bug.

In this book, math concepts are introduced and discussed in parallel with analogous Python concepts. To help keep things straight, Python concepts and code are in `typewriter font`, often color highlighted to emphasize syntax. Code snippets include shell prompts >>>, to differentiate user input from Python output. Shell prompts are sometimes written `2>>` to indicate the indentation level.

It's a two-way street. Understanding coding will help you understand the math, and understanding the math will help you understand coding.

That's just the way math and coding work. On the plus side, if you put in the work, you will see and understand aspects of the world you never even knew existed. When you master coding, you will be able to understand aspects of coding you never even knew existed.

All code in the text is executed with Python3. The web is your friend. You can find more than you want to know about any aspect of Python with even rudimentary searches. For beginners, the book [3] is the best I've seen. The official source for Python documentation is here[1]. However, you may find other sites easier to digest, at least on a first pass.

If you wish to learn more about discrete math, the bibliography at the end is a good start. Among the books in the bibliography, [6] is an excellent beginning.

The book is divided into chapters, each of which is divided into sections. The third section in the fourth chapter is referenced as §4.3. The second equation in the third section of the fourth chapter is referenced as (4.3.2). The second theorem in the third section of the fourth chapter is referenced as Theorem 4.3.2. The second exercise in the fourth chapter is referenced as Exercise 4.2.

In the text, denotes the end of a proof, $\implies$ means *implies*, and $\iff$ means *if and only if*.

[1] `https://docs.python.org/`

Chapter 1
Objects

1.1 Types

The four built-in primitive types in Python are `int`, `float`, `bool`, and `str`,

```
>>>  type(23)
int
>>>  type(23.4)
float
>>>  type(True)
bool
>>>  type('alpha')
str
>>>  type('23')
str
```

`int`, `float`, `bool`, and `str` are *types*[1]

```
>>>  type(int)
type
```

An `int` is an integer, positive, negative, or zero. A `float` is Python's approximation of a real number, consisting of an integer part and a fractional part. A `str`, a *string,* and consists of a sequence of characters. A `bool` is a *boolean,* and takes on the values `True` or `False`.

Since

```
>>>  int(23.4)
23
>>>  23.4 - int(23.4)
0.3999999999999986
```

[1] Depending on your Python environment, `type(23)` may return `<class 'int'>` instead.

the fractional part is not exactly `.4`; this is because a Python `float` is only an approximate representation of a real number. Equality is == and the negation of equality is `!=`, so

```
>>>  23 == 15
False
>>>  23 != 15
True
```

One can also input

```
>>>  a = 23
>>>  b = 15
>>>  a == b
False
>>>  a != b
True
```

Here 23 and 15 are *assigned* to the *variables* `a` and `b`. In Python, equality is == and assignment is =.[2]

Variables names may be as long as you like, and be made up of numbers, letters, and the underscore _, but they can't start with a number,

```
>>>  2abc = 23
Error: invalid syntax
>>>  number_of_apples = 2
>>>  number_of_oranges = 17
>>>  number_of_apples + number_of_oranges
19
```

The strings `'int'`, `'float'`, `'bool'`, and `'str'` are *keywords*. There are around 35 keywords in Python. You may get a complete list by running

```
>>>  import keywords
>>>  keyword.kwlist
```

Data in Python is stored as *objects*. Every `object` has an *identity*,[3] a *type* and a *value*. An `object`s type determines the operations that the `object` supports. Python is a *strongly typed* language: An `object`s identity and type never change once the `object` has been created.

```
>>>  id(23)
4387796800
>>>  id('alpha')
4391932528
```

The values of some `object`s can change. Objects whose values can change are said to be *mutable*. Objects whose values are unchangeable are called *immutable*.

[2] In math, the same symbol = is used for both assignment and equality.

[3] Think of an `object`'s identity as its location in memory.

Objects of type `int`, `float`, `bool`, and `str` are immutable. Later we meet `object`s of type `list`, `set`, and `dict`, which are mutable. Let's attempt to capitalize the first letter of `'beta'`,

```
>>>  a = 'beta'
>>>  a[0]
'b'
>>> a[0] = 'B'
Error: 'str' object does not support item assignment
```

Instead, we take the *slice* `a[1:]` of `a`, and prepend to it the string `'A'`,

```
>>>  a[1:]
'lpha'
>>>  'A' + a[1:]
'Alpha'
```

We explain slices in §1.3.

Objects are assigned to variables. Python is a *dynamically typed* language: Variables do not have `id`s, nor do variables have `type`s; only the corresponding `object`s have `id`s and `type`s. Thus `id(a)` and `type(a)` return the `id` and `type` of the `object` assigned to the variable `a`.

```
>>>  a = 23
>>>  id(23) == id(a)
True
>>>  type(23) == type(a)
True
```

and

```
>>>  a = 27
>>>  id(a)
4392818624
>>>  type(a)
int
>>>  a = 'alpha'
>>>  id(a)
4391932528
>>>  type(a)
str
```

Here we have the variable `a` pointing first to the `object` `23`, then to the `object` `27`, then to the `object` `'alpha'`.

If a variable has no `object` assigned to it, it makes no sense to ask for its `type` and `id`,

```
>>>  type(new_variable)
Error: name 'new_variable' is not defined
>>>  id(new_variable)
Error: name 'new_variable' is not defined
```

Here *not defined* means no `object` is assigned to it.

An `object` exists only at *runtime*, that is only when the code is executing in the Python interpreter. Turn the computer off, and the `object` doesn't exist. Thus, strictly speaking, an integer and an `int` are not the same: an `int` is an `object`, existing at runtime, while an integer is a number, existing within mathematics.

If your code contains the statement

```
>>>  a = 23
```

and you turn the computer off, you still have the number 23, since numbers exist independently of computers, independently of us, and independently of the physical universe.[4] You also have a variable `a`, because that's just a string in your code (stored, say, on non-volatile memory). But when the code is running, you have the `object` `23`, living inside the computer's volatile memory, and assigned to the variable `a`. An `object` is similar to a thought, which has no existence unless you are thinking it.

A string is delimited by single quotes `'` or double quotes `"`,

```
>>>  type('alpha')
str
>>>  type("alpha")
str
>>>  'alpha' == "alpha"
True
```

The string with no characters is the *empty string* `''` or `""`. Multi-line strings are delimited with triple quotes,

```
>>>  a = '''123
...  456'''
>>>  a
'123\n456'
```

Here `\n` is the *newline* character, and the three dots `...` indicate the input is a continuation of the input on the previous line, so

```
>>>  len(a)
7
```

where `len(a)` is the length of the string `a`. Note `\n` is a single character, not two, corresponding to byte `0a` (Figure 1.3). Continuing,

```
>>>  b = """123
...  456"""
>>>  b
'123\n456'
>>>  a == b
True
>>>  c = '''
```

[4] You can't even say the number 23 exists for all time, because numbers are independent of time

```
...   123
...   456
...   '''
>>>   c
'\n123\n456\n'
>>>   a == c or b == c
False
>>>   len(c)
9
```

so a consists of seven characters and two lines, and c consists of nine characters and four lines, two of which are empty lines.

The print(a) statement is used to present the object a in a more user-friendly fashion,

```
>>>   a
'123\n456'
>>>   print(a)
123
456
```

bools may be compared

```
>>>   True == False
False
>>>   True == True and False == False
True
```

The logical expression a and b or c and d is ambiguous. Which to do first, and or or? This is solved by inserting parentheses.[5] For example, the logical expressions

- (a and b) or (c and d)
- a and (b or c) and d

are unambiguous, and they have different truth values

```
>>>   a = True
>>>   b = False
>>>   c = True
>>>   d = False
>>>   (a and b) or (c and d) == a and (b or c) and d
False
```

See Exercise 1.13.

Some strs, not all strs, may be converted to ints or floats or bools,

[5] Actually, when there are no parentheses, Python always evaluates and before or, but you often want a different order.

```
>>>  int('23')
23
>>>  int('alpha')
Error: invalid literal for int( ) with base 10: 'alpha'
>>>  float('23')
23.0
>>>  bool('23')
True
>>>  bool('alpha')
True
>>>  bool('')
False
```

Conversely, `int`s or `float`s or `bool`s may be converted to `str`s,

```
>>>  str(23)
'23'
>>>  str(23.4)
'23.4'
>>>  str(True)
'True'
>>>  str('alpha')
'alpha'
```

We caution the reader that the above use of the term *converted* is, strictly speaking, not correct: `int`s, `float`s, `bool`s, and `str`s are immutable, they can't be converted. For example, the string `'23'`

```
>>>  id('23')
4422129648
```

is not converted: A new object, `int('23')`, is created,

```
>>>  id(int('23'))
4379756352
>>>  int('23') == 23
True
```

and the old object, `'23'`, is left alone. Nevertheless, it is convenient to continue to say *converted*, as long as we interpret the term correctly.

To clarify things, let us define a *function* to be a rule, a procedure, a process, a factory, a black box, anything that returns an `object`, when fed some other `object` (Figure 1.1). Our definition here is that of a Python `function`. The mathematical concept of function is defined in §3.5.

Now let's go back to `int`s. These are `object`s. How are they created? There has to be a function in the Python code designed to return these `int`s. This function is a sort of "`int` factory". This factory creates only `object`s of type `int`, not `object`s of type `str`, or any other type.

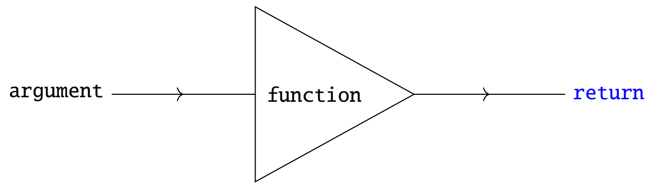

Fig. 1.1 A function.

What is the name of this factory? It is `int`. Thus `type(23)` is `int`, *because that is the name of the function that creates the* `object` 23. Similarly `type('alpha')` is `str`, because that is the name of the function that creates the `object 'alpha'`.[6] The same remarks hold for `float`, `bool`, and all other types.

When fed the string `'23'`, the function `int` is designed to return the object 23, so the `object` 23 has type `int`. When fed the empty string, the function `int` is designed to return the object `0` (`int() == 0` is `True`), so the `object` `0` has type `int`.

When fed the string `'alpha'`, the function `int` doesn't know what to do, because `int` is not designed to handle such strings, so an error is raised. The conclusion is that a type is an "object factory," and

```
>>>  type(int)
type
```

is just saying `int` is an object factory. Object factories are `object`s, *everything* is an `object`,

```
>>>  id(int)
4378625280
```

A last remark about comparison of `object`s. Two `object`s `a` and `b` are compared with the keyword `is`. If

```
>>>  a is b
True
```

then the `object`s (corresponding to the variables) `a` and `b` are the same, This is different than `a == b`, which compares the *values* of `a` and `b`, not the full `object`s (see Exercise 1.7 and Exercise 2.14).

1.2 Octal, Binary, and Hexadecimal

Decimal notation is a representation of numbers; as such, decimal notation is a string. Explicitly

[6] This is not as abstract as it sounds. If you're holding a soda can in your hand, and someone asks "what type of soda is that?", you would say "Pepsi", since that is the name of the factory the soda can came from.

```
>>>  a = '23'
>>>  type(a)
str
>>>  type(int(a))
int
>>>int(a)
23
```

When you input `23`, you are typing the character `'2'` followed by the character `'3'`. In other words, you are entering the string `'23'`. However Python automatically interprets it as an `int`,

```
>>>  23 == int('23')
True
```

When we see the string `'23'`, we do not see the number 2 followed by the number 3; we see the single number 23, which is $2 \cdot 10 + 3$, where $+$ is addition, and $\cdot$ is multiplication.[7] When we see the string `'423'`, we do not see the number 4 followed by the number 2 followed by the number 3; we see the single number

$$4 \cdot 10^2 + 2 \cdot 10 + 3 = 4 \cdot 100 + 2 \cdot 10 + 3 = 423.$$

More generally, in octal or in base 8, the string `'23'` corresponds to the number $2 \cdot 8 + 3$ which equals the number 19. In octal, the string `'423'` corresponds to

$$4 \cdot 8^2 + 2 \cdot 8 + 3 = 4 \cdot 64 + 2 \cdot 8 + 3 = 275.$$

To indicate this, we also may prefix the string `'23'` with a `'0o'`, a zero followed by lowercase `'o'`,

```
>>>  int('23',8)
19
>>>  0o23 == 19 == int('23',8) == int('0o23',8)
True
```

The closely related function `oct( )` takes an `int` and returns the octal *string* representation,

```
>>>  oct(19)
'0o23'
>>>  19 == int(oct(19),8) == 0o23
True
>>>  int('23',8) == oct(19)
False
```

Thus the function `oct( )` is the *inverse* of the function `int( ,8)`: whatever the `int a`, we have

[7] In Python, multiplication is `a*b` and powers are `a**5 = a*a*a*a*a`. In math, multiplication is ab or $a \cdot b$, and powers are $a^5 = aaaaa = a \cdot a \cdot a \cdot a \cdot a$.

```
>>> int(oct(a),8) == a
True
```

Moreover, in octal representations, the digits are 0, 1, 2, 3, 4, 5, 6, 7 only. There are no *digits* 8 or 9,

```
>>> int('9',8)
Error: invalid literal for int( ) with base 8: '9'
```

but the *numbers* 8 and 9 still makes sense,

```
>>> int('11',8)
9
>>> int('0o11',8)
9
>>> oct(9)
'0o11'
```

In binary or base 2, the string `'111'` corresponds to the number 7,

$$1 \cdot 2^2 + 1 \cdot 2 + 1 = 1 \cdot 4 + 1 \cdot 2 + 1 = 7.$$

To indicate this, we also may prefix the string `'111'` with a `'0b'`, a zero followed by lowercase `'b'`,

```
>>> int('111',2)
7
>>> 0b111 == 7 == int('111',2) == int('0b111',2)
True
```

The closely related function `bin( )` takes an `int` and returns the binary *string* representation,

```
>>> bin(7)
'0b111'
>>> 7 == int(bin(7),2) == 0b111
True
>>> int('111',2) == bin(7)
False
```

The function `bin( )` is the *inverse* of the function `int( ,2)`: whatever the `int` `a`, we have

```
>>> int(bin(a),2) == a
True
```

Moreover, in binary representations, the digits are 0 and 1 only. There are no other digits,

```
>>> int('5',2)
Error: invalid literal for int( ) with base 2: '5'
```

but the number 5 still makes sense,

```
>>>  int('101',2)
5
>>>  int('0b101',2)
5
>>>  bin(5)
'0b101'
```

In hexadecimal (hex for short) or base 16, the string `'111'` corresponds to the number 273,

$$1 \cdot 16^2 + 1 \cdot 16 + 1 = 1 \cdot 256 + 1 \cdot 16 + 1 = 273.$$

To indicate this, we also may prefix the string `'111'` with a `'0x'`, a zero followed by lowercase `'x'`,

```
>>>  int('111',16)
273
>>>  0x111 == 273 == int('111',16) == int('0x111',16)
True
```

The closely related function `hex( )` takes an `int` and returns the hex or hexadecimal *string* representation,

```
>>>  hex(273)
'0x111'
>>>  273 == int(hex(273),16) == 0x111
True
>>>  int('111',16) == hex(273)
False
```

The function `hex( )` is the *inverse* of the function `int( ,16)`: whatever the `int` `a`, we have

```
>>>  int(hex(a),16) == a
True
```

Moreover, in hex or hexadecimal representations, the digits 0 through 9 are not enough, we also need digits a, b, c, d, e, f representing the other integers below 16,

```
>>>  int('a',16)
10
>>>  int('b',16)
11
>>>  int('c',16)
12
>>>  int('d',16)
13
>>>  int('e',16)
14
>>>  int('f',16)
```

```
15
```

Also one has

```
>>>  int('A',16)
10
>>>  int('B',16)
11
>>>  int('C',16)
12
>>>  int('D',16)
13
>>>  int('E',16)
14
>>>  int('F',16)
15
```

so one may use uppercase $A, \ldots, F$ and lowercase $a, \ldots, f$ interchangeably.

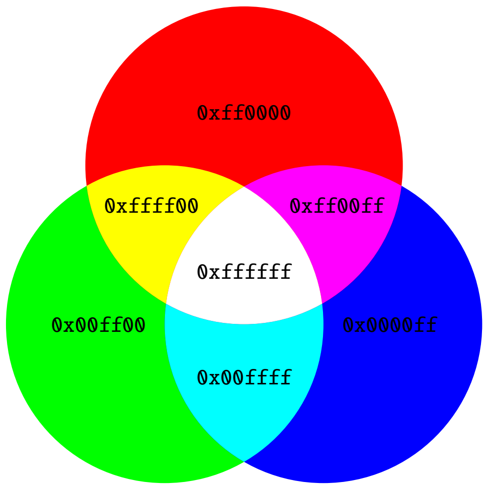

Fig. 1.2 HTML color codes.

A *bit* is a binary digit 0 or 1. An 8-bit number is a *byte*. Since $2^4 = 16$, a 1-digit hex is a 4-bit number,

```
>>>  (bin(0x0), bin(0x2), bin(0x5), bin(0xf))
('0b0', '0b10', '0b101', '0b1111')
>>>  (hex(0b0), hex(0b10), hex(0b101), hex(0b1111))
('0x0', '0x2', '0x5', '0xf')
```

Since a byte is an 8-bit number, a byte is a 2-digit hex. The smallest byte is `0x00`, corresponding to the number 0, and the largest byte is `0xff`, corresponding to the number 255. There are $2^8 = 16^2 = 256$ possible bytes.

HTML color codes are given by 3 bytes, or 6 hex digits, with each byte representing the intensity of red, green, and blue respectively, see Figure 1.2. Often the bytes in a multi-digit hex are separated by colons, so magenta is written `ff:00:ff`.

The first 128 bytes, 0 through 127, are the original *ASCII characters*. These are generated by

```
>>>  chr(10)
'\n'
>>>  chr(0x0a)
'\n'
>>>  chr(65)
'A'
>>>  chr(0x41)
'A'
```

If we stick to these bytes, the 8-th bit in each byte is 0, so these characters are 7-bit (Figure 1.3). If we include the remaining 128 characters, we obtain 8-bit *extended* ASCII characters, which include accented letters (such as é) and math symbols. *These* 8-bit characters are what we defined earlier as bytes. Usually the term *character* is broadened even further, up to 32 bits; these are *Unicode* characters. Unicode currently handles most written languages and still has room for even more. This includes left-to-right scripts like English and right-to-left scripts like Arabic. Chinese, Russian, Japanese, Urdu, and many other languages are also represented within Unicode.

Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII
00	*null*	10	*Ctrl* P	20	*space*	30	0	40	@	50	P	60	‘	70	p
01	*Ctrl* A	11	*Ctrl* Q	21	!	31	1	41	A	51	Q	61	a	71	q
02	*Ctrl* B	12	*Ctrl* R	22	"	32	2	42	B	52	R	62	b	72	r
03	*Ctrl* C	13	*Ctrl* S	23	#	33	3	43	C	53	S	63	c	73	s
04	*Ctrl* D	14	*Ctrl* T	24	$	34	4	44	D	54	T	64	d	74	t
05	*Ctrl* E	15	*Ctrl* U	25	%	35	5	45	E	55	U	65	e	75	u
06	*Ctrl* F	16	*Ctrl* V	26	&	36	6	46	F	56	V	66	f	76	v
07	*Ctrl* G	17	*Ctrl* W	27	’	37	7	47	G	57	W	67	g	77	w
08	*Ctrl* H	18	*Ctrl* X	28	(	38	8	48	H	58	X	68	h	78	x
09	*tab*	19	*Ctrl* Y	29	)	39	9	49	I	59	Y	69	i	79	y
0a	*newline*	1a	*Ctrl* Z	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	*Ctrl* K	1b	*Ctrl* [	2b	+	3b	;	4b	K	5b	[	6b	k	7b	{
0c	*Ctrl* L	1c	*Ctrl* \	2c	,	3c	<	4c	L	5c	\	6c	l	7c	|
0d	*Ctrl* M	1d	*Ctrl*]	2d	-	3d	=	4d	M	5d	]	6d	m	7d	}
0e	*Ctrl* N	1e	*Ctrl* ^	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	*Ctrl* O	1f	*Ctrl* _	2f	/	3f	?	4f	O	5f	_	6f	o	7f	*delete*

Fig. 1.3 The 7-bit ASCII characters.

A *file* is a string of characters, where the last character has ASCII code 0, the *null character*. For our purposes, we equate *character* with *byte*, so *a file is a string of bytes, terminated by the null byte.*

While this definition is natural, it wasn't until the coming of UNIX in 1970 that this became the norm. Prior to that, a file had structure, and was divided into various sections depending on the OS. UNIX taught us that at the OS level, it's better to think of files as having no structure, as just strings of bytes, and to have structure imposed on files by the particular application (e.g. document-processing, e-mail, printing, etc.) opening the file.

1.3 Arithmetic Operations

If a and b are numbers, the standard arithmetic operations $a + b$, $a - b$, ab yield numbers. These operations are valid whether a and b are stored as `int`s or `float`s. If a and b are `int`s, they may be represented in decimal, binary, octal, hexadecimal, and one may mix and match,

```
>>>  0b11 + 25
28
>>>  0x11 + 0b1111 - 23 * 0x111
-6247
>>>  0o666 - 732
-294
>>>  bin(0o666 - 732)
'-0b100100110'
```

Division a/b yields the quotient as a `float`, even if a and b are `int`s, so

```
>>>  a = 10
>>>  b = 2
>>>  c = 4
>>>  a/b
5.0
>>>  a/c
2.5
```

If you want the quotient to be an integer, use `a//b` instead

```
>>>  a = 10
>>>  b = 2
>>>  c = 4
>>>  a//b
5
>>>  a//c
2
>>>  a = 10.0
>>>  b = 3.0
>>>  a/b
3.3333333333333335
>>>  a//b
3.0
```

Note $a//b$ throws away the remainder, if there is one. This is *floor division*.

What if we want the remainder? This is given by `a%b` (“a mod b”)

```
>>>  a = 10
>>>  b = 2
>>>  c = 4
>>>  a%b
0
>>>  a%c
2
>>>  a = 0b111101
>>>  b = 0xf
>>>  (a, b, a%b)
(61, 15, 1)
>>>  a = 10.0
>>>  b = 3.0
>>>  a%b
1.0
```

`%` is the *modulus operator*. `(a, b, a%b)` is a `tuple`, which we’ll discuss later, and is here merely a device to print out `a`, `b`, and `a%b` in one step.

Powers are written `a**n`, so

```
>>>  a**7 == a*a*a*a*a*a*a
True
```

for any `int` a.

Strings may be added. The sum is the *concatenation* of the strings,

```
>>>  a = 'alpha'
>>>  b = 'beta'
>>>  a + b
'alphabeta'
>>>  b + a
'betaalpha'
>>>  a + ' ' + b
'alpha beta'
>>>  a + b == b + a
False
```

Thus $a + b \neq b + a$ for strings. Strings may be multiplied by `int`s,

```
>>>  a = 'alpha'
>>>  3*a
'alphaalphaalpha'
>>>  2*a
'alphaalpha'
>>>  0*a
''
>>>  (-1)*a
''
>>>  a*a
Error: can't multiply sequence by non-int of type 'str'
```

So the operations + and * have different meanings, depending on the types of the `object`s being added or multiplied. This is called *operator overloading*.

If `a` is a string, then `a[i]` is the character at the i-th index, with i starting from 0.

```
>>>  a = 'alpha'
>>>  a[0],a[1],a[2],a[3]
('a','l','p','h')
>>>  len(a)
5
>>>  a[5]
Error: string index out of range
>>>  a == a[0] + a[1] + a[2] + a[3] + a[4]
True
```

If `len(a)` is n, then the last character is `a[n-1]`, not `a[n]`. The last character may also be recovered as `a[-1]`, so

```
>>>  a[n-1] == a[-1]
True
```

A *slice* of a string `a` is a portion of `a`. For example, `a[2:6]` is the portion of the string `a` starting at index 2 and ending at index $6 - 1 = 5$, so

```
>>>  a = 'alpharomeo'
>>>  a[2:6]
'phar'
>>>  a[2:6] == a[2] + a[3] + a[4] + a[5]
True
```

To go to the end of the string, insert nothing after the colon,

```
>>>  a[2:]
'pharomeo'
```

To start from the beginning of the string, insert 0 or nothing before the colon,

```
>>>  a[:6]
'alphar'
```

Then the slice `a[:]` is the whole string, not just the same value, but the same `object`.

```
>>>  b = a[:]
>>> a is b
True
```

If a were a `list`, this last would be `False` (Chapter §2).

1.4 Bits and Bytes

Recall an integer is represented in binary as a sequence of bits, for example

$$\begin{aligned} 101011 &= 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 2^5 + 2^3 + 2^1 + 2^0 = 32 + 8 + 2 + 1 = 43. \end{aligned}$$

n-bit numbers may be represented by the *corners of the cube in n-dimensional space*. For example, all possible 3-bit numbers may be arranged on the corners of the cube in three dimensions (Figure 1.4). All possible bytes may be arranged on the corners of the cube in eight dimensions.

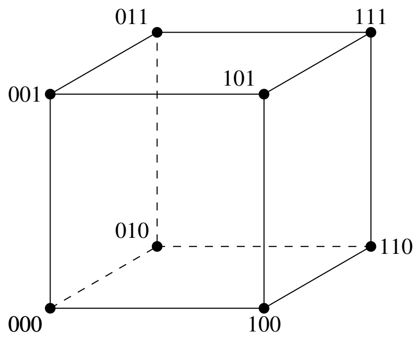

Fig. 1.4 Corners of the cube.

The IPv4 address of a computer on the internet is a 32-bit number, or equivalently a 4-byte number, or an 8-hex number. This is usually written $a.b.c.d$, where a, b, c, d are between 0 and 255 inclusive, such as 74.125.43.99, or $4a : 7d : 2b : 63$ as a 4 byte number.

A computer's hardware (ethernet or wifi) address is a 6-byte number. My laptop's hardware address is 1c:36:bb:2d:b7:5c; in decimal, it is 31021394147164.

Remember a computer's hardware address is *who it is*, versus its IPv4 address, which is *where it is*: If you turn a computer off, it still has a hardware address, but no IPv4.

Now ask how many n-bit numbers have exactly k digits equal to ones? For example, there are six 4-bit numbers with exactly two 1's; they are 1100, 1010, 1001, 0110, 0011, 0101. The general answer follows from Theorem 2.3.2 in Chapter 2,

Theorem 1.4.1 *The number of n-bit numbers with exactly k ones is*

$$\frac{n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1)}{1 \cdot 2 \cdot 3 \cdot \cdots \cdot k}.$$

Proof This is an immediate consequence of Theorem 2.3.2. □

For example,

$$\frac{4 \cdot 3}{1 \cdot 2} = 6,$$

as we saw above.

In all the exercises, don't just plug in the code into your console. First see if you can figure out the answer in your mind or with paper and pencil, then check your guess against the computer's result.

Exercises

Exercise 1.1 What do these inputs return and why?

```
>>>  int(int('23.4')) == int('23.4')
>>>  float(float('23')) == float('23')
>>>  bool('.0001')
>>>  bool(bool('.0001')) == bool('.0001')
>>>  str(str(23)) == str(23)
>>>  int(True)
>>>  int(False)
>>>  float(True)
>>>  float(False)
>>>  int('alpha')
>>>  int('23') == float('23')
```

Exercise 1.2 What are all the keywords in Python? How did you find them?

Exercise 1.3 `int`s are immutable `object`s, so how come we can convert the `int` `23` to a string via `str(23)`?

Exercise 1.4 Is an `object` part of the physical universe or not? Explain.

Exercise 1.5 `type(23)` returns `int`. `type(int)` returns `type`. `type(type)` returns what? Why?

Exercise 1.6 How many HTML color codes are there?

Exercise 1.7 Let `a = 23` and `b = 46`. What does

```
>>>  (a + a == b, a + a is b)
```

return? Why? (Compare with Exercise 2.14.)

Exercise 1.8 How many IPv4 internet addresses are there?

Exercise 1.9 Show that the hex representation of 74.125.43.99 is $4a : 7d : 2b : 63$.

Exercise 1.10 Let `a` be a `str`. Write code that returns `a` with enough leading spaces so that the right-most letter of `a` is 70 spaces from the left margin.

Exercise 1.11 How many possible hardware addresses can there be? What is *your* laptop's hardware wifi address in hex? in decimal? in binary?

Exercise 1.12 How many 10-bit numbers are there with exactly 3 ones? How many 10-bit numbers are there with the same number of zeros as ones?

Exercise 1.13 Let `(h1,m1,s1)` and `(h2,m2,s2)` be times, expressed in hours, minutes, and seconds. Insert parentheses in

```
h1 > h2 or h1 == h2 and m1 > m2 or m1 == m2 and s1 > s2
```

so it returns `True` if and only if time `(h2,m2,s2)` is chronologically before time `(h1,m1,s1)`. Test your answer!

Exercise 1.14 If `p` is a string containing a substring `a`, then `p.replace(a,b)` replaces every occurrence of the string `a` in `p` by the string `b`. Use `p.replace` to write the 129 byte number

```
p =  '''00:f1:63:a4:49:8c:bd:82:de:73:ca:fb:54:e1:7b:
   41:4f:14:6e:69:94:f3:c7:72:c7:69:ba:4a:ae:25:
   50:df:ce:c4:61:10:26:17:db:a4:fe:1c:4c:92:6c:
   c4:fb:16:d3:57:1e:4b:28:9f:b5:6d:2c:00:ec:2f:
   23:1f:a2:67:c4:d1:13:ad:b1:47:dc:79:51:b8:fe:
   39:41:11:bb:36:13:9d:61:58:e6:bd:02:1d:4b:ce:
   57:f5:32:7d:b6:9f:23:67:ff:2d:5e:51:dd:a8:50:
   44:a8:59:0b:9f:4d:e5:0c:15:bd:63:3e:77:2f:b2:
   c1:17:c1:f1:19:a0:e9:19:a5'''
```

in decimal.

Chapter 2
Binomial Theorem

2.1 Polynomials

A *polynomial* in a *variable* x is an expression of the form

$$p = 5x^3 - 17x + 1.$$

For symbolic or algebra manipulations with a variable x, we need to load the `sympy` module,

```
>>>  from sympy import *
```

Throughout this chapter, we assume this module is loaded. The module needs to be loaded only once at the beginning, after opening your notebook.

To carry out symbolic manipulations, we work with symbols

```
>>>  var('x')
```

x

Symbols may be assigned to variables,

```
>>>  a = var('x')
>>>  a
```

x

```
>>>  type(a)
sympy.core.symbol.Symbol
```

so `Symbol` is a type like `int`, but belonging to the submodule `symbol` of the `core` submodule of the module `sympy`.

For clarity, we will use the same name for symbols and their corresponding variables. For example we always assign the symbol x to the variable `x`. Here is an example where we first assign a symbol to `x`, then assign a number to `x`,

```
>>>  x = var('x')
>>>  2*x + 3
```

$2x + 3$

```
>>>  x = 10
>>>  2*x + 3
```

23

Now we work with polynomials.

```
>>>  x = var('x')
>>>  p = 5*x**3 - 17*x + 1
>>>  p
```

$5x^3 - 17x + 1$

```
>>>  q = x + 5
>>>  p + q
```

$5x^3 - 16x + 6$

```
>>>  p*q
```

$(x + 5)(5x^3 - 17x + 1)$

```
>>>  expand(p*q)
```

$5x^4 + 25x^3 - 17x^2 - 84x + 5$

```
>>>  (p*q).as_poly(x)
```

$\text{Poly}(5x^4 + 25x^3 - 17x^2 - 84x + 5, x, \textit{domain} = \mathbf{Z})$

```
>>>  degree(p)
3
>>>  degree(q)
1
>>>  degree(p + q)
3
>>>  degree(p*q)
4
>>>  (p*q).as_poly(x).coeffs()
[5,25,-17,-84,5]
```

The *degree* of a polynomial p is the highest power of x in p.

One can have polynomials in two or more variables

```
>>>  (a,b,t) = var('a b t')
>>>  p = a**3*b - 5*a*b**2 + t*b + t**5*a*b + t*a
>>>  degree(p,a)
3
>>>  degree(p,b)
2
>>>  degree(p,t)
5
>>>  p
```

$a^3b - 5ab^2 + abt^5 + at + bt$

Here p is considered as a polynomial in three variables a, b, t. One can also consider p as a polynomial in t with coefficients containing a's and b's,

```
>>>  p.as_poly(t)
```

Poly($abt^5 + (a + b)t + a^3b - 5ab^2, t, domain = \mathbf{Z}[a, b]$)

```
>>>  p.as_poly(t).coeffs()
[a*b, a + b, a**3*b - 5*a*b**2]
```

or as a polynomial in a with coefficients containing t's and b's,

```
>>>  p.as_poly(a)
```

Poly($ba^3 + (-5b^2 + bt^5 + t)a + bt, a, domain = \mathbf{Z}[t, b]$)

```
>>>  p.as_poly(a).coeffs()
[b, -5*b**2 + b*t**5 + t, b*t]
```

or as a polynomial in b with coefficients containing t's and a's,

```
>>>  p.as_poly(b)
```

Poly($-5ab^2 + (a^3 + at^5 + t)b + at, a, domain = \mathbf{Z}[t, a]$)

```
>>>  p.as_poly(b).coeffs()
[-5a, a**3 + a*t**5 + t, a*t]
```

2.2 Lists

To write code that generates Pascal's triangle (§2.4), we introduce a new type, the `list`. A `list` is a sequence of values, separated by commas, and enclosed in brackets,

```
>>>  a = ['a',1,3,1,'alpha']
>>>  id(a)
4508915104
>>>  type(a)
list
>>>  len(a)
5
```

Be careful: A `list` is enclosed in brackets `[ ]`, whereas (later) a `dict` and a `set` are enclosed in braces `{ }`, and a `tuple` is enclosed in parentheses `( )`. The entries in the list are accessed as `a[0]`, `a[1]`, `a[2]`, and so on. Thus for the above list,

```
>>>  a[0]
'a'
>>>  a[1]
```

```
1
>>>  a[2]
3
>>>  a[3]
1
>>>  a[4]
'alpha'
>>>  a[5]
Error: list index out of range
```

The list with no entries is the *empty list*,

```
>>>  e = [ ]
>>>len(e)
0
```

Objects may be *appended* to a list

```
>>>  a.append('new')
>>>  a
['a',1,3,1,'alpha','new']
>>>  a[5]
'new'
>>>  len(a)
6
>>>  id(a)
4508915104
```

Even though `len(a)` is 6, `len([a])` is 1 since `[a]` is a list with one entry. Appending modifies the list but returns nothing, nothing was printed out. A list `a` is mutable: modifying `a` did not change its `id`. The `append` method takes only one argument,

```
>>>  a.append(23,'beta')
Error: append() takes exactly one argument
```

but

```
>>>  b = [23,'beta']
>>>  a.append(b)
```

works,

```
>>>  a
['a',1,3,1,'alpha','new',[23,'beta']]
```

since `b` is one `object`, a `list`. To append several `object`s, use *list addition*. As for strings, list addition is concatenation,

```
>>>  a + b
['a',1,3,1,'alpha','new',[23,'beta'],23,'beta']
>>>  len(a+b)
9
```

A *range* is a sequence `range(1,n)` of integers. Here `range(1,n)` starts at 1 and stops at $n-1$, just below n: $1, 2, 3, \ldots, n-1$.

```
>>>  type(range(1,10))
range
```

A `range` may be converted to a `list`,

```
>>>  r = range(1,10)
>>>  list(r)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The *Fibonacci sequence* is

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

where each term in the sequence in the sum of the previous two terms. Let `fib1(n)` denote the n-th term in the sequence, starting from $n = 0$, so `fib1(0) == 1`, `fib1(1) == 1`, `fib1(2) == 2`, and so on. Here is code generating the n-th Fibonacci number, as a *function* `fib1` having a single *parameter* `n`,

```
0>>  def fib1(n):
1>>    if n == 0 or n == 1:
2>>      return 1
1>>    elif n == 2:
2>>      return 2
1>>    else:
2>>      a = 2
2>>      b = 1
2>>      for k in range(2,n):
3>>        a,b = a+b,a
2>>      return a
```

`fib1` is a `function` (§1.1)

```
>>>  type(fib1)
function
```

You give the function `fib1` an *argument* `n`, and it returns an `int fib1(n)`.

The *header* of the function is the statement `def fib1(n):`. A header is always ended by a colon `:`. Below the header are ten *indented* lines of code that are the *body* of the function. The last statement in the body of the function `fib1` is `return a`. The body is a *block* of code. The three statements

```
if n == 0 or n == 1:
elif n == 2:
else:
```

are *conditionals*. Each conditional is a header, followed by its own block of code. In this case, these three headers are followed by three blocks of one, one, and five lines of code respectively.

If the first conditional returns `True`, the first body is executed. Otherwise, if the first conditional returns `False` and the second conditional returns `True`, the second body is executed. Otherwise, if the first and second conditionals both return `False`, the third body is executed. So the logic flow here is *if*, then *else if*, then *else*.

The statement `for k in range(2,n):` is the header of a *for loop*. Below it is one indented line of code that is the body of the `for` loop. The body of the `for` loop, `a,b = a+b,a`, is executed *repeatedly,* once for each `int` `k` in `range(2,n)`. Here the body assigns the `tuple` `a+b,a` to the tuple `a,b`. Tuples are discussed in §3.5. For now, the effect is to assign a to b, and $a + b$ to a.

Once out of the range, the code `return`s `a`. To repeat, the last statement in the body of `fib1` is not in the body of the `for` loop, it executes only after the loop is finished.

The *increment* of an `int` `k` is `k+1`. Inside the `for` loop, the body executes with `k` at the start of `range(1,n)`, so with `k=1`, then `k` is incremented and the body is executed again with `k=2`, and so on, until `k=n`, at which point the body does not execute and the `return` statement is executed. Thus the last `k` for which the body executes is `k=n-1`.

Be careful, Python is very picky about indentation. All lines in a body of code have to be pushed to the right by the same amount, e.g. either 1 tab, or 2 spaces, or 4 spaces, etc. An *indent* is the amount of whitespace a body of code is pushed to the right. *Whitespace* is any combination of spaces and tabs.

How does Python know when a body of code ends? When the lines are no longer indented. Because of this, we sometimes replace the prompt >>> with 1>>, 2>>, etc. where 1, 2, etc. indicate the number of indents in the line. If there are no indents, we write >>> or 0>>.

The `elif` block of code is unnecessary in `fib1`. This is because `range(2,n)` is empty when `n==2`, so the for loop doesn't kick in, and `fib1(2)` returns 2. Removing this block, the code becomes

```
0>>  def fib1(n):
1>>     if n == 0 or n == 1:
2>>        return 1
1>>     else:
2>>        a = 2
2>>        b = 1
2>>        for k in range(2,n):
3>>           a,b = a+b,a
2>>        return a
```

In fact the `if`/`else` logic is unnecessary as well, if the range starts from 1,

```
0>>  def fib1(n):
1>>     a = 1
1>>     b = 1
1>>     for k in range(1,n):
2>>        a,b = a+b,a
1>>     return a
```

The *recursive* approach to generating the Fibonacci sequence is to note that fib1(n) is the sum of fib1(n-1) and fib1(n-2), and to let the code reflect this directly,

```
0>>  def fib2(n):
1>>     if n == 0 or n == 1:
2>>        return 1
1>>     else:
2>>        return fib2(n-1) + fib2(n-2)
```

This code is recursive in the sense that the function *calls itself* during its execution. This function may be written

```
0>>  def fib2(n):
1>>     if n == 0 or n == 1:
2>>        return 1
1>>     return fib2(n-1) + fib2(n-2)
```

This code has the same effect, because the second return statement does not execute if the first return statement executes: the second return statement is outside the block of code following the if header.

While clear and compelling, fib2 recalculates terms which it already knows, so it takes a lot longer to execute. Compare the running times of fib1(n) and fib2(n), you will see a difference as soon as $n = 35$. Nevertheless, there are many situations where recursive coding is just the right technique.

A simple technique to avoid recalculating the same items is to keep a record of what has been calculated, then look up the record each time before calculating. This technique is called *memoization*.

```
0>>  memo = { 0:1, 1:1 }
0>>  def fib3(n):
1>>     if n in memo:
2>>        return memo[n]
1>>     else:
2>>        memo[n] = fib3(n-1) + fib3(n-2)
2>>        return memo[n]
```

Here memo is a dict, discussed in §3.4. The dict memo assigns a value memo[n] to each n. The first statement has a two-fold purpose: It states memo is a dict, and assigns memo[0]=1 and memo[1]=1. For now, just think of memo[n] as a record of the n-th Fibonacci number. If this record exists, return it; otherwise, compute the sum of the previous two Fibonacci numbers, and create a new record. Compare the running times of fib1(n) and fib3(n). fib3 is almost as fast as fib1, but you won't see a difference till n is in the thousands.

Again, this code may be written

```
0>>  memo = { 0:1, 1:1 }
0>>  def fib3(n):
1>>     if n in memo:
```

```
2>>         return memo[n]
1>>      memo[n] = fib3(n-1) + fib3(n-2)
1>>      return memo[n]
```

or, even shorter,

```
0>>   memo = { 0:1, 1:1 }
0>>   def fib3(n):
1>>      if n not in memo:
2>>         memo[n] = fib3(n-1) + fib3(n-2)
1>>      return memo[n]
```

Notice `memo` is defined, or *initialized*, *outside* the function `fib3`. This has to be so, to have access to pre-computed values independent of which particular iteration of `fib3` is running. If `memo` were initialized *inside* `fib3`, it would be re-computed each time `fib3` runs, defeating its purpose.

2.3 Binomial Theorem

Let x and a be two variables. Multiplying out, since $ax = xa$, we get

$$(x+a)^2 = (x+a)(x+a) = x^2 + xa + ax + a^2 = x^2 + 2ax + a^2. \tag{2.3.1}$$

Similarly,

$$\begin{aligned}(x+a)^3 &= (x+a)(x+a)^2 = (x+a)(x^2+2ax+a^2)\\ &= x^3 + 2x^2a + xa^2 + ax^2 + 2axa + a^3\\ &= x^3 + 3ax^2 + 3a^2x + a^3.\end{aligned} \tag{2.3.2}$$

A *binomial of degree* n is a polynomial of the form $(x+a)^n$. Thus

$$\begin{aligned}(x+a)^2 &= x^2 + 2ax + a^2\\ (x+a)^3 &= x^3 + 3ax^2 + 3a^2x + a^3\\ (x+a)^4 &= x^4 + 4ax^3 + 6a^2x^2 + 4a^3x + a^4\\ &\text{etc}\end{aligned} \tag{2.3.3}$$

In Python,

```
>>>   (x, a) = var('x a')
>>>   p = (x + a)**4
>>>   p
```

$(x+a)^4$

```
>>>   expand(p)
```

$a^4 + 4a^3x + 6a^2x^2 + 4ax^3 + x^4$

```
>>> p.as_poly(x)
```

Poly $\left(x^4 + 4x^3a + 6x^2a^2 + 4xa^3 + a^4, x, domain = \mathbf{Z}[a]\right)$

```
>>> p.as_poly(x).coeffs()
[1, 4*a, 6*a**2, 4*a**3, a**4]
>>> p.as_poly(a).coeffs()
[1, 4*x, 6*x**2, 4*x**3, x**4]
```

There is a pattern in (2.3.3). On each line, the powers of x decrease, while the powers of a increase. We want to find the pattern for the coefficients $1, 2, 1$, and $1, 3, 3, 1$, and $1, 4, 6, 4, 1$, and so on. These coefficients are the binomial coefficients. More generally, the *binomial coefficient*

$$\binom{n}{k}, \qquad 0 \le k \le n,$$

is defined to be *the coefficient of $x^{n-k}a^k$ when you multiply out* $(x+a)^n$. For example, since

$$(x+a)^2 = x^2 + 2ax + a^2 \qquad \text{and} \qquad (x+a)^3 = x^3 + 3ax^2 + 3a^2x + a^3,$$

we have $\binom{2}{0} = 1$, $\binom{2}{1} = 2$, $\binom{2}{2} = 1$, and $\binom{3}{0} = 1$, $\binom{3}{1} = 3$, $\binom{3}{2} = 3$, $\binom{3}{3} = 1$.

The goal of the next section is to obtain a formula for $\binom{n}{k}$. The existence of such a formula for $\binom{n}{k}$ is the *binomial theorem*, due to Newton.

By the very definition of the binomial coefficients, we have

$$\begin{aligned}(x+a)^n = \binom{n}{0}x^n + \binom{n}{1}x^{n-1}a + \binom{n}{2}x^{n-2}a^2 \\ + \dots + \binom{n}{n-1}xa^{n-1} + \binom{n}{n}a^n.\end{aligned} \tag{2.3.4}$$

When $(x+a)^n$ is multiplied out, the coefficients of both x^n and a^n are 1, so $\binom{n}{0} = 1$ and $\binom{n}{n} = 1$.

Using summation notation,[1] this may be written as

Theorem 2.3.1 (Newton's Binomial Theorem) *For $n \ge 0$,*

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} a^k x^{n-k}.$$

The binomial coefficient $\binom{n}{k}$ is also called "n-choose-k" because of

Theorem 2.3.2 *The number of ways you can choose k objects from n objects is $\binom{n}{k}$.*

[1] $\sum_{k=0}^{n} a_k$ is short for the sum $a_0 + a_1 + \cdots + a_n$.

Proof When you multiply out the product

$$(x+a)^n = (x+a)(x+a)\dots(x+a),$$

you are multiplying x's and a's. For a fixed k, the terms containing x^k correspond to choosing k x's and $n-k$ a's in this product, that is choosing k objects from n objects. Since the coefficient of x^k is then the number of these terms, it follows that the number of these terms is $\binom{n}{k}$. □

Since

$$(x+a)^n = (a+x)^n,$$

the coefficient of $x^{n-k}a^k$ when you multiply out $(x+a)^n$ equals the coefficient of $a^{n-k}x^k$ when you multiply out $(x+a)^n$, so we have

$$\binom{n}{k} = \binom{n}{n-k}, \qquad 0 \le k \le n,$$

so the binomial coefficients remain unchanged when k is replaced by $n-k$.

2.4 Pascal's Triangle

The key step in finding a formula for $\binom{n}{k}$ is to notice

$$(x+a)^{n+1} = (x+a)(x+a)^n.$$

Let's work this out when $n=3$. Then the left side is $(x+a)^4$. Insert the right side of (2.3.4) into the right side of this last equation. You get

$$\begin{aligned}
&\binom{4}{0}x^4 + \binom{4}{1}x^3a + \binom{4}{2}x^2a^2 + \binom{4}{3}xa^3 + \binom{4}{4}a^4 \\
&= (x+a)\left(\binom{3}{0}x^3 + \binom{3}{1}x^2a + \binom{3}{2}xa^2 + \binom{3}{3}a^3\right) \\
&= \binom{3}{0}x^4 + \binom{3}{1}x^3a + \binom{3}{2}x^2a^2 + \binom{3}{3}xa^3 \\
&\qquad + \binom{3}{0}x^3a + \binom{3}{1}x^2a^2 + \binom{3}{2}xa^3 + \binom{3}{3}a^4 \\
&= \binom{3}{0}x^4 + \left(\binom{3}{1} + \binom{3}{0}\right)x^3a + \left(\binom{3}{2} + \binom{3}{1}\right)x^2a^2 \\
&\qquad + \left(\binom{3}{3} + \binom{3}{2}\right)xa^3 + \binom{3}{3}a^4.
\end{aligned}$$

Equating corresponding coefficients of x, we get,

$$\binom{4}{1} = \binom{3}{1} + \binom{3}{0}, \qquad \binom{4}{2} = \binom{3}{2} + \binom{3}{1}, \qquad \binom{4}{3} = \binom{3}{3} + \binom{3}{2}.$$

In general, the same exact calculation establishes

Theorem 2.4.1

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}, \qquad 1 \le k \le n. \tag{2.4.1}$$

This allows us to build *Pascal's triangle* (Figure 2.1), where, apart from the ones on either end, each term ("the child") in a given row is the sum of the two terms ("the parents") located directly above in the previous row.

$$\begin{array}{lccccccccccccccccccccc}
n=0: &&&&&&&&&&& 1 \\
n=1: &&&&&&&&&& 1 && 1 \\
n=2: &&&&&&&&& 1 && 2 && 1 \\
n=3: &&&&&&&& 1 && 3 && 3 && 1 \\
n=4: &&&&&&& 1 && 4 && 6 && 4 && 1 \\
n=5: &&&&&& 1 && 5 && 10 && 10 && 5 && 1 \\
n=6: &&&&& 1 && 6 && 15 && 20 && 15 && 6 && 1 \\
n=7: &&&& 1 && 7 && 21 && 35 && 35 && 21 && 7 && 1 \\
n=8: &&& 1 && 8 && 28 && 56 && 70 && 56 && 28 && 8 && 1 \\
n=9: && 1 && 9 && 36 && 84 && 126 && 126 && 84 && 36 && 9 && 1 \\
n=10: & 1 && 10 && 45 && 120 && 210 && 252 && 210 && 120 && 45 && 10 && 1
\end{array}$$

Fig. 2.1 Pascal's triangle.

In Pascal's triangle, the very top row has one number in it: This is the *zeroth row* corresponding to $n = 0$ and the binomial expansion of $(x + a)^0$. The *first row* corresponds to $n = 1$ and has the numbers 1 and 1 and corresponds to the binomial expansion of $(x + a)^1$. We say the *zeroth entry* ($k = 0$) in the *first row* ($n = 1$) is 1. Similarly, the *zeroth entry* ($k = 0$) in the *second row* ($n = 2$) is 1, and the *second entry* ($k = 2$) in the *second row* ($n = 2$) is 1. The *second entry* ($k = 2$) in the *fourth row* ($n = 4$) is 6. In general, the entries are counted starting from $k = 0$, and end with $k = n$, so there are $n + 1$ entries in row n. With this understood, the k-th entry in the n-th row is the binomial coefficient $\binom{n}{k}$. So 10-choose-2 is $\binom{10}{2} = 45$.

Notice $\binom{n}{1} = \binom{n}{n-1} = n$, and $\binom{n}{0} = \binom{n}{n} = 1$. Insert $x = 1$ and $a = 1$ in the binomial theorem to get

$$2^n = \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n-1} + \binom{n}{n}. \tag{2.4.2}$$

You conclude *the sum of the binomial coefficients along the n-th row of Pascal's triangle is* 2^n (remember n starts from 0).

Now insert $x = 1$ and $a = -1$. You get

$$0 = \binom{n}{0} - \binom{n}{1} + \binom{n}{2} - \cdots \pm \binom{n}{n-1} \pm \binom{n}{n}.$$

Hence: *the alternating*[2] *sum of the binomial coefficients along the n-th row of Pascal's triangle is zero*.

Here is the code generating the Pascal triangle rows, as a function `next_row` having a single parameter `parents`,

```
0>>  def next_row(parents):
1>>     children = []
1>>     n = len(parents)
1>>     for k in range(1,n):
2>>        new_entry = parents[k] + parents[k-1]
2>>        children.append(new_entry)
1>>     return [1] + children + [1]
```

`next_row` is a `function` (§1.1)

```
>>>  type(next_row)
function
```

that returns a list: You give `next_row` an argument, which in this case is a list `a`, and it returns the list `next_row(a)`,

```
>>>  a = [1,11,7]
>>>  type(a)
list
>>>  next_row(a)
[1, 12,18,1]
>>>  type(next_row(a))
list
```

Now we generate the rows of Pascal's triangle,

```
>>>  next_row([1])
[1,1]
>>>  next_row([1,1])
[1,2,1]
>>>  next_row([1,2,1])
[1,3,3,1]
>>>  next_row([1,3,3,1])
[1,4,6,4,1]
```

[2] *Alternating* means the plus-minus pattern $+ - + - + - \dots$.

Let's look at the code some more. The header of the function is the statement `def next_row(parents):`. Below it are six indented lines of code that are the body of the function. Below the `for` header are two indented lines of code that are the body of the `for` loop. The body of the `for` loop is executed repeatedly, once for each `int` `k` in `range(1,n)`, where `n` is the length of the list `parents`. Once out of the range, the code `return`s the children's row, with the ones added on either side.

Here is a formula for $\binom{n}{k}$:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{1 \cdot 2 \cdot \dots \cdot k}, \qquad 1 \le k \le n, \tag{2.4.3}$$

so

$$\binom{7}{3} = \frac{7 \cdot 6 \cdot 5}{1 \cdot 2 \cdot 3} = 35 = \binom{7}{4} \qquad \text{and} \qquad \binom{10}{2} = \frac{10 \cdot 9}{1 \cdot 2} = 45 = \binom{10}{8}.$$

The formula (2.4.3) is easy to remember: There are k terms in the numerator as well as the denominator, the denominator starts at 1 and increases, and the numerator starts at n and decreases.

We use the parent-child relationship (2.4.1) to establish (2.4.3). We do this by applying *induction*. The idea of induction is as follows, and is based on the fact that (2.4.3) *is a formula that depends on n.*

Let $P(n)$ be any formula that depends on n. If

1. $P(1)$ is valid, (the *base* step)
2. $P(n+1)$ is valid whenever $P(n)$ is valid, (the *inductive* step)

then $P(n)$ is valid for $1, 2 = 1+1, 3 = 2+1$, and so on, so $P(n)$ is valid for all positive integers $n \ge 1$. This is discussed further in §4.7. Here is our first use of induction.

Theorem 2.4.2 (2.4.3) *is valid for all* $n \ge 1$.

Proof (2.4.3) is valid when $n = 1$, because $\binom{1}{1} = 1$. This establishes the base step. Now assume (2.4.3) is valid. We have to show (2.4.3) remains valid if we replace n by its increment $n+1$, so we have to show

$$\binom{n+1}{k} = \frac{(n+1)(n+1-1) \cdot \dots \cdot (n+1-k+1)}{1 \cdot 2 \cdot \dots \cdot (k-1) \cdot k}, \tag{2.4.4}$$

for $1 \le k \le n+1$.

Since $\binom{n+1}{n+1} = 1$ and the right side of (2.4.4) when $k = n+1$ is 1, (2.4.4) is valid for $k = n+1$.

Since (2.4.3) is valid when $k = 0$ and $k = 1$, we have $\binom{n}{0} = 1$ and $\binom{n}{1} = n$. By (2.4.1),

$$\binom{n+1}{1} = \binom{n}{1} + \binom{n}{0} = n+1,$$

which is the right side of (2.4.4) when $k = 1$. Thus (2.4.4) is valid for $k = 1$.

The remaining cases of (2.4.4) to be checked are $2 \le k \le n$. By assumption, for $2 \le k \le n$,

$$\binom{n}{k-1} = \frac{n(n-1)\cdot\dots\cdot(n-k+2)}{1\cdot 2\cdot\dots\cdot(k-1)}$$

and

$$\binom{n}{k} = \frac{n(n-1)\cdot\dots\cdot(n-k+1)}{1\cdot 2\cdot\dots\cdot k}.$$

By (2.4.1), we get

$$\begin{aligned}\binom{n+1}{k} &= \frac{n(n-1)\cdot\dots\cdot(n-k+1)}{1\cdot 2\cdot\dots\cdot k} + \frac{n(n-1)\cdot\dots\cdot(n-k+2)}{1\cdot 2\cdot\dots\cdot(k-1)}\\ &= \frac{n(n-1)\cdot\dots\cdot(n-k+1)}{1\cdot 2\cdot\dots\cdot(k-1)}\cdot\left(\frac{1}{k}+\frac{1}{n-k+1}\right)\\ &= \frac{n(n-1)\cdot\dots\cdot(n-k+2)\cdot(n-k+1)}{1\cdot 2\cdot\dots\cdot(k-1)}\cdot\frac{n+1}{(n-k+1)k}\\ &= \frac{(n+1)(n+1-1)\cdot(n+1-2)\cdot\dots\cdot(n+1-k+1)}{1\cdot 2\cdot\dots\cdot(k-1)\cdot k}.\end{aligned}$$

But this is the right side of (2.4.4), thus (2.4.4) is valid, establishing the inductive step. By induction, (2.4.3) is a valid formula for all $n \geq 1$. □

Now we express the binomial coefficients in terms of factorials. Given a positive integer n, *n-factorial* is the product

$$n! = n\cdot(n-1)\cdot(n-2)\cdot\dots\cdot 2\cdot 1.$$

So $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, and so on. As a function,

```
0>>  def fact(n):
1>>     if n == 1:
2>>        return 1
1>>     else:
2>>        return n * fact(n-1)
```

This function is *recursive*, it calls itself within its body.

```
>>>  fact(10)
3628800
```

Here the header `def fact(n):` has a body consisting of four lines. In this body, there are two headers, `if n == 1:` and `else:`. To each of these headers corresponds a body, each consisting of one line of code. The `if` header contains a *condition* `n == 1`. If the condition is true, the first body is executed; if not, the second body is executed.

Thus, to compute `fact(10)`, the flow is as follows: Since `n = 10`, the condition `n == 1` returns `False`, leading to the execution of the second body, which involves `fact(9)`. Thus computing `fact(10)` forces the computing of `fact(9)`. To compute `fact(9)`, `n = 9`, so the condition `n == 1` returns `False`, leading to the execution of the second body, which involves `fact(8)`. Thus computing `fact(9)` forces the computing of `fact(8)`. To compute `fact(8)`, `n = 8`, so the condition `n == 1` returns `False`, leading to the execution of the second body, which involves `fact(7)`. Thus computing `fact(8)` forces the computing of `fact(7)`. The flow continues like this, down to `n = 1`. When `n = 1`, the condition `n == 1` returns `True`, leading to the execution of the first body, which returns 1.

Once `fact(1)` is returned, it is multiplied by 2, and the result $2 \cdot 1$ is returned as `fact(2)`, which is then multiplied by 3, and the result $3 \cdot 2 \cdot 1$ is returned as `fact(3)`, which is then multiplied by 4, and the result $4 \cdot 3 \cdot 2 \cdot 1$ is returned as `fact(4)`. The flow continues like this, until `fact(10)` returns $10!$.

By a slight modification of the code, the same amount of work yields all the factorials $1!, 2!, 3!, \dots, n!$,

```
0>>  def fact2(n):
1>>     if n == 1:
2>>        return [1]
1>>     previous = fact2(n-1)
1>>     return previous.append(n * previous[-1])
```

Then

```
>>>  fact2(10)
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Above we defined $n!$ for $n \geq 1$. If we complete the definition by setting $0! = 1$, then the binomial coefficient formula (2.4.3) may be rewritten

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \qquad 0 \leq k \leq n,$$

as can be verified by cancellation.

Here is code that returns the entire Pascal's triangle

```
0>>  def pascal():
1>>     current = [1]
1>>     while True:
2>>        new = next_row(current)
2>>        yield new
2>>        current = new
```

Here the function header `def pascal():` is followed by an indented body of 5 lines, and the *while loop* header `while True:` is followed by an indented body of 3 lines. The body of the `while` loop executes repeatedly until the condition in the header returns `False`. In this case, the condition is `True`, hence never returns `False`, so the body executes repeatedly forever, if it were not for `yield`, which pauses the function after each execution.

Run it as follows

```
>>>  p = pascal()
>>>  next(p)
```

```
[1,1]
>>>  next(p)
[1,2,1]
>>>  next(p)
[1,3,3,1]
>>>  next(p)
[1,4,6,4,1]
```

The keyword `yield` is discussed in §3.2.

2.5 Elementary Symmetric Polynomials

Recall $(x+a)^n$ is short for

$$(x+a)^n = (x+a)(x+a)\dots(x+a),$$

the product with n factors. We generalize (2.3.4) when $(x+a)^n$ is replaced by the product

$$(x-t_1a)(x-t_2a)\dots(x-t_na). \tag{2.5.1}$$

We also replace the $+$'s by $-$'s because that's what we'll need later. For example, when $n=2$,

$$(x-t_1a)(x-t_2a) = x^2-(t_1+t_2)ax+t_1t_2a^2 = x^2-p_1ax+p_2a^2, \tag{2.5.2}$$

where

$$p_1 = t_1+t_2, \qquad p_2 = t_1t_2.$$

Notice when $t_1=1$ and $t_2=1$, (2.5.2) reduces to (2.3.1). In Python,

```
>>>  (a,x,t1,t2) = var('a x t1 t2')
>>>  p  = (x - t1*a) * (x - t2*a)
>>>  expand(p)
```

$x^2-axt_1-axt_2+a^2t_1t_2$

```
>>>  p.as_poly(x)
```

$x^2-(at_1+at_2)x+a^2t_1t_2$

Now let's look at $n=3$,

$$\begin{aligned}&(x-t_1a)(x-t_2a)(x-t_3a)\\ &\quad= x^3-(t_1+t_2+t_3)x^2a+(t_1t_2+t_2t_3+t_3t_1)xa^2-t_1t_2t_3a^3\\ &\quad= x^3-p_1x^2a+p_2xa^2-p_3a^3,\end{aligned} \tag{2.5.3}$$

where

$$p_1 = t_1+t_2+t_3, \qquad p_2 = t_1t_2+t_2t_3+t_3t_1, \qquad p_3 = t_1t_2t_3.$$

Notice when $t_1 = 1$ and $t_2 = 1$ and $t_3 = 1$, (2.5.3) reduces to (2.3.2). In Python, this is returned by the code

```
>>>  (a,x,t1,t2,t3) = var('a x t1 t2 t3')
>>>  p  = (x - t1*a) * (x - t2*a) * (x - t3*a)
>>>  p.as_poly(x)
```

In general, the product (2.5.1) expands to[3]

$$\prod_{i=1}^{n}(x - t_i a) = p_0 x^n - p_1 x^{n-1} a + p_2 x^{n-2} a^2 - \cdots \pm p_{n-1} x a^{n-1} \mp p_n a^n, \qquad (2.5.4)$$

where $p_0 = 1$ and the

$$p_k = p_k(t_1, \dots, t_n), \qquad 1 \le k \le n,$$

are polynomials in $t_1, \dots, t_n$, the *elementary symmetric polynomials*,

$$p_1 = \sum_i t_i, \quad p_2 = \sum_{i<j} t_i t_j, \quad p_3 = \sum_{i<j<k} t_i t_j t_k, \quad \dots,$$

and so on. When $t_1 = t_2 = \cdots = t_n = 1$, (2.5.4) reduces to the binomial theorem, so

$$p_k(1, 1, \dots, 1) = \binom{n}{k}, \qquad 0 \le k \le n.$$

Moreover, replacing x and a by 1, (2.5.4) reduces to

$$\prod_{i=1}^{n}(1 - t_i) = 1 - p_1 + p_2 - p_3 + \cdots \pm p_n. \qquad (2.5.5)$$

If also $t_1 = t_2 = \cdots = t_n = 1$, this last equation says the the alternating sum of the binomial coefficients along the n-th row of Pascal's triangle is zero, which we saw in §2.4.

2.6 q-Binomial Theorem

This section follows [7].

Let q and x be variables. The *q-derivative* of a function $f(x)$ is

$$D_q f(x) = \frac{f(qx) - f(x)}{qx - x}.$$

For example, if $f(x) = 1$,

[3] The symbol $\prod_{i=1}^{n} a_i$ is short for the product $a_1 a_2 \dots a_n$.

$$D_q 1 = \frac{1-1}{qx-x} = 0,$$

if $f(x) = x$,

$$D_q x = \frac{qx-x}{qx-x} = 1,$$

and if $f(x) = x^2$,

$$D_q x^2 = \frac{(qx)^2 - x^2}{qx - x} = \frac{q^2-1}{q-1}\frac{x^2}{x} = (q+1)x.$$

Define

$$[n]_q = \frac{q^n-1}{q-1} = \frac{1-q^n}{1-q}.$$

Then $[1]_q = 1$ and we have

$$D_q 1 = 0, \qquad D_q x = 1, \qquad D_q x^2 = [2]_q x.$$

We call $[n]_q$ " q-n", so $[2]_q$ is q-two, $[5]_q$ is q-five, etc.

Theorem 2.6.1 *For $n \geq 0$, $D_q x^n = [n]_q x^{n-1}$.*

Proof

$$D_q x^n = \frac{(qx)^n - x^n}{qx-x} = \frac{q^n-1}{q-1}x^{n-1} = [n]_q x^{n-1}.$$

By multiplying,

$$q^n - 1 = (q-1)(q^{n-1} + q^{n-2} + \cdots + q + 1), \tag{2.6.1}$$

so

$$[n]_q = \frac{q^n-1}{q-1} = q^{n-1} + q^{n-2} + \cdots + q + 1 \tag{2.6.2}$$

is a polynomial in q with $[n]_1 = n$. Thus for q near 1, $[n]_q$ is near n. For example,

$$[1]_q = 1, \qquad [2]_q = 1+q, \qquad [3]_q = 1+q+q^2, \qquad [4]_q = 1+q+q^2+q^3.$$

```
>>>  q = var('q')
>>>
0>>  def _q_(n):
1>>     return cancel((q**n - 1) / (q-1))
>>>
>>>  type(_q_)
function
>>>  _q_(5)
```

$q^4 + q^3 + q^2 + q + 1$

The q-derivative D_q is *linear*: If f and g are polynomials,

$$D_q(f+g) = D_q f + D_q g.$$

In this section we derive the binomial theorem in the q-setting. To this end, we first define the q-factorial

$$[n]_q! = [n]_q[n-1]_q \dots [2]_q[1]_q$$

for $n \geq 1$ and $[0]_q! = 1$. Then $[n]_q$ is defined for $n \geq 0$ and the code is

```
0>>  def qfact(n):
1>>     if n == 0:
2>>        return 1
1>>     return _q_(n) * qfact(n-1)
>>>
>>>  qfact(5)
```

$(q+1)\left(q^2+q+1\right)\left(q^3+q^2+q+1\right)\left(q^4+q^3+q^2+q+1\right)$

```
>>>  expand(qfact(5))
```

$q^{10}+4q^9+9q^8+15q^7+20q^6+22q^5+20q^4+15q^3+9q^2+4q+1$

```
>>>  qfact(5).subs(q,1)
```

120

```
>>>  qfact(5).subs(q,1) == fact(5)
True
```

For $n \geq 1$, the *q-binomial coefficient* is

$$\begin{bmatrix} n \\ k \end{bmatrix}_q = \frac{[n]_q[n-1]_q \dots [n-k+1]_q}{[1]_q[2]_q \dots [k]_q} \tag{2.6.3}$$

for $1 \leq k \leq n$, and $\begin{bmatrix} n \\ 0 \end{bmatrix}_q = 1$. Define also $\begin{bmatrix} 0 \\ 0 \end{bmatrix}_q = 1$. Then $\begin{bmatrix} n \\ k \end{bmatrix}_q$ is defined for $0 \leq k \leq n$ and $n \geq 0$, and $\begin{bmatrix} n \\ n \end{bmatrix}_q = 1$. For example,

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}_q = 1, \qquad \begin{bmatrix} 1 \\ 1 \end{bmatrix}_q = \frac{[1]_q}{[1]_q} = 1, \qquad \begin{bmatrix} 2 \\ 1 \end{bmatrix}_q = \frac{[2]_q}{[1]_q} = [2]_q = q+1.$$

When $q=1$, $\begin{bmatrix} n \\ k \end{bmatrix}_q$ equals $\binom{n}{k}$,

$$\begin{bmatrix} n \\ k \end{bmatrix}_1 = \binom{n}{k}, \qquad 0 \leq k \leq n.$$

By canceling factors as before,

$$\begin{bmatrix} n \\ k \end{bmatrix}_q = \frac{[n]_q!}{[k]_q!\,[n-k]_q!}, \qquad 0 \leq k \leq n.$$

hence

$$\begin{bmatrix} n \\ k \end{bmatrix}_q = \begin{bmatrix} n \\ n-k \end{bmatrix}_q, \qquad 0 \le k \le n.$$

According to (2.6.3), the q-binomial coefficients are ratios of polynomials in q. Below we see these ratios simplify, and the q-binomial coefficients are in fact polynomials in q. For example, by recalling $[5]_q$ is a polynomial in q and canceling,

$$\begin{aligned}\begin{bmatrix} 5 \\ 2 \end{bmatrix}_q &= \frac{[5]_q[4]_q}{[1]_q[2]_q} = \frac{[5]_q}{[1]_q}\frac{q^4-1}{q-1}\frac{q-1}{q^2-1} \\ &= [5]_q\frac{q^4-1}{q^2-1} = [5]_q\frac{(q^2-1)(q^2+1)}{q^2-1} \\ &= (q^4+q^3+q^2+q+1)(q^2+1).\end{aligned}$$

We derive the q-analog of (2.4.1).

Theorem 2.6.2 *For $n \ge 1$,*

$$\begin{bmatrix} n+1 \\ k \end{bmatrix}_q = \begin{bmatrix} n \\ k \end{bmatrix}_q + q^{n-k+1}\begin{bmatrix} n \\ k-1 \end{bmatrix}_q, \qquad 1 \le k \le n. \tag{2.6.4}$$

Proof We do the cases $k = 1$ and $2 \le k \le n$ separately. When $k = 1$,

$$\begin{aligned}\begin{bmatrix} n+1 \\ 1 \end{bmatrix}_q &= [n+1]_q = 1 + q + \cdots + q^n \\ &= (1 + q + \cdots + q^{n-1}) + q^n = [n]_q + q^n \\ &= \begin{bmatrix} n \\ 1 \end{bmatrix}_q + q^{n-1+1}\begin{bmatrix} n \\ 0 \end{bmatrix}_q.\end{aligned}$$

This establishes the case $k = 1$. When $2 \le k \le n$, notice first, by Exercise 2.17,

$$\frac{1}{[k]_q} + \frac{q^{n-k+1}}{[n-k+1]_q} = \frac{1}{[n-k+1]_q[k]_q}.$$

Then, by (2.6.3),

$$\begin{aligned}
&\begin{bmatrix} n \\ k \end{bmatrix}_q + q^{n-k+1}\begin{bmatrix} n \\ k-1 \end{bmatrix}_q \\
&\quad = \frac{[n]_q[n-1]_q\dots[n-k+1]_q}{[1]_q[2]_q\dots[k]_q} + q^{n-k+1}\frac{[n]_q[n-1]_q\dots[n-k+2]_q}{[1]_q[2]_q\dots[k-1]_q} \\
&\quad = \frac{[n]_q[n-1]_q\dots[n-k+1]_q}{[1]_q[2]_q\dots[k-1]_q} \times \left(\frac{1}{[k]_q} + \frac{q^{n-k+1}}{[n-k+1]_q}\right) \\
&\quad = \frac{[n]_q[n-1]_q\dots[n-k+1]_q}{[1]_q[2]_q\dots[k-1]_q} \times \frac{1}{[n-k+1]_q[k]_q} \\
&= \frac{[n]_q[n-1]_q\dots[n-k+2]_q}{[1]_q[2]_q\dots[k]_q} \\
&= \begin{bmatrix} n+1 \\ k \end{bmatrix}_q .
\end{aligned}$$

This establishes (2.6.4). □

As a consequence,

Theorem 2.6.3 *Each* $\begin{bmatrix} n \\ k \end{bmatrix}_q$ *is a polynomial in* q, $1 \le k \le n$.

Proof We prove this by induction (as described just before Theorem 2.4.2). When $n = 1$, the result is clear, establishing the base step. To establish the inductive step, assume $\begin{bmatrix} n \\ k \end{bmatrix}_q$, $1 \le k \le n$, are polynomials in q. Then by (2.6.4), $\begin{bmatrix} n+1 \\ k \end{bmatrix}_q$, $1 \le k \le n$, are polynomials in q. Since this is also true of $\begin{bmatrix} n+1 \\ n+1 \end{bmatrix}_q = 1$, this establishes the inductive step. By induction, the result is true for all positive integers n. □

Note in §2.3, we started with (2.4.1) then derived (2.4.3), but here we are starting with (2.6.3) then deriving (2.6.4). In fact (2.4.1) and (2.4.3) are equivalent, and (2.6.4) and (2.6.3) are equivalent.

Here is code for $\begin{bmatrix} n \\ k \end{bmatrix}_q$ using (2.6.4),

```
0>>  def qbincoeff(n,k):
1>>     if k == 0 or k == n or n == 0 or n == 1:
2>>        return 1
1>>     a = qbincoeff(n-1,k-1)
1>>     b = qbincoeff(n-1,k)
1>>     return expand(q**(n-k)*a+b)
```

Then `qbincoeff(4,2)` returns $q^4 + q^3 + 2q^2 + q + 1$.

Now let the *q-binomial* be

$$[x+a]_q^n = (x+a)(x+qa)\dots(x+q^{n-1}a), \qquad n \ge 1, \tag{2.6.5}$$

and $[x+a]_q^0 = 1$. Then $q = 1$ implies

$$[x+a]_1^n = (x+a)^n;$$

then by Exercise 2.16, $x = 0$ implies

$$[x+0]_q^n = a^n\, q^{1+2+\cdots+(n-1)} = a^n\, q^{n(n-1)/2}.$$

The q-binomials $[x+a]_q^n$ are the correct q-analogs of the binomials $(x+a)^n$ because they behave like powers x^n,

Theorem 2.6.4 *For $n \geq 1$, $D_q[x+a]_q^n = [n]_q[x+a]_q^{n-1}$.*

Proof

$$\begin{aligned}
[x+a]_q^n &= [x+a]_q^{n-1}(x+q^{n-1}a),\\
[qx+a]_q^n &= (qx+a)(qx+qa)\ldots(qx+q^{n-1}a)\\
&= (qx+a)q^{n-1}(x+qa)\ldots(x+q^{n-2}a)\\
&= (qx+a)q^{n-1}[x+a]_q^{n-1}.
\end{aligned}$$

Hence

$$\begin{aligned}
D_q[x+a]_q^n &= \frac{[qx+a]_q^n - [x+a]_q^n}{qx-x}\\
&= \Big((qx+a)q^{n-1} - (x+q^{n-1}a)\Big)\,\frac{[x+a]_q^{n-1}}{qx-x}\\
&= \frac{q^n x - x}{qx-x}\,[x+a]_q^{n-1} = [n]_q[x+a]_q^{n-1}.
\end{aligned}$$

Here is code for the q-binomial,

```
>>>  (a,x,q) = var('a x q')
0>>  def qbinomial(n):
1>>     if n==1:
2>>        return x+a
1>>     return qbinomial(n-1) * (x+q**(n-1)*a)
>>>
>>>  qbinomial(2)
```

$(a+x)(aq+x)$

```
>>>  qbinomial(2).as_poly(x)
```

$x^2 + (aq+a)x + a^2q$

```
>>>  qbinomial(3)
```

$(a+x)(aq+x)(aq^2+x)$

```
>>>  qbinomial(3).as_poly(x)
```

$x^3 + (aq^2+aq+a)x^2 + (a^2q^2+a^2q^2+a^2q)x + a^3q^3$

Now we can derive

Theorem 2.6.5 (Gauss' q-Binomial Theorem) *For $n \geq 1$,*

$$[x+a]_q^n = \sum_{k=0}^{n} q^{k(k-1)/2} \begin{bmatrix} n \\ k \end{bmatrix}_q a^k x^{n-k},$$

or

$$\begin{aligned}[x+a]_q^n &= q^{n(n-1)/2} a^n + \begin{bmatrix} n \\ 1 \end{bmatrix}_q q^{(n-1)(n-2)/2} a^{n-1} x \\ &\quad + \begin{bmatrix} n \\ 2 \end{bmatrix}_q q^{(n-2)(n-3)/2} a^{n-2} x^2 + \cdots + [n]_q a x^{n-1} + x^n.\end{aligned}$$

Proof $[x+a]_q^n$ is a polynomial in x, a, and q, so we can write

$$[x+a]_q^n = Aa^n + Ba^{n-1}x + Ca^{n-2}x^2 + \dots. \tag{2.6.6}$$

To establish the Theorem, we show $A = q^{n(n-1)/2}$, $B = [n]_q\, q^{(n-1)(n-2)/2}$, $C = \begin{bmatrix} n \\ 2 \end{bmatrix}_q q^{(n-2)(n-3)/2}$, and so on. Now $(0+a)_q^n = a^n q^{n(n-1)/2}$, so $A = q^{n(n-1)/2}$. Applying D_q to both sides of (2.6.6), by Theorem 2.6.4 we get

$$[n]_q [x+a]_q^{n-1} = Ba^{n-1} + Ca^{n-2}[2]_q x + \dots.$$

Plugging $x = 0$ yields

$$B = [n]_q\, q^{(n-1)(n-2)/2}.$$

Applying D_q twice to (2.6.6) and plugging in $x = 0$, yields

$$[2]_q\, C = [n]_q [n-1]_q\, q^{(n-2)(n-3)/2}.$$

By (2.6.3), these are the correct coefficients $A, B, C, \dots$. □

Of course the q-binomial theorem reduces to the binomial theorem when $q = 1$. Note (2.6.6) can be written

$$\begin{aligned}[x+a]_q^n &= x^n + [n]_q a x^{n-1} + \begin{bmatrix} n \\ 2 \end{bmatrix}_q q a^2 x^{n-2} \\ &\quad + \begin{bmatrix} n \\ 3 \end{bmatrix}_q q^3 a^3 x^{n-3} + \begin{bmatrix} n \\ 4 \end{bmatrix}_q q^6 a^4 x^{n-4} + \dots\end{aligned} \tag{2.6.7}$$

Now in (2.6.7) replace x and a by 1 and x respectively, obtaining,

$$[1+x]_q^n = 1 + [n]_q x + \begin{bmatrix} n \\ 2 \end{bmatrix}_q q x^2 + \begin{bmatrix} n \\ 3 \end{bmatrix}_q q^3 x^3 + \cdots + x^n. \tag{2.6.8}$$

What happens if we let $n = \infty$, i.e. let n become very large? If q is strictly less than 1, $|q| < 1$, then $q^\infty = 0$, in the sense that q^n approaches zero as n approaches infinity. For example, if $q = .1$, then

$$q^2 = .01, \quad q^3 = .001, \quad q^4 = .0001, \quad q^5 = .00001, \quad \ldots.$$

In this case, we can insert $n = \infty$ into $[n]_q$ and obtain

$$[\infty]_q = \frac{1 - q^\infty}{1 - q} = \frac{1}{1 - q},$$

which leads to

$$\begin{aligned} \begin{bmatrix} \infty \\ k \end{bmatrix}_q &= \frac{[\infty]_q [\infty]_q \ldots [\infty]_q}{[1]_q [2]_q \ldots [k]_q} \\ &= \frac{1}{(1-q)(1-q)\ldots(1-q)} \frac{(1-q)(1-q)\ldots(1-q)}{(1-q)(1-q^2)\ldots(1-q^k)} \\ &= \frac{1}{(1-q)(1-q^2)\ldots(1-q^k)}. \end{aligned}$$

Inserting $n = \infty$ in $[1+x]_q^n$, we obtain the infinite product

$$[1+x]_q^\infty = (1+x)(1+qx)(1+q^2x)\ldots. \tag{2.6.9}$$

Inserting $n = \infty$ in (2.6.8), we conclude the infinite product (2.6.9) equals the infinite sum

$$1 + \frac{1}{1-q}x + \frac{q}{(1-q)(1-q^2)}x^2 + \frac{q^3}{(1-q)(1-q^2)(1-q^3)}x^3 + \ldots. \tag{2.6.10}$$

In summation notation, the equality between (2.6.9) and (2.6.10) becomes[4]

Theorem 2.6.6 (Euler's Identity)

$$\prod_{n=0}^{\infty}(1+q^n x) = \sum_{n=0}^{\infty} \frac{q^{n(n-1)/2}}{(1-q)(1-q^2)\ldots(1-q^n)} x^n.$$

In effect, when $n = \infty$, Gauss' q-binomial theorem becomes Euler's identity.

Exercises

Exercise 2.1 What does the following code do?

```
0>>  i = 0
0>>  while i < 11:
1>>     print(i)
```

[4] This is not the only *Euler's Identity*, there are many.

```
1>>     i = i + 2
```

What happens if you remove the last statement and re-run the code?

Exercise 2.2 What does the following code do?

```
>>>  a = input('Enter any integer: ')
0>>  if int(a) > 5:
1>>     print('what you entered is more than 5')
0>>  else:
1>>     print('what you entered is less or equal to 5')
```

What happens if you leave off `int`?

Exercise 2.3 With

```
>>>  (a,b,t) = var('a b t')
>>>  p = a**3*b - 5*a*b**2 + t**5*a*b
```

how do `p.as_poly(a).coeffs()` and `p.as_poly(a).all_coeffs()` differ?

Exercise 2.4 Let `a` be a list. What does `a.extend('alpha')` do? What's the difference between

```
>>>  a.append([23,'alpha'])}
>>>  a.extend([23,'alpha'])}
```

Exercise 2.5 Expand $(x + a)(x + qa)(x + q^2a)$ by hand, then check your answer by repeating this in Python.

Exercise 2.6 Here is a function that is *recursive*, it calls itself. Recall `a//b` is the integer quotient (§1.3). Also `n%2 == 0` is the same as saying there is no remainder after dividing n by 2, or, in other words, n is *even*. What does the following code do?

```
0>>  def collatz(n):
1>>     if n == 1:
2>>        return [1]
1>>     elif n%2 == 0:
2>>        return [n] + collatz(n//2)
1>>     else:
2>>        return [n] + collatz(3*n + 1)
```

Run

```
>>>  collatz(n)
```

with n equal to 1, 2, 3, 4, up till 10, and also 27.

Exercise 2.7 Show that for $n \geq 1$,

$$x^n - a^n = (x-a)(x^{n-1} + x^{n-2}a + x^{n-3}a^2 + \cdots + a^n).$$

Conclude if $f(x)$ is any polynomial, there is another polynomial $g(x,a)$ satisfying

$$f(x) - f(a) = (x-a)g(x,a).$$

Exercise 2.8 Show that for any positive numbers t_1, t_2

$$\left(\frac{t_1+t_2}{2}\right)^2 \geq t_1t_2.$$

Exercise 2.9 Let $p_k(t_1,\dots,t_n)$ be the elementary symmetric polynomials. Show that

$$p_k(1,q,q^2,\dots,q^{n-1}) = \begin{bmatrix} n \\ k \end{bmatrix}_q, \qquad 0 \leq k \leq n.$$

Exercise 2.10 Let $q_k(t_1,\dots,t_n) = p_k(t_1,\dots,t_n)/\binom{n}{k}$, $0 \leq k \leq n$, be the *normalized* elementary symmetric polynomials. Show that for any positive numbers t_1, t_2, t_3, and $n=3$,

$$q_1^2 \geq q_0q_2, \qquad q_2^2 \geq q_1q_3.$$

For general n, these are *Newton's inequalities*:

$$q_1^2 \geq q_0q_2, \qquad q_2^2 \geq q_1q_3, \qquad q_3^2 \geq q_2q_4, \qquad q_4^2 \geq q_3q_5, \qquad \dots$$

Exercise 2.11 Expand

$$(a+t_1b)(a+t_2b)(a+t_3b)(a+t_4b) = a^4 + p_1a^3b + p_2a^2b^2 + p_3ab^3 + p_4b^4,$$

and write out the polynomials p_1, p_2, p_3, p_4.

Exercise 2.12 Write a function `bincoeff(n,k)` that returns $\binom{n}{k}$ using (2.4.3).

Exercise 2.13 Define the function

```
0>>  def binomial(n):
1>>      return (x+a)**n
```

What does the code

```
>>>  binomial(n).as_poly(x).coeffs()[k] == bincoeff(n,k)
```

return? Why?

Exercise 2.14 What does

```
>>>  (a + a == b, a + a is b)
```

return if `a = [23,'alpha']` and `b = [23,'alpha',23,'alpha']`? Why? (Compare with Exercise 1.7.)

Exercise 2.15 Build the top 5 rows of the q-Pascal triangle.

Exercise 2.16 Show by induction (as described just before Theorem 2.4.2) that

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}, \qquad n \geq 1.$$

Exercise 2.17 Show that

$$\frac{q^{n-k+1}}{[n-k+1]_q} + \frac{1}{[k]_q} = \frac{1}{[n-k+1]_q[k]_q}.$$

Exercise 2.18 Let f and g be polynomials, and let $h = f \cdot g$. Derive the q-*(product rule)*

$$D_q h(x) = D_q f(x) \cdot g(qx) + f(x) \cdot D_q g(x).$$

Exercise 2.19 Use the q-(product rule) to derive Theorem 2.6.1.

Exercise 2.20 Write a function `eulerprod(n)` that returns

$$p_n(x) = (1+x)(1+qx)\dots(1+q^{n-1}x).$$

Use

```
>>>  eulerprod(n).as_poly(x).coeffs()[1]
```

to compute the coefficient of x in $p_n(x)$ for $n = 1, 2, 3, 4, 5$. Use Euler's identity and $n = \infty$ to show

$$\frac{1}{1-q} = 1 + q + q^2 + q^3 + \dots$$

Exercise 2.21 Use

```
>>>  eulerprod(n).as_poly(x).coeffs()[2]
```

to compute the coefficient of x^2 in $p_n(x)$ for $n = 1, 2, 3, 4, 5$. Use Euler's identity and $n = \infty$ to show

$$\frac{q}{(1-q)(1-q^2)} = q + q^2 + 2q^3 + 2q^4 + 3q^5 + 3q^6 + 4q^7 + 4q^8 + \dots$$

Exercise 2.22 Use the binomial theorem to show

$$\left(1 + \frac{x}{n}\right)^n = \sum_{k=0}^{n} n^{-k} \binom{n}{k} x^k.$$

Using $1/\infty = 0$, simplify $n^{-k}\binom{n}{k}$ to show

$$\infty^{-k}\binom{\infty}{k} = \frac{1}{k!}.$$

Conclude

$$\left(1 + \frac{x}{\infty}\right)^{\infty} = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Chapter 3
Sets

A *set* is a collection of things. For example you have the set of students in the class, the set of students taller than 5′6″, the set of integers between 1 and 5, and so on. Sets in Python are enclosed with braces,

```
>>>  A = {"alpha",23}
>>>  id(A)
9632421360
>>>  type(A)
set
>>>  A
{'alpha',23}
```

If a is in a set A, we say a is an *element* of A, so "an element in A" is another way of saying a "thing in A".

```
>>>  'alpha' in A
True
>>>  'beta' in A
False
```

The key property of sets is: *Sets are determined by their elements*. If every element of A is an element of B and every element of B is an element of A, then the sets A and B are considered the same. In particular, this implies sets are *unordered*,

```
>>>  C = {23,'alpha'}
>>>  A == C
True
```

The *cardinality* of a set is the number of elements in it.

```
>>>  len(A)
2
```

so the cardinality of A is 2. We write $|A|$ for the cardinality of A.

Sets can also be created by feeding a `list` or a `range` to `set`,

```
>>>  a = ['a',1,23,1,'alpha']
>>>  a
['a',1,23,1,'alpha']
>>>  type(a)
list
>>>  len(a)
5
>>>  B = set(a)
>>>  type(B)
set
>>>  len(B)
4
>>>  B
{1, 23, 'a', 'alpha'}
>>>  U = set(range(1,10))
>>>  U
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

As you see, sets are not the same as lists.

A set is *empty* if it has no elements. Since sets are determined by their elements, there is only one empty set.

```
>>>  E = set()
>>>  E
set()
>>>  len(E)
0
>>>  F  = set()
>>>  E == F
True
```

Be careful though, `{}` isn't the empty set, it's the empty `dict` (§3.4).

```
>>>  D = {}
>>>  type(D)
dict
>>>  D
{}
```

A set is *nonempty* if it has at least one element. We say A is a *subset* of B if every element of A is an element of B, and we then write $A \subset B$. Equivalently, we say B is a *superset* of A. For example, the set of blue horses is a subset of the set of all horses. Also every set A is a subset of itself.

```
>>>  A.issubset(B)
True
>>>  B.issuperset(A)
True
```

```
>>>  A.add('beta')
>>>  A.issubset(B)
False
>>>  A
{23, 'alpha', 'beta'}
>>>  id(A)
9632421360
```

Notice sets are mutable.

The key property of sets then can be rephrased as: If $A \subset B$ and $B \subset A$, then $A = B$.

Theorem 3.0.1 *A set of cardinality n has $\binom{n}{k}$ subsets of cardinality k.*

Proof This is an immediate consequence of Theorem 2.3.2. □

So equation (2.4.2) is saying the *power set of* A, the set of *all* subsets of A, has 2^n elements. Because of this, the power set is often written 2^A. In other words,

$$\left|2^A\right| = 2^{|A|}.$$

Sometimes subsets of a given set U are specified by a condition, for example

$$A = \{x \text{ in } U : x \text{ is even}\},$$

where the colon : is short for "such that" or "satisfying". In Python, this can be written using *set comprehension* as

```
>>>  {x for x in range(1,10) if x%2 == 0}
{2, 4, 6, 8}
```

In general, a subset of the set `U` is specified by the set comprehension

`{x for x in U <condition>}.`

More generally, one may insert a `function` into a set comprehension,

```
>>>  {3*x+5 for x in range(1,20) if x%2 == 0}
{11, 17, 23, 29, 35, 41, 47, 53, 59}
```

3.1 Union and Intersection

If A and B are sets, then the *union* of A and B is the set of elements that are in A *or* in B,

$$A \cup B = \{x : x \text{ is in } A \text{ or is in } B\}.$$

```
>>>  A.union(B)
{1, 23, 'a', 'alpha', 'beta'}
```

If A and B are subsets of a set U, we may use set comprehension,

```
>>>  U = {1, 2, 23, 'a', 'beta', 'c', 'alpha'}
>>>  C = { x for x in U if x in A or x in B }
>>>  A.union(B) == C
True
```

The *intersection* of A and B is the set of elements that are in A *and* in B,

$$A \cap B = \{x : x \text{ is in } A \text{ and is in } B\}.$$

```
>>>  A.intersection(B)
{23, 'alpha'}
>>>  C = { x for x in U if x in A and x in B }
>>>  A.intersection(B) == C
True
```

Theorem 3.1.1 (Distributivity) *For any sets A, B, C,*

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

Proof We have to show an equality between sets. Thus we have to show every element of $A \cap (B \cup C)$ is an element of $(A \cap B) \cup (A \cap C)$, and vice-versa. Let a be an element of $A \cap (B \cup C)$. Then a is in A and a is in $B \cup C$. Thus a is in A and a is in B or in C. In the first case, a is in $A \cap B$. In the second case, a is in $A \cap C$. Thus a is in $A \cap B$ or in $A \cap C$, hence in $(A \cap B) \cup (A \cap C)$. This shows $A \cap (B \cup C) \subset (A \cap B) \cup (A \cap C)$. For the reverse inequality, suppose a is in $(A \cap B) \cup (A \cap C)$. Then a is in $A \cap B$ or a is in $A \cap C$. In either case, a is in A, and, in either case, a is in $B \cup C$. Thus a is in $A \cap (B \cup C)$. This shows $A \cap (B \cup C) \supset (A \cap B) \cup (A \cap C)$. By the key property of sets, $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$. □

More generally, by induction one can show (Exercise 3.3)

$$(A_1 \cup A_2 \cup \ldots \cup A_n) \cap B = (A_1 \cap B) \cup (A_2 \cap B) \cup \ldots \cup (A_n \cap B). \qquad (3.1.1)$$

If $A \cap B$ is empty, then we say A and B have no elements in common, or A and B are *disjoint*.

```
>>>  A.isdisjoint(B)
False
```

The *difference* $B - A$ is the set of elements in B that are not in A. $B - A$ is also called the *complement* of A in B.

```
>>>  B-A
{1,'a'}
```

Theorem 3.1.2 (Additivity) *For sets A and B,*

$$A = (A \cap B) \cup (A - B).$$

Proof If a is in A, then either a is in B, or a is not in B. In the first case, a is in $A \cap B$. In the second case, a is in $A - B$. Hence, in either case, a is in $(A \cap B) \cup (A - B)$. Thus $A \subset (A \cap B) \cup (A - B)$. For the reverse inequality, let a be in $(A \cap B) \cup (A - B)$. Then a is in $A \cap B$ or a is in $A - B$. In the first case, a is in A and B, while the second case, a is in A but not in B. Hence, in either case, a is in A. Thus $A \supset (A \cap B) \cup (A - B)$. By the key property of sets, $A = (A \cap B) \cup (A - B)$. □

3.2 Finite and Infinite Sets

A set A is *finite* if it has n elements for some positive integer n. The set of integers strictly between 1 and 5 is finite, as it has 3 elements. If A is not finite, we say A is *infinite*. In Chapter 4, we will see that the set of integers is infinite.

Recall (§2.2) `list`s and `range`s are both *subscriptable*: Since sets are unordered, trying to access an element of a set by subscript makes no sense,

```
>>>  s = set(range(1,10))
>>>  s[0]
Error: 'set' object is not subscriptable
>>>  s[-1]
Error: 'set' object is not subscriptable
```

Since `list`s and `range`s are both subscriptable, what's the difference? With `r = range(1,n)` and `a = list(r)`, the latter data is immediately created, while the former is only created as needed. The latter occupies $n - 1$ locations in memory as soon as it is created, while the former occupies only whatever space is needed when the code is run. To see this, run both commands with with $n = 1000000$. This point is especially vivid for infinite sets.

```
0>>  def posint():
1>>     i=1
1>>     while 1:
2>>        yield i
2>>        i += 1
```

Because of the keyword `yield`, `posint` is a function[1] that returns a `generator`, an object that generates *all* positive integers,

```
>>>  type(posint)
function
>>>  Zpos = posint()
```

[1] Remember, `posint` is a function, and `posint()` is what the function returns.

```
>>>  type(Zpos)
generator
```

`Zpos` is the infinite set of all positive integers

```
>>>  1 in Zpos
True
>>>  100 in Zpos
True
>>>  1000000 in Zpos
True
```

but

```
 >>> list(Zpos)
```

will never return, as Python is attempting to store the list, all the way to infinity, on the spot. Similarly, because the set of positive integers is infinite, checking

```
>>>  -2 in Zpos
```

will go on forever and never return. Python has to check all positive integers to determine whether or not -2 is a positive integer! To drive this point home, run the following code

```
0>>   for a in Zpos:
1>>     print(a)
```

`Zpos` is an example of an `iterator`. An `iterator` is any Python object which allows you extract the next element,

```
>>>  next(Zpos)
1
>>>  next(Zpos)
2
>>>  next(Zpos)
3
```

Any function with the keyword `yield` in its body returns a `generator`, and any `generator` is an `iterator`. An `iterator` may be obtained from a list by feeding it to `iter()`,

```
>>>  a = [1, 3, 'a', 'alpha', 'b']
>>>  next(a)
Error: 'list' object is not an iterator
>>>  i = iter(a)
>>>  type(i)
list_iterator
>>>  next(i)
1
>>>  next(i)
```

```
3
>>>  next(i)
'a'
```

Similarly,

```
>>>  s = 'alpha'
>>>  next(s)
Error: 'str' object is not an iterator
>>>  i = iter(s)
>>>  type(i)
str_iterator
>>>  next(i)
'a'
>>>  next(i)
'l'
>>>  next(i)
'p'
>>>  next(i)
'h'
>>>  next(i)
'a'
>>>  next(i)
StopIteration
```

The last step returning `StopIteration`, because there are no more items in i. Similarly, one can create `iterator`s out of sets.

Any Python object that can be fed to `iter` is iterable. So sets, lists, and strings are iterable. However, an `int` is not iterable,

```
>>>  a = 5
>>>  i = iter(a)
Error: 'int' object is not iterable
```

In particular, applying `iter` twice is the same as applying it once

```
>>>  s = 'alpha'
>>>  i = iter(s)
>>>  i == iter(i)
True
```

So an `iterator` is iterable. Any iterable `object` may be entered into a `for` loop,

```
>>>  for y in [1, 3, 'a', 'alpha', 'b']:
>>>  for char in 'alpha':
>>>  for element in {1, 3, 'a', 'alpha', 'b'}:
>>>  for n in range(1,10):
>>>  for n in Zpos:
```

When the loop executes, the iterable `object` becomes an `iterator`, then the loop iterates through the entries of `object` by using `next`. But this happens behind the scenes, all you have to do is write the loop header as above.

One last point is that an `iterator` is not subscriptable,

```
>>>  s = 'alpha'
>>>  s[0]
'a'
>>>  i = iter(s)
>>>  i[0]
Error: 'str_iterator' object is not subscriptable
```

So subscriptable does not imply iterable, and iterable does not imply subscriptable.

3.3 Inclusion-Exclusion

If A is a set, $|A|$ denotes the cardinality of A. An empty set E has no elements, so $|E| = 0$.

Recall sets A and B are *disjoint* if they have no elements in common. This is the same as saying $A \cap B$ is the empty set. When A and B are disjoint, it is immediate that

$$|A \cup B| = |A| + |B|. \tag{3.3.1}$$

If $A_1, A_2, \dots, A_n$ are sets, we say they are *pairwise disjoint* if any two sets A_i and A_j are disjoint.

Theorem 3.3.1 (Additivity) *If $A_1, A_2, \dots, A_n$ are pairwise disjoint sets, then*

$$|A_1 \cup \ldots \cup A_n| = |A_1| + |A_2| + \cdots + |A_n|. \tag{3.3.2}$$

Proof We prove this by induction (as described just before Theorem 2.4.2). When $n = 1$, (3.3.2) says $|A_1| = |A_1|$, so the base step is immediate. Now assume (3.3.2) is valid, and let $A_1, A_2, \dots, A_n, A_{n+1}$ be $(n+1)$ pairwise disjoint sets. Then A_{n+1} and $A_1 \cup A_2 \cup \ldots \cup A_n$ are disjoint, so by (3.3.1) and (3.3.2),

$$\begin{aligned}|A_1 \cup \ldots \cup A_{n+1}| &= |A_1 \cup \ldots \cup A_n| + |A_{n+1}| \\ &= (|A_1| + |A_2| + \cdots + |A_n|) + |A_{n+1}| \\ &= |A_1| + |A_2| + \cdots + |A_n| + |A_{n+1}|.\end{aligned}$$

This establishes the inductive step, hence, by induction, (3.3.2) is valid for all positive integers n. □

If A and B intersect, then the common elements shouldn't be counted twice, so the correct formula is

Theorem 3.3.2 *If A and B are sets,*

$$|A \cup B| = |A| + |B| - |A \cap B|. \tag{3.3.3}$$

Proof By Exercise 3.2,

$$A \cup B = B \cup (A - B).$$

But B and $A - B$ are disjoint, so

$$|A \cup B| = |B| + |A - B|.$$

On the other hand, $A \cap B$ and $A - B$ are disjoint, so

$$|A| = |A \cap B| + |A - B|.$$

Subtracting the last equation from the previous one, (3.3.3) follows. □

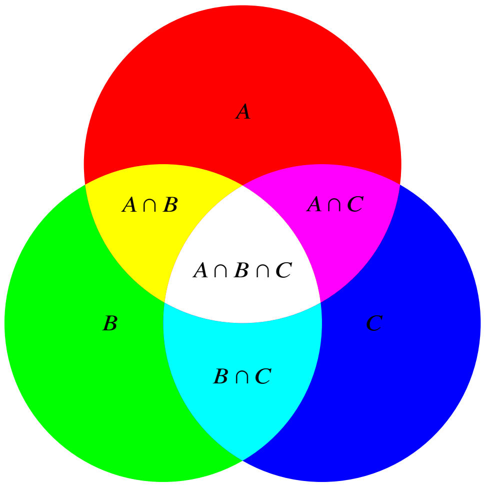

Fig. 3.1 Inclusion-exclusion principle.

If A, B, C are sets, the formula is

$$\begin{aligned} |A \cup B \cup C| =& |A| + |B| + |C| - |A \cap B| \\ & - |B \cap C| - |C \cap A| + |A \cap B \cap C|. \end{aligned} \tag{3.3.4}$$

Note there are now 3 terms with single sets, and 3 terms with intersections of pairs (Figure 3.1). In general,

Theorem 3.3.3 (Inclusion-Exclusion Principle) *If $A_1, A_2, \ldots, A_n$ are sets,* [2]

[2] Note the similarity with (2.5.5). This is not an accident.

$$|A_1 \cup \ldots \cup A_n| = \sum |A_i| - \sum |A_i \cap A_j| + \sum |A_i \cap A_j \cap A_k| - \ldots. \tag{3.3.5}$$

Notice the signs alternating between + and − as you go through the formula. Here the first sum is over all single sets, so there are n terms in the first sum. The second sum over all intersection pairs, so the number of terms in the second sum is the number of ways a pair of sets can be chosen from n sets, hence there are $\binom{n}{2}$ terms in the second sum. Similarly, the third sum is over all intersection triples, so there are $\binom{n}{3}$ terms in the third sum. In general, the k-th sum has $\binom{n}{k}$ terms.

For example, if you have 4 sets A, B, C, D, the first sum has 4 terms $|A|$, $|B|$, $|C|$, $|D|$, the second sum 6 terms $|A \cap B|$, $|A \cap C|$, $|A \cap D|$, $|B \cap C|$, $|B \cap D|$, $C \cap D|$, the third sum 4 terms $|A \cap B \cap C|$, $|A \cap B \cap D|$, $|A \cap C \cap D|$, $|B \cap C \cap D|$, and the final sum 1 term $|A \cap B \cap C \cap D|$.

Similarly, if you have 5 sets, the first sum has 5 terms, the second sum 10 terms, the third sum 10 terms, the fourth sum 5 terms, and the final sum 1 term.

In (3.3.5), the first sum is over $1 \le i \le n$, the second sum over $1 \le i < j \le n$, the third sum over $1 \le i < j < k \le n$, and so on.

Proof The issue here is that each element a in $A_1 \cup \ldots \cup A_n$ be counted exactly once on the right side of (3.3.5). More specifically, suppose we add an element a to $A_1 \cup \ldots \cup A_n$. Then the left side of (3.3.5) increases by 1. If we show that the right side of (3.3.5) also increases by 1, (3.3.5) will then be proved, by induction on the number of elements in $A_1 \cup \ldots \cup A_n$.

If a (the element we're adding) is only a member of a single set A_i, then a is not a member of any intersection pair, nor of any intersection triple, and so on, so a is indeed counted once on the right side of (3.3.5).

If a is a member of sets A_i and A_j, then a is a member of the intersection pair $A_i \cap A_j$, and not a member of any other intersection pair, nor of any intersection triple $A_i \cap A_j \cap A_k$, and so on, so, in (3.3.5), a is counted twice in the first sum, once in the second sum, and not at all in the remaining sums. Hence a is indeed counted once on the right side of (3.3.5).

If a is a member of sets A_i and A_j and A_k, then a is a member of the intersections pairs $A_i \cap A_j$, $A_j \cap A_k$, $A_k \cap A_i$, and the intersection triple $A_i \cap A_j \cap A_k$, and not a member of any other intersection pair, nor of any other intersection triple, and so on, so, in (3.3.5), a is counted three times in the first sum, three times in the second sum, once in the third sum, and not at all in the remaining sums. Since $3 - 3 + 1 = 1$, a is indeed counted once on the right side of (3.3.5).

If a is a member of k sets selected from $A_1, \ldots, A_n$, then a is a member of $\binom{k}{2}$ intersection pairs, $\binom{k}{3}$ intersection triples, and so on, so, a is counted

$$\binom{k}{1} - \binom{k}{2} + \binom{k}{3} - \ldots$$

times on the right side of (3.3.5). But (§2.4) the alternating sum of the binomial coefficients along the k-th row of Pascal's triangle is zero, hence this last sum equals $\binom{k}{0} = 1$. Thus a is indeed counted once on the right side of (3.3.5). □

3.4 Dictionaries

A `dict` is roughly a set whose elements are key-value pairs. As such, a `dict` is partially subscriptable: It is subscriptable only if the keys are present,

```
>>>  d = { 'name':'Bob', 'age':64,'height':'70in' }
>>>  id(d)
4509242064
>>>  type(d)
dict
>>>  d[2]
Error: 2
>>>  d['weight']
Error: 'weight'
>>>  d['name']
'Bob'
>>>  d['degree']
Error: 'degree'
```

The empty `dict` is `{}`.[3] One may recover the keys as a list or as a set,

```
>>>  k = d.keys()
>>>  type(k)
dict_keys
>>>  k
dict_keys(['name', 'age', 'height'])
>>>  list(k)
['name', 'age', 'height']
>>>  set(k)
{'age', 'height', 'name'}
```

Similarly, one may recover the values,

```
>>>  v = d.values()
>>>  type(v)
dict_values
>>>  v
dict_values(['Bob', 64, '70in'])
>>>  list(v)
['Bob', 64, '70in']
>>>  set(k)
{64, '70in', 'Bob'}
```

Values may be modified,

[3] This is a historical accident. For consistency, the empty `set` should be written `{}` and the empty `dict` should be written `{:}`. However, correcting this inconsistency in Python would necessitate checking millions of lines of code around the world.

```
>>>  d['name'] = 'Anne'
>>>  d
{ 'name':'Anne', 'age':64,'height':'70in' }
>>>  id(d)
4509242064
```

hence `dict`s are mutable. However, keys may not be modified, they are *immutable*. A `dict` is iterable,

```
0>>  for a in d:
1>>     print(a)
name
age
height
```

Values in a `dict` may be any `object`s, while keys must be immutable `object`s.[4] Conversely, if A is a set, we may create a `dict` from it with values all equal to 1,

```
>>>  A = { 23, 'alpha', 'b' }
>>>  d = { 23:1, 'alpha':1, 'b':1 }
>>>  set(d.keys()) == A
True
```

More generally, a `dict` with keys from A and with values all equal to 0 or 1 corresponds to a subset of A

```
>>>  A = { 23, 'alpha', 'b' }
>>>  d = { 23:1, 'alpha':1, 'b':0 }
>>>  B = set(d.keys())
>>>  B.issubset(A)
True
```

In this sense, one can think of `dict`s with keys in A as generalizations of subsets of A.

3.5 Maps

This section discusses `tuple` (Python) and ordered pairs, products of sets, relations, and maps (math).

Given a and b, we want to define an object (a, b) so that

$$(a, b) = (c, d) \qquad \Longrightarrow \qquad a = c \quad \text{and} \quad b = d. \tag{3.5.1}$$

We might try $(a, b) = \{a, b\}$, but that doesn't work because sets are unordered, $\{a, b\} = \{b, a\}$, and (3.5.1) insists the first, a, matches with the first, c, and the

[4] This is not quite correct, keys must be *hashable*, which is more restrictive than immutable.

second, b, matches with the second, d. It doesn't matter how (a, b) is constructed, as long as the defining property (3.5.1) holds.

We define the *ordered pair* (a, b) by setting

$$(a, b) = \{\{a\}, \{a, b\}\}. \tag{3.5.2}$$

Thus (a, b) is a set with two elements, the first being the set $\{a\}$, and the second being the set $\{a, b\}$. Thus the first element of (a, b) has cardinality 1, and the second element of (a, b) has cardinality 1 or 2 according to $a = b$ or $a \neq b$.

If A is a collection of sets, let $\cup A$ denote the union of the elements of A. For example, if A has only one element, $A = \{a\}$, then $\cup A = a$. If A has two elements $\{a, b\}$, then $\cup A = a \cup b$. Let $\cap A$ be the intersection of the elements in A. For example, if A has only one element, $A = \{a\}$, then $\cap A = a$. If A has two elements $\{a, b\}$, then $\cap A = a \cap b$.

Theorem 3.5.1 *If (a, b) is defined by (3.5.2), then (3.5.1) holds.*

Proof $\cup(a, b)$ is the union of the elements of (a, b), that is the union of $\{a\}$ and $\{a, b\}$, so $\cup(a, b) = \{a, b\}$. $\cap(a, b)$ is the intersection of the elements of (a, b), that is the intersection of $\{a\}$ and $\{a, b\}$, so $\cap(a, b) = \{a\} \cap \{a, b\} = \{a\}$. It follows that $\cup\cup(a, b) = a \cup b$ and $\cap\cup(a, b) = a \cap b$. Since $\cup\{a\} = a$, we also have $\cup\cap(a, b) = a$. Hence if $(a, b) = (c, d)$, then

$$a = \cup \cap (a, b) = \cup \cap (c, d) = c,$$

and

$$a \cup b = \cup \cup (a, b) = \cup \cup (c, d) = c \cup d,$$

$$a \cap b = \cap \cup (a, b) = \cap \cup (c, d) = c \cap d.$$

But $b = ((a \cup b) - a) \cup (a \cap b)$ by Exercise 3.11. Hence

$$b = ((a \cup b) - a) \cup (a \cap b) = ((c \cup d) - c) \cup (c \cap d) = d.$$

A `tuple` is a sequence of values, separated by commas, and enclosed in parentheses,

```
>>>  a = ( 23,'alpha' )
>>>  type(a)
tuple
>>>  id(a)
4523120912
>>>  len(a)
2
>>>  a[0]
23
>>>  a[-1]
'alpha'
```

Just like an ordered pair (a, b), the `tuple` `(23,'alpha')` is ordered.

```
>>>  ( 23,'alpha' ) == ('alpha',23)
False
```

When writing a `tuple`, the enclosing parentheses are unnecessary

```
>>>  23,'alpha'
(23,'alpha')
```

A set is unordered, but can be turned into a `tuple`,

```
>>>  a = { 23,'alpha' }
>>>  tuple(a)
(23,'alpha')
>>>  b = { 'alpha',23 }
>>>  tuple(b)
('alpha',23)
>>>  a == b
True
>>> tuple(a) == tuple(b)
False
```

Ordered pairs can be generalized to ordered triples (a, b, c), ordered quadruples (a, b, c, d), etc, and `tuple`s can be written with three, four, or more elements, `(23,'alpha','beta')`. Tuples may be added

```
>>>  a = (23,'alpha' )
>>>  tuple(a)
(23,'alpha')
>>>  b =  ('beta',23)
>>>  a + b
(23,'alpha','beta',23)
```

Like `list`s or `set`s, `tuple`s are iterable; like `list`s, `tuple`s are subscriptable; however, unlike `list`s or `set`s, `tuple`s are immutable,

```
>>>  a = (23,'alpha')
>>>  a == (a[0],a[1])
True
>>>  a[0] = 35
Error: 'tuple' object does not support item assignment
```

Given sets X and Y, the *product* $X \times Y$ is the set of all ordered pairs (x, y), with x in X and y in Y,

$$X \times Y = \{(x, y) : x \text{ in } X, y \text{ in } Y\}.$$

A *relation* between X and Y is a subset f of $X \times Y$. If (x, y) is in f, we say x is related to y and y is related to x under f. The broadest relation between X and Y is $f = X \times Y$, everything in X is related to everything in Y. The strictest relation between X and X is the empty relation, where nothing is related. The *diagonal relation* is $d = \{(x, x) : x \text{ in } X\}$. Then every x in X is related only to itself under d.

The *source* of a relation $f \subset X \times Y$ is the set of x in X that are related to some y in Y under f. So the source is where the relation “starts”. The *target* of a relation $f \subset X \times Y$ is the set of y that are related to x for some x in X. So the target is where the relation “ends”. The target is denoted $f(X)$ and also called the *image* of X under f.

For example, let X be the set of all people, and Y the set of all horses. Let f be the relation x rides y, so (x, y) is in f if x rides y. The source is the set of people who ride horses, and the target is the set of horses that are ridden. Moreover, given x,

$$\{y : (x, y) \text{ in } f\}$$

is the set of horses that x rides, and given y,

$$\{x : (x, y) \text{ in } f\}$$

is the set of people that ride y. The source of f is X if all people ride horses. The target is Y if all horses are ridden.

A *map* from X to Y (in the strict mathematical sense) is a relation between X and Y where each x in X is related to exactly one y in Y. For example, the people-horse relation f is a map if all people ride horses, and each person rides exactly one horse. Then we have a map from people to horses.

Note a Python `dict` is analogous to a map. The keys are in the source, and the values are in the target.

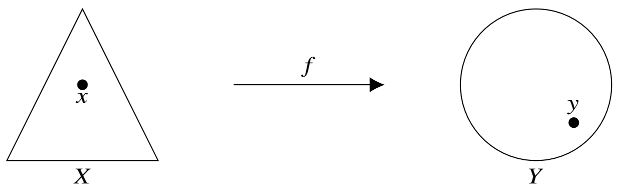

Fig. 3.2 A map.

If f is a map and (x, y) is in f, we write $y = f(x)$. For example, $f(\text{Jim})$ is the horse that Jim rides.

A map f from X to Y is *injective* if no two x’s go to the same y: $f(x) = y = f(x')$ implies $x = x'$. For example, the people-horse relation f is an injective map if all people ride horses and no two people ride the same horse.

A map f from X to Y is *surjective* if the target of f is Y: every y is related to some x. For example, the people-horse relation f is a surjective map if all people ride horses and every horse is ridden.

If f is a map from X to Y, and g is a map from Y to Z, the *composition* h of f with g is the map from X to Z given by applying f then g, $h(x) = g(f(x))$. The *identity* on X is the map i from X to X which leaves everything unchanged, $i(x) = x$.

A map g from Y to X is an *inverse* of a map f from X to Y if the composition of f with g is the identity on X, $g(f(x)) = x$ for all x in X, and the composition of g with f is the identity on Y, $f(g(y)) = y$ for all y in Y. Simply put, this says the map g undoes what the map f does: f is reversible.

If f has an inverse g, we say f is *bijective*. A map is bijective if and only if it is injective and surjective (Exercise 3.12).

For example, the people-horse relation f is a bijective map if all people ride horses, all horses are ridden, each person rides exactly one horse, and each horse is ridden by exactly one person. In other words, f is a bijective map if people and horses can be paired off in a one-to-one fashion.

We say sets X and Y *have the same cardinality* if there is a bijective map between X and Y. This applies even when X and Y are infinite sets.

Exercises

Exercise 3.1 Following the proof of Theorem 3.1.1, show for any sets A, B, U,

$$U - (A \cup B) = (U - A) \cap (U - B), \qquad U - (A \cap B) = (U - A) \cup (U - B).$$

This is often paraphrased as: *The complement of the union is the intersection, and the complement of the intersection is the union.*

Exercise 3.2 Show that $(A \cup B) \cap B = B$ and $(A \cup B) - B = A - B$. By replacing A by $A \cup B$ in Theorem 3.1.2, show $A \cup B = B \cup (A - B)$.

Exercise 3.3 Show (3.1.1) by induction.

Exercise 3.4 If A and B are subsets of U, write code returning $A - B$ as a set comprehension using `not in`.

Exercise 3.5 Let `f` be any `function` returning a list. Insert the statement `yield` at the end the function body. What does `f` return now? What if `f` originally returned a `str`?

Exercise 3.6 If `yield` were replaced by `return` in the function `posint`, what would be the type of `posint()`?

Exercise 3.7 A survey asks students whether they have a laptop, a car, or a mobile phone. 40 have a laptop; 60 have a car; and 50 have a mobile phone. 25 have a laptop and a car. 30 have a car and have a mobile phone. 35 have a laptop and a mobile phone. 10 students have all three. How many students have at least one of the three?

Exercise 3.8 How many integers $1 \le a \le 10000$ are divisible by 2, 3, or 5?

Exercise 3.9 Let V be a set with $|V| = k$. What is the number of `dict`s `d` with `len(d) == n` and `d.values()` in V?

Exercise 3.10 The values of the roman numerals are given by the `dict`

```
{'M':1000,'D':500,'C':100,'L':50,'X':10,'V':5,'I':1}
```

- Write a function `highest` that returns the index of the first occurrence of the highest-value numeral:

  `highest('XII') = 0`,   `highest('CDLVCDXII') = 1`.

- Write a function `roman` that translates a number from roman to decimal as follows. In roman notation, numerals are initially written left-to-right in decreasing order,

  ```
  >>>  roman('MCCLXII') == 1000 + 200 + 50 + 10 + 2
  True
  ```

  If, however, a numeral has greater value than numerals in a block of numerals to its left, then that block is counted with a minus. Thus `roman('IX')` is $-1 + 10 = 9$, and

  ```
  >>>  roman('MCLDII') == 1000 - (100+50) + 500 + 2
  True
  ```

Exercise 3.11 Show that for any sets A, B, $B = ((A \cup B) - A) \cup (B \cap A)$.

Exercise 3.12 Show that a map f from X to Y is bijective if and only if it is injective and surjective.

Exercise 3.13 Print out a `dict` whose keys are `range(0,256)` and whose values are the ASCII characters.

Chapter 4
Integers Z

In this chapter, we go back to basics and study integers from scratch. We do this because we will need to contrast standard arithmetic with modular arithmetic in Chapter 6.

What are the *integers* **Z**? Integers are often called the “whole numbers.” This is like answering “Who is John?” by saying “John is a person.” It’s just not informative. Sometimes integers are listed

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

This actually lists only seven integers, and the three dots are supposed to mean something like “there are more”. Since there are infinitely many integers, we cannot list them all.

We have no recourse other than to describe integers by their properties. There are *ten basic properties* or *axioms* that characterize the integers. They are

1. additive commutativity (4.1.1)
2. additive associativity (4.1.2)
3. existence of zero (4.1.3)
4. existence of negatives (4.2.1)
5. multiplicative commutativity (4.3.1)
6. multiplicative associativity (4.3.2)
7. existence of one (4.3.3)
8. distributivity (4.4.1)
9. minimality (4.5.1)
10. zero is neither positive nor negative (4.5.2)

Everything you know about arithmetic is a consequence of these axioms. In fact, our arithmetic manipulations in the previous chapters depended on these axioms, or their consequences, as developed in this chapter. While many of these axioms may seem obvious, in Chapter 6, we study the *modular integers,* where things behave differently. Without providing details, let us here describe the framework.

In §A.1, we show there is essentially one set $\mathbf{Z}$ equipped with addition and multiplication operations satisfying these axioms.

It is important to keep the context in mind. For example we introduce the concept of *reciprocal*, in three different contexts.

When the context is the rational numbers $\mathbf{Q}$ (Chapter 7), every rational $a \neq 0$ has a reciprocal, for example, the reciprocal of 2 is $1/2$, as you learned in grade school. When the context is the integers $\mathbf{Z}$ (the present chapter), only $a = 1$ and $a = -1$ have reciprocals. When the context is the modular integers $\mathbf{Z}_n$ (Chapter 6), a has a reciprocal if and only if a and the modulus n have no common factor.

So we get different answers, different behavior, depending on the context. Now we turn to the systematic development of the basic properties of $\mathbf{Z}$. In §A.1, we are by necessity completely formal in defining exactly what $\mathbf{Z}$ is. Until then, we describe the axioms of $\mathbf{Z}$ more informally as properties.

4.1 Addition

Integers may be *added*. If a and b are integers, $a + b$ is their *sum*. A basic property is *commutativity*: It doesn't matter which order we add the integers:

$$a + b = b + a \qquad \text{for every } a \text{ and } b. \tag{4.1.1}$$

How do we add a bigger bunch of integers, say $a + b + c$? One way is first add a and b, then add c to the sum $a + b$, yielding $(a + b) + c$. Another way is to first add b and c, then add a to the sum $b + c$, yielding $a + (b + c)$. *Associativity* guarantees that it doesn't matter which way you add them, the answer is the same:

$$(a + b) + c = a + (b + c) \qquad \text{for every } a \text{ and } b \text{ and } c. \tag{4.1.2}$$

Once you believe in associativity, then you can add any number of integers, it doesn't matter in which order you add them.

There is a special integer 0, called *zero*, satisfying

$$a + 0 = 0 + a = a \qquad \text{for every } a. \tag{4.1.3}$$

If an integer a is not zero, then we say a is *nonzero*.

Can there be more than a single zero, i.e. an integer satisfying (4.1.3)? That's a weird question, but let's show the answer is no.

Theorem 4.1.1 *There is only one integer zero.*

Proof Suppose z and z' are integers both satisfying the same property (4.1.3), i.e. $a + z = z + a = a$ for every a, and $a + z' = z' + a = a$ for every a. Then inserting $a = z'$ in the first equation yields $z + z' = z'$, and inserting $a = z$ in the second equation yields $z + z' = z$. Thus

$$z = z + z' = z'.$$

4.2 Subtraction

To every integer a corresponds its *negative* $-a$: This is the integer b which when added to a yields zero:

$$a + b = a + (-a) = 0. \tag{4.2.1}$$

Negativity is visualized as follows: If Tom gives Mary \$5, then after that Tom takes \$5 from Mary, Tom's net worth is unchanged. Therefore

$$\text{giving \$5} + \text{taking \$5} = \text{nothing changed},$$

which we write as

$$-(\text{giving \$5}) = \text{taking \$5}.$$

To repeat, $-a$ is the integer you add to a to obtain the sum 0. For example, $-0 = 0$, since $0 + 0 = 0$.

Could an integer have more than a single negative? That's another weird question, but let's show the answer is no.

Theorem 4.2.1 *Each integer has exactly one negative.*

Proof Suppose a is an integer and both b and c are negatives of a, so we are given $a + b = 0$ and $a + c = 0$. By (4.1.1), we then have $c + a = 0$, so

$$b = 0 + b = (c + a) + b = c + (a + b) = c + 0 = c.$$

Here we used (4.1.3), followed by $c + a = 0$, followed by (4.1.2), followed by the given $a + b = 0$, followed by (4.1.3). So $b = c$. □

So each integer a has a single negative, which we always write as $-a$.

Theorem 4.2.2

$$-(-a) = a \qquad \textit{for every } a.$$

Proof Since $(-a) + a = a + (-a) = 0$, it follows from (4.2.1) that a is the negative of $-a$. □

Integers may be *subtracted*. If a and b are integers, we say $c = a - b$ is their *difference* if $a = c + b$. *Subtraction* $a - b$ is simply the sum of a and the negative of b,

$$a - b = a + (-b).$$

So if you understand addition and negatives, you understand subtraction.

4.3 Multiplication

Integers can be *multiplied*: If a and b are integers, ab is their *product*. A basic property is *commutativity*: It doesn't matter which order we multiply integers:

$$ab = ba \qquad \text{for every } a \text{ and } b. \tag{4.3.1}$$

How do we multiply a bunch of integers abc? One way is first multiply a and b, then multiply c with the product ab, yielding $(ab)c$. Another way is to first multiply b and c, then multiply a with the product bc, yielding $a(bc)$. *Associativity* guarantees that it doesn't matter which way you multiply them, the answer is the same:

$$(ab)c = a(bc) \qquad \text{for every } a \text{ and } b \text{ and } c. \tag{4.3.2}$$

Once you believe in associativity, then you can multiply any number of integers, it doesn't matter in which order you multiply them.

There is a special integer 1, called *one*, satisfying

$$a \cdot 1 = 1 \cdot a = a \qquad \text{for every } a. \tag{4.3.3}$$

Can there be more than a single one, i.e. an integer satisfying (4.3.3)? Let's show the answer is no.

Theorem 4.3.1 *There is only one integer one.*

Proof Suppose o and o' are integers both satisfying (4.3.3), i.e. $ao = oa = a$ for every a, and $ao' = o'a = a$ for every a. Then inserting $a = o'$ in the first equation yields $oo' = o'$, and inserting $a = o$ in the second equation yields $oo' = o$. Thus

$$o = oo' = o'.$$

4.4 Distributivity

Addition and multiplication are related by *distributivity*:

$$a(b + c) = ab + ac \qquad \text{for every } a \text{ and } b \text{ and } c. \tag{4.4.1}$$

Here are some consequences of distributivity:

Theorem 4.4.1 $0a = 0$.

Proof To check this, look at the chain of equations,

$$\begin{aligned}0a &= 0 + 0a = ((-a) + a) + 0a\\ &= (-a) + (a + 0a) = (-a) + (1a + 0a)\\ &= (-a) + (1 + 0)a = (-a) + 1a = (-a) + a = 0.\end{aligned}$$

This is a chain of eight equations, and so needs eight reasons to be true: The reasons are (4.1.3), then (4.2.1), then (4.1.2), then (4.3.3), then (4.4.1), then (4.1.3), then (4.3.3), then (4.2.1). □

As a consequence,

Theorem 4.4.2 $(-1)a = -a$.

Proof We have

$$a + (-1)a = 1a + (-1)a = (1 + (-1))a = 0a = 0,$$

because (4.3.3), then (4.4.1), then (4.2.1), then Theorem 4.4.1. □

Combining Theorem 4.2.2 and Theorem 4.4.2 yields

$$(-1)(-1) = -(-1) = 1. \tag{4.4.2}$$

4.5 Positivity

Let $\mathbf{Z}^+$ be the set of all integers a that are *sums of ones*,[1]

$$a = 1 + 1 + \cdots + 1.$$

These are the *positive* integers. So $\mathbf{Z}^+$ contains

$$1, 1 + 1, 1 + 1 + 1, \ldots$$

To *increment* an integer a means to add 1 to it, resulting in the *increment* $a + 1$. Thus $\mathbf{Z}^+$ consists of 1 and its successive increments. Recursively, a number n is a positive number if and only if $n = 1$ or n is the increment of a positive number.

We write $a > 0$ to mean a is positive. By definition of $\mathbf{Z}^+$, the integer 1 is positive, $1 > 0$.

Let $\mathbf{Z}^-$ be the set of all negatives of the positive integers,

$$a = -(1 + 1 + \cdots + 1).$$

These are the *negative* integers. So $\mathbf{Z}^-$ contains

[1] This is explained in detail in §A.1.

$$-1, -(1+1), -(1+1+1), \dots$$

We write $a < 0$ to mean a is negative.

Be careful to distinguish between *a negative integer* and *the negative of an integer*: The negative of the integer -1 is a positive integer. More generally, a is positive implies $-a$ is negative, and a is negative implies $-a$ is positive.

The integers are *minimal*, in the sense

$$\textit{every number is positive, or negative, or zero,} \tag{4.5.1}$$

and moreover

$$\textit{zero is neither positive nor negative.} \tag{4.5.2}$$

In particular this is saying 1 and 0 are distinct integers. Since $a = a+1$ implies $0 = 1$, it follows that a and $a+1$ are distinct integers for any integer a. This last property is called *positivity*, because it forces $\mathbf{Z}^+$ and $\mathbf{Z}^-$ to be disjoint (Theorem 4.5.2).

Theorem 4.5.1 *The set* $\mathbf{Z}^+$ *of positive integers is infinite.*

Proof If $\mathbf{Z}^+$ were finite, then there would be two different sums of ones that are the same integer. But then their difference is a sum of ones that equals zero. But (4.5.2) states zero is neither positive nor negative. □

Since a sum of ones added to another sum of ones results in yet another sum of ones,

$$\textit{the sum of positive numbers is positive.} \tag{4.5.3}$$

By (4.4.1), a sum of ones multiplied with another sum of ones results in another sum of ones,

$$\textit{the product of positive numbers is positive.} \tag{4.5.4}$$

So positive times positive is positive. If $a < 0$ and $b > 0$, then $-a > 0$, so $(-a)b > 0$, so

$$ab = -(-1)(ab) = -(-a)b < 0.$$

Hence negative times positive is negative. Similarly negative times negative is positive.

The four properties (4.5.1), (4.5.2), (4.5.3), (4.5.4) are summarized by saying $\mathbf{Z}$ is totally ordered ((A.0.1)).

As a consequence, we have: For any integers a and b,

$$ab = 0 \implies a = 0 \quad \text{or} \quad b = 0. \tag{4.5.5}$$

This implies the *cancellation property*: For any integers a, b, c,

$$ab = ac \quad \text{and} \quad a \neq 0 \implies b = c. \tag{4.5.6}$$

Theorem 4.5.2 *No integer is both positive and negative:* $\mathbf{Z}^+$ *and* $\mathbf{Z}^-$ *do not intersect.*

Proof Suppose a is both positive and negative. Then by definition of negative, $a = -b$ for some positive b. By definition of $-b$, this means $a + b = 0$, which contradicts (4.5.2) and (4.5.3). Thus no integer is both positive and negative. □

The upshot is: The integers $\mathbf{Z}$ are divided into disjoint sets $\mathbf{Z}^+$, $\{0\}$, and $\mathbf{Z}^-$, whose union is $\mathbf{Z}$,

$$\mathbf{Z}^+ \cup \{0\} \cup \mathbf{Z}^- = \mathbf{Z},$$

which is the same as saying for every integer a,

$$a > 0 \qquad \text{or} \qquad a = 0 \qquad \text{or} \qquad a < 0.$$

More generally, we say $a > b$ or $b < a$ when we mean $a - b > 0$. Then $a = b + c$ with $c > 0$ is the same as saying $a > b$, and for any integers a and b,

$$a > b \qquad \text{or} \qquad a = b \qquad \text{or} \qquad a < b.$$

The *absolute value* $|a|$ of an integer a is

$$|a| = \begin{cases} a & a \geq 0, \\ -a & a < 0. \end{cases}$$

Theorem 4.5.3

$$a > b \quad \textit{and} \quad b > c \quad \Longrightarrow \quad a > c.$$

Proof $a > b$ means $a - b > 0$, $b > c$ means $b - c > 0$, and (4.5.3) implies

$$a - c = (a - b) + (b - c) > 0,$$

so $a > c$. □

Similarly,

Theorem 4.5.4 *$a > b$ and $c > d$ imply $a + c > b + d$, and $a > b$ and $c > 0$ imply $ac > bc$.*

Proof $a > b$ means $a - b > 0$, $c > d$ means $c - d > 0$, and (4.5.3) implies

$$(a + c) - (b + d) = (a - b) + (c - d) > 0,$$

so $a + c > b + d$. By (4.5.4),

$$ac - bc = (a - b)c > 0,$$

so $ac > bc$. □

We define $a \geq b$ to mean $a > b$ or $a = b$, and $a \leq b$ to mean $a < b$ or $a = b$. Then this last theorem may be rewritten

Theorem 4.5.5 *$a \geq b$ and $c \geq d$ imply $a+c \geq b+d$, and $a \geq b$ and $c \geq 0$ imply $ac \geq bc$.*

Let us show $a > b$ implies $-a < -b$. If we let $\implies$ denote *implies*, then

$$\begin{aligned} a > b &\implies a-b > 0 \\ &\implies -(a-b) < 0 \\ &\implies (-a)-(-b) < 0 \\ &\implies -a < -b. \end{aligned}$$

Theorem 4.5.6 *There are no integers between* 0 *and* 1*: For any integer a, then $a \leq 0$ or $a \geq 1$. More generally, for any integer n, there are no integers between n and $n+1$: Either $a \leq n$ or $a \geq n+1$, for every integer a.*

Proof Either $a \leq 0$ or $a > 0$. If $a > 0$, then a is a sum of one or more ones

$$a = 1+1+\cdots+1 = 1+c$$

with $c \geq 0$, so $a \geq 1$. For the general case, a is between n and $n+1$ exactly when $a-n$ is between 0 and 1. But there are no integers between 0 and 1, so there are no integers between n and $n+1$. □

4.6 Division

We say b *divides* a if $a = bc$, where a, b, and c are integers. In this case we write $a/b = c$ and we call c the *quotient*. For example, the *even* integers are the integers divisible by 2, and the *odd* integers are the integers that are not.

The simplest case is when $a = 1$: We say an integer c is a *reciprocal* of an integer b if

$$bc = cb = 1.$$

Remember both b and c are integers. So for example, 1 is a reciprocal of 1, since $1 \cdot 1 = 1$. Also zero does not have a reciprocal, since $0b = 0$ is never equal to 1. Finally below (4.4.2) shows -1 is the reciprocal of -1.

Could an integer have more than a single reciprocal? Let's show the answer is no.

Theorem 4.6.1 *Every integer has at most one reciprocal.*

Proof Suppose a is an integer and both b and c are reciprocals of a, so we are given $ab = 1$ and $ac = 1$. By (4.1.1), we then have $ca = 1$, so

$$b = 1b = (ca)b = c(ab) = c1 = c.$$

Here we used (4.3.3), followed by $ca = 1$, followed by (4.3.2), followed by the given $ab = 1$, followed by (4.3.3). So $b = c$. □

So each integer a has at most one reciprocal, which we always write as $1/a$. Not every integer has a reciprocal, for example 0 doesn't have one. What other integers have reciprocals?

A *unit* is an integer that has a reciprocal. The most important result about reciprocals is

Theorem 4.6.2 *The only units are* ± 1. *If* $a \neq \pm 1$, *then* a *does not have a reciprocal.*

Proof To see why, if $a \neq \pm 1$, then $a = 0$ or $a > 1$ or $a < -1$. We already know $a = 0$ does not have a reciprocal. Suppose now $a > 1$. Since positive times negative is negative and 1 is positive, a reciprocal b of a must be positive. Thus $b \geq 1$ hence

$$ab > 1b = b \geq 1,$$

so ab can't equal 1, so b is not a reciprocal of a. Since b was any positive integer, a does not have a reciprocal. Now suppose $a < -1$. If a had a reciprocal b, then $-a$ would have a reciprocal $-b$, since $1 = ab = (-a)(-b)$. But $-a > 1$, so this can't happen, so a doesn't have a reciprocal. □

If b has a reciprocal, then we may write the quotient

$$a/b = a \cdot \frac{1}{b}$$

as the product of a and the reciprocal of b. Even if b does not have a reciprocal, integers may sometimes be divided by b. For example, even though there is no integer $1/2$, you can divide 6 by 2 since $6/2 = 3$, but can't divide 6 by 4: There is no integer c satisfying $6 = 4c$. Exactly when we can or can't divide is taken up in §5.1.

To summarize, Theorem 4.6.2 says you can divide every integer by a only when $a = \pm 1$. You can't divide any integer by 0, you can't divide every integer by 2, but you can sometimes divide by 2; you can't divide every integer by 3, but you can sometimes divide by 3, etc.

4.7 Induction

Let S be a set of integers, so $S \subset \mathbf{Z}$. We say S is *inductive* if

1. S contains 1, and
2. S contains the increment $n + 1$ of each n in S.

Given what we've seen of Python, it may better to call such a set *recursive*. However, we stick to the standard terminology.

In §4.5, $\mathbf{Z}^+$ was defined as the set of all sums of ones. Since the increment of a sum of ones is a sum of ones, $\mathbf{Z}^+$ is inductive. In fact, $\mathbf{Z}^+$ is the *smallest* inductive subset of $\mathbf{Z}$: If S is any inductive subset of $\mathbf{Z}$, then S contains all sums of ones, hence

S contains $\mathbf{Z}^+$. Thus an equivalent definition is: $\mathbf{Z}^+$ *is the smallest inductive subset of* $\mathbf{Z}$.

As an immediate consequence of this, we have the *Principle of Induction*:

Let $P(n)$ be a statement that depends on n. If

1. *$P(1)$ is valid, and*
2. *$P(n+1)$ is valid whenever $P(n)$ is valid,*

then $P(n)$ is valid for all positive integers $n \geq 1$.

Let a be an integer. One can show (§A.1) there is a unique map f from $\mathbf{Z}^+$ to $\mathbf{Z}$ satisfying

$$f(1) = a, \qquad f(n+1) = f(n)\,a, \quad n \geq 1.$$

This map is the *power* map of a, and is written $f(n) = a^n$. Then $a^1 = a$ and

$$a^{n+1} = a^n a, \qquad n \geq 1.$$

One can also show (§A.1) there is a unique map g from $\mathbf{Z}^+$ to $\mathbf{Z}$ satisfying

$$g(1) = 1 + a, \qquad g(n+1) = g(n) + a^{n+1}, \quad n \geq 1.$$

This map is the *geometric sum* map, and is written

$$g(n) = 1 + a + \cdots + a^n = \sum_{k=0}^{n} a^k.$$

The existence of the power map f and the geometric sum map g is established after Theorem A.1.4.

Then we can derive

$$a^{n+1} - 1 = (a-1)(a^n + a^{n-1} + \cdots + a + 1), \qquad n \geq 1, \tag{4.7.1}$$

by induction, as follows. Writing (4.7.1) in terms of f and g, we have to show

$$f(n+1) - 1 = (a-1)g(n) \tag{4.7.2}$$

for $n \geq 1$. For $n = 1$, (4.7.2) is valid since

$$f(2) - 1 = a^2 - 1 = (a-1)(a+1) = (a-1)g(1).$$

Assume (4.7.2) is valid. Then

$$\begin{aligned}
f(n+2) - 1 &= f(n+1)a - 1 \\
&= f(n+1)a - f(n+1) + f(n+1) - 1 \\
&= (a-1)f(n+1) + (a-1)g(n) \\
&= (a-1)(f(n+1) + g(n)) = (a-1)g(n+1).
\end{aligned}$$

By induction, this establishes (4.7.2), hence (4.7.1). The identity (4.7.1) was used in (2.6.1).

Let S be a set of positive integers. For example, S may be the set of integers between 10 and 20, or the set of even integers. We say a is the *minimum* of S if

1. a is in S,
2. $b \geq a$ for all b in S.

When a is the minimum of S, we write $a = \min S$. For example, $1 = \min \mathbf{Z}^+$, since $a \geq 1$ when $a > 0$.

Theorem 4.7.1 (Well-Ordering Principle) *Every nonempty set S of positive integers has a minimum.*

Proof Assume S has no minimum. Since S is nonempty, we may select some n_0 in S. Since S has no minimum, there is n_1 in S satisfying $n_1 < n_0$. By Theorem 4.5.6, this is the same as saying $n_1 \leq n_0 - 1$. Since S has no minimum, there is n_2 in S satisfying $n_2 < n_1$. By Theorem 4.5.6, this is the same as saying $n_2 \leq n_1 - 1$, which in turn implies $n_2 \leq n_0 - 2$. Since S has no minimum, there is n_3 in S satisfying $n_3 \leq n_2 - 1 \leq n_0 - 3$. Continuing in this manner, we obtain a sequence of positive integers $n_0, n_1, n_2, n_3, \dots$ satisfying

$$n_k \leq n_0 - k, \qquad k \geq 1.$$

But this can't happen, because for $k = n_0+1$, this says $n_k \leq -1 < 0$, which contradicts the positivity of n_k, because an integer can't be both positive and negative (Theorem 4.5.2). Hence S has a minimum. □

Since modular integers (Chapter 6) are both positive and negative, the downward chain $0, -1, -2, -3, \dots$ consists of positive integers and has no minimum. Thus Theorem 4.7.1 does not hold for modular integers.

If S is a set of integers, we say $a = \max S$ if a is in S and $a \geq k$ for all k in S. We say S is *bounded above* if there is some integer m satisfying $m \geq k$ for all k in S. Exercise 4.8 shows that a set that is bounded above has a max.

Exercises

Exercise 4.1 Show that negative times negative is positive.

Exercise 4.2 Show (4.5.5).

Exercise 4.3 Show the cancellation property (4.5.6).

Exercise 4.4 Suppose a and b are integers with $|a| < b$. Suppose b divides a, so $a = bc$ for some integer c. Show that $c = a = 0$.

Exercise 4.5 Suppose a and b are integers with $b > 0$. Let S be the set of positive integers $k \geq 1$ satisfying $a - kb < 0$. Show that S is not empty.

Exercise 4.6 Continuing the previous exercise, let $q = \min S - 1$. Then show $qb \leq a < (q + 1)b$.

Exercise 4.7 Let S be a nonempty set of integers, and let m be an integer. Suppose S is *bounded below* by m. This means $k \geq m$ for all k in S. Show that $\min S$ exists.

Exercise 4.8 Let S be a nonempty set of integers, and let m be an integer. Suppose S is *bounded above* by m. This means $k \leq m$ for all k in S. Show that $\max S$ exists.

Chapter 5
Prime Numbers

5.1 Divisibility

If $c = ab = ba$, then we say a is a *factor* of c, or c is a *multiple* of a, or *a divides c*, or c is *divisible* by a. For example, 15 is a multiple of 3, 16 is not a factor of 4 nor of 29, 3 divides 12, but 17 is not divisible by 3. Since $0 = 0b$ for any integer b, *every integer divides zero*.

A simple but important fact is

Theorem 5.1.1 *If d divides a and b, then d divides $a + b$ and $a - b$.*

Proof Suppose d divides a and b. Then $a = a'd$ and $b = b'd$, so by distributivity,

$$a + b = a'd + b'd = (a' + b')d,$$

so d divides $a + b$. Since $a - b = a + (-b)$, and d divides b if and only if d divides $-b$, d also divides $a - b$. □

By taking $a = 0$, we see *a divides b if and only if a divides $-b$*.

Recall the binomial coefficients. These are

$$\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k!}, \qquad 1 \le k \le n.$$

Since these are integers, $k!$ divides $n(n-1)\dots(n-k+1)$, which is a product of k consecutive positive integers.

Theorem 5.1.2 *The product of k consecutive positive integers is divisible by $k!$.*

Now take the positive integers greater than one: $2, 3, 4, \dots$ and start multiplying them in all possible ways

$$2\cdot 2, 2\cdot 3, 2\cdot 4, 3\cdot 3, 2\cdot 5, 2\cdot 6, \dots$$

We get

$$4, 6, 8, 9, 10, 12, \dots$$

If we multiply all possibilities, do we end up with *all* integers greater than one, or are some integers missing? Are there integers that won't appear in the resulting list? Yes, these integers

$$2, 3, 5, 7, 11, 13, \dots$$

are missing. These are the positive primes.

An integer a is *composite* if $a = bc$ for other integers b and c, neither equal to ± 1 ($\pm$ means plus-or-minus). For example, $-6 = (-2)3$ is composite. An integer a is *prime* or is a *prime number* if a is not composite. We stress that a prime may be negative: -5 is a prime.

Let's show 5 is a prime number. If 5 were composite, then $5 = ab$ with both $a > 1$ and $b > 1$, so a and b equal 2, 3, or 4. If a and b are both 2, then $ab = 4 \neq 5$. If $a \geq 2$ and $b \geq 3$, then $ab \geq 23 = 6 \neq 5$. Hence in no case is $ab = 5$. Thus 5 is prime. If -5 were composite, $-5 = bc$, then $5 = (-b)c$ is composite. Hence -5 is also prime.

Theorem 5.1.3 *There are infinitely many prime numbers.*

Proof Argue by contradiction. If there were only finitely many positive primes, call them $a, b, c, \dots$, then

$$n = 1 + abc \dots$$

cannot be divided by any of $a, b, c, \dots$, so n must be a positive prime. This contradicts the assumption that $a, b, c, \dots$ were all the positive primes. □

The *fundamental theorem of arithmetic* (§5.5) states that every integer other than ± 1 is a product of prime numbers, and, up to ordering, in a unique manner. *Factoring* an integer n means displaying its prime factors. Thus $12 = 2 \cdot 2 \cdot 3$ so `[2,2,3]` is the *prime factorization* of 12. Similarly, `[3, 3, 3607, 3803]` is the prime factorization of 123456789.

A basic problem is to find an efficient factoring procedure for a given integer n. While there are simple solutions to this problem for moderately sized integers, they quickly become unworkable for large integers.

The simplest procedure, for n positive, is to try all integers $1 < p < n$. If p divides n, and we're checking integers in order from smallest to largest, then p is a prime factor of n, and we have reduced the problem to factoring $n//p$. This is a recursive procedure. Since divisibility of n by p is the same as `n%p == 0`, the code is

```
0>>  def factor(n):
1>>     for p in range(2,n):
2>>        if n%p == 0:
3>>           return [p] + factor(n//p)
1>>     return [n]
```

This function returns a `list`. It starts by checking whether 2 divides n. If so, the functions prepends `[2]` to `factor(n//2)`. Otherwise, checks whether 3 divides n, and so on. If the list is still empty at the end of the loop, n is prime and the function returns `[n]`.

Much better codes can factor around 80 digit numbers in a reasonable time. But the best current codes together with the fastest computers cannot factor generic 400 digit numbers in fewer than hundreds of years.

5.2 Division Algorithm

Let a and b be integers with $b \geq 1$. The *division algorithm* states there is an integer q and and an integer r satisfying

$$a = qb + r, \qquad 0 \leq r < b. \tag{5.2.1}$$

For example, $a = 9$ and $b = 2$ implies $q = 4$ and $r = 1$, since $9 = 4 \cdot 2 + 1$, and $a = 7$ and $b = 10$ implies $q = 0$ and $r = 7$, since $7 = 0 \cdot 10 + 7$. The integers q and r are the *quotient* and the *remainder*.

To see why such a q and r must exist, look at the arithmetic sequence

$$\dots, a - 3b, a - 2b, a - b, a, a - b, a - 2b, a - 3b, \dots$$

We claim that this sequence is eventually negative: There is a positive multiple kb of b such that $a - kb$ is negative. If $a \geq 0$, choosing $k = a + 1$ works, since $b \geq 1$ and $a - (a + 1)b \leq a - a - 1 < 0$. If $a < 0$, choosing $k = 0$ works, since $a - 0b = a < 0$. Therefore in either case, the sequence is eventually negative.

Let's focus on the least integer k for which the corresponding term $a - kb$ in this sequence is negative. More specifically, let

$$S = \{k \text{ in } \mathbf{Z} : a - kb < 0\}.$$

We just saw S is nonempty.

We claim S is bounded below. If $a \geq 0$, then k in S implies $kb > a \geq 0$ hence $k > 0$. If $a < 0$, then $a \geq ab$, so k in S implies $kb > a \geq ab$, hence $k > a$. Hence, in either case, S is bounded below.

By Exercise 4.7, $\min S$ exists, so we may set $q + 1 = \min S$ and $r = a - qb$. By definition of S, $a-(q+1)b < 0$ or $r = a-qb < b$. Since q is not in S, $r = a-qb \geq 0$. We conclude q and r satisfy (5.2.1).

Moreover, the integers q and r are unique. To see why, let q' and r' be integers satisfying

$$a = q'b + r', \qquad 0 \leq r' < b. \tag{5.2.2}$$

and suppose $r' \geq r$. Then both r and r' are between 0 and b, so either $0 \leq r' - r < b$ or $0 \leq r - r' < b$. We may safely assume $0 \leq r' - r < b$, otherwise we switch the roles of r and r'. Now subtract (5.2.1) from (5.2.2). We get

$$b(q - q') = (r' - r), \tag{5.2.3}$$

so b divides $r' - r$. But $0 \leq r' - r < b$, so we must have $r' = r$ (Exercise 4.4). Inserting this into (5.2.3), by Exercise 4.5.5, we obtain $q = q'$.

We have derived the

Theorem 5.2.1 (Division Algorithm) *Let a and b be integers with $b \geq 1$. Then there is a unique quotient q and remainder r satisfying* (5.2.1).

5.3 Euclidean Algorithm

Now start with any integers a_0 and a_1 with $a_1 \geq 1$. Apply the division algorithm to obtain a quotient q_1 and remainder a_2 satisfying

$$a_0 = a_1 q_1 + a_2, \qquad 0 \leq a_2 < a_1.$$

If $a_2 = 0$, stop. Otherwise, apply the division algorithm again with a_1 and a_2 to obtain a quotient q_2 and remainder a_3 satisfying

$$a_1 = a_2 q_2 + a_3, \qquad 0 \leq a_3 < a_2.$$

If $a_3 = 0$, stop. Otherwise, apply the division algorithm again with a_2 and a_3 to obtain a quotient q_3 and remainder a_4 satisfying

$$a_2 = a_3 q_3 + a_4, \qquad 0 \leq a_4 < a_3.$$

Continuing in this way, we obtain a finite decreasing sequence $a_1 > a_2 > a_3 > a_4 > \dots$ of positive integers. We end up with

$$\begin{aligned} a_0 &= a_1 q_1 + a_2 \\ a_1 &= a_2 q_2 + a_3 \\ a_2 &= a_3 q_3 + a_4 \\ &\dots\dots \\ a_{n-2} &= a_{n-1} q_{n-1} + a_n \\ a_{n-1} &= a_n q_n + 0, \end{aligned} \tag{5.3.1}$$

after n steps. This is the *euclidean algorithm*.

For example, if $a_0 = 33$ and $a_1 = 10$, then $a_2 = 3$ and $a_3 = 1$, so the sequence `[33,10,3,1]` stops in 3 steps. If $a_0 = 98765423232121332$ and $a_1 = 123453678912$, the sequence

```
[98765423232121332, 123453678912, 11028943092, 2135304900,
    352418592, 20793348, 19725024, 1068324, 495192, 77940,
    27552, 22836, 4716, 3972, 744, 252, 240, 12]
```

stops in 17 steps.

The code returning these results is

```
0>>  def euclid(a,b):
1>>     r = a%b
1>>     if r == 0:
2>>        return [(a,b)]
1>>     else:
2>>        return [a] + euclid(b,r)
```

5.4 GCD

If a and b are integers, a *common divisor* is an integer d that divides both a and b. For example, 3 is a common divisor of 6 and -9, but is not a common divisor of 6 and 10. There may be several common divisors. Among the positive common divisors, there is a greatest. This is the *greatest common divisor* or $\gcd(a, b)$.

For example, $\gcd(a, 0) = a$ and $\gcd(0, b) = b$. If $d = \gcd(a, b)$, then $-d = \gcd(a, b)$. To make $\gcd(a, b)$ unique, we always take the positive greatest common divisor as the gcd of a and b. Then

$$\gcd(\pm a, \pm b) = \gcd(a, b).$$

Theorem 5.4.1 *If a_0 and a_1 are integers with $a_1 \geq 1$ and the euclidean algorithm stops after n steps, then $a_n = \gcd(a_0, a_1)$.*

Proof Look at the sequence of equations in the euclidean algorithm. If d is a common divisor of a_0 and a_1, then from the first equation d divides a_2, hence is a common divisor of a_1 and a_2. Then from the second equation, d is a common divisor of a_2 and a_3. Proceeding forward in this manner, we conclude d divides a_n.

On the other hand, from the last equation, a_n divides a_{n-1}. From the next-to-last equation, a_n divides a_{n-1} and a_{n-2}. Proceeding backward, we conclude a_n divides a_0 and a_1, hence a_n is a common divisor of a_0 and a_1. □

For example, from above,

$$\gcd(33, 10) = 1,$$

and

$$\gcd(98765423232121332, 123453678912) = 12.$$

Based on this Theorem, we have

```
0>>  def gcd(a,b):
1>>     return euclid(a,b)[-1]
```

We say an integer g is a *linear combination* of integers a and b if there are integers s and t satisfying

$$sa + tb = g. \tag{5.4.1}$$

For example, 1 is a linear combination of 5 and 11 since $1 = 45-44 = 10\cdot 5+(-4)\cdot 11$, and -3 is a linear combination of 12 and -21 since $-3 = -24 + 21 = (-2)\cdot 12 + (-1)\cdot(-21)$.

An important consequence is

Theorem 5.4.2 *$g = \gcd(a, b)$ is a linear combination of a and b.*

Proof If $b = 0$, this is clear. If $b \geq 1$, look at the equations (5.3.1) with $a = a_0$ and $b = a_1$. Since $a_2 = a_0 - q_1 a_1$, a_2 is a linear combination of a and b. Since $a_3 = a_1 - q_2 a_2$, a_3 is a linear combination of a and b. Continuing in this manner, $g = a_n$ is a linear combination of a and b. □

A consequence of this is

Theorem 5.4.3 *If a prime p divides ab, then p divides a or p divides b.*

Proof Either p divides a or not. If p does not divide a, then $\gcd(a, p) = 1$, so 1 is a linear combination of a and p: there are integers s and t satisfying

$$sa + tp = 1.$$

Muliplying by b, we get

$$s(ab) + tpb = b.$$

Now p divides ab, so p divides the left side, hence p divides the right side, which is b. □

If p is prime and a is any integer, there are two possibilities. Either p divides a, or p does not divide a. Since p only has divisors 1 and p, in the former case, $\gcd(a, p) = p$, while in the latter case, $\gcd(a, p) = 1$.

For later, it is important to compute (s, t) is (5.4.1) in terms of a, b, and g. For this, with notation as in the Euclidean algorithm, apply Theorem 5.4.2 to (a_0, a_1), (a_1, a_2),... to get the *extended euclidean algorithm*

$$\begin{aligned}
a_0 &= a_1 q_1 + a_2 & s_0 a_0 + t_0 a_1 &= g\\
a_1 &= a_2 q_2 + a_3 & s_1 a_1 + t_1 a_2 &= g\\
a_2 &= a_3 q_3 + a_4 & s_2 a_2 + t_2 a_3 &= g\\
&\dots & &\dots\\
a_{n-2} &= a_{n-1} q_{n-1} + a_n & s_{n-2} a_{n-2} + t_{n-2} a_{n-1} &= g\\
a_{n-1} &= a_n q_n + 0, & s_{n-1} a_{n-1} + t_{n-1} a_n &= g
\end{aligned}$$

At each stage, the choices of coefficients s_0, t_0, s_1, t_1, ... are not unique. We seek a consistent choice of coefficients across these equations. Substituting the first equation the left column into the first equation of the right column yields $s_0(a_1 q_1 + a_2) + t_0 a_1 = g$, or $(s_0 q_1 + t_0) a_1 + s_0 a_2 = g$. This suggests we insist

$$(s_0 q_1 + t_0, s_0) = (s_1, t_1) \quad \text{or } (s_0, t_0) = (t_1, s_1 - t_1 q_1).$$

Repeating this with the second equations, then third, and so on, we get

$$\begin{aligned}(s_0, t_0) &= (t_1, s_1 - t_1 q_1)\\ (s_1, t_1) &= (t_2, s_2 - t_2 q_2)\\ (s_2, t_2) &= (t_3, s_3 - t_3 q_3)\\ &\cdots\cdots\\ (s_{n-1}, t_{n-1}) &= (0, 1),\end{aligned} \tag{5.4.2}$$

where the last equation follows from $a_n = g$. This yields a recursive procedure for computing (s_0, t_0) (Exercise 5.5).

5.5 Fundamental Theorem of Arithmetic

Recall an integer a is composite if $a = bc$ for some integers b and c not equal to 1. For example, 12 is composite in two different ways, since $12 = 2 \cdot 6$ and $12 = 3 \cdot 4$. Since $6 = 2 \cdot 3$ and $4 = 2 \cdot 2$ are composite, we may write

$$12 = 2 \cdot 2 \cdot 3 = 2 \cdot 3 \cdot 2 = 3 \cdot 2 \cdot 2$$

as a product of primes. Thus, apart from the order, and apart from signs ± 1, 12 can be written as a product of primes in only one way.

The fact that this is true in general is the

Theorem 5.5.1 (Fundamental Theorem of Arithmetic) *Every positive integer $n > 1$ is a product of positive primes,*

$$n = p_1 p_2 \ldots, \tag{5.5.1}$$

in a unique manner, except for a change in order.

(5.5.1) is the *prime factorization* of n.

Proof Suppose there is a positive integer greater than 1 which cannot be written as a product of primes, and let n be the least such positive integer (Theorem 4.7.1). Then $n > 1$ and n cannot be prime, otherwise n is trivially a product of primes. Thus $n = bc$ for some $1 < b < n$ and $1 < c < n$. Since n was chosen to be the least positive integer not having a prime factorization, both b and c must be expressible as products of primes. Since $n = bc$, this implies n is so expressible, contradicting the choice of n. Thus there is no such n. In other words, every positive integer n greater than 1 is a product of primes.

Now we establish uniqueness. Suppose an integer is expressed as a product of primes in two different ways,

$$n = p_1 p_2 \cdots = q_1 q_2 \ldots .$$

Since p_1 divides the left side, p_1 divides the right side. By Theorem 5.4.3, p_1 must divide one of the q's. By re-ordering the q's, we may assume this q is q_1. By Exercise 5.7, $p_1 = q_1$. By the cancellation property (4.5.6), cancelling $p_1 = q_1$ from both sides,

$$p_2 p_3 \cdots = q_2 q_3 \ldots.$$

Repeating the same logic with p_2, we may cancel $p_2 = q_2$ to obtain

$$p_3 p_4 \cdots = q_3 q_3 \ldots.$$

Repeating again, we may keep cancelling until the there are no primes remaining on the left side. But when this happens, there can be no more primes remaining on the right side. Hence the p's are equal to the q's, except possibly for the order. □

An immediate consequence is

Theorem 5.5.2 *If $p_1, p_2, \ldots, p_n$ are* distinct *primes dividing a, then the product $p_1 p_2 \ldots p_n$ divides a.*

Integers a and b are *relatively prime* if they have no common divisor, $\gcd(a, b) = 1$. Let a be a positive integer. How many integers in

$$\{0, 1, 2, \ldots, n-1\}$$

are relatively prime to a? By definition, the number of such integers is $\phi(a)$. This is the Euler *ϕ-function*.

For example, 1, 5, 7, and 11 are relatively prime to 12, so $\phi(12) = 4$. Also, if p is prime, any positive integer less than p is relatively prime to p, so $\phi(p) = p - 1$.

We use the inclusion-exclusion principle (3.3.5) to find a formula for $\phi(a)$.

Let a be a positive integer, and let p be a prime dividing a. For example, if $a = 12 = 2 \cdot 2 \cdot 3$, then $p = 2$ or $p = 3$. Let A be the set of integers b, $0 \le b < a$, that are divisible by p. Then A consists of the multiples $0p, 1p, 2p, \ldots,$ of p less than a, so $|A| = a/p$.

If p and q are distinct primes dividing a, then pq divides a. If A is the set of integers b, $0 \le b < a$, that are divisible by pq, then, by the same reasoning as above, $|A| = a/pq$.

Now let $p, q, r, \ldots$ be the *distinct* prime factors of a, and let

$$A_p = \{b : 0 \le b < a \text{ and } b \text{ is a multiple of } p\}.$$

By what we just discussed, $|A_p| = a/p$, $|A_p \cap A_q| = a/pq$, $|A_p \cap A_q \cap A_r| = a/pqr$, etc. Hence by the inclusion-exclusion principle,

$$|A_p \cup A_q \cup A_r \cup \ldots| = \sum \frac{a}{p} - \sum \frac{a}{pq} + \sum \frac{a}{pqr} - \ldots$$

Now $A_p \cup A_q \cup A_r \cup \ldots$ is the set of integers b, $0 \le b < a$, that have a common factor with a, so its complement $(A_p \cup A_q \cup A_r \cup \ldots)^c$ in $\{0, 1, 2, \ldots, a-1\}$ are the integers b, $0 \le b < a$, that are relatively prime to a, and

$$\begin{aligned}\left|(A_p \cup A_q \cup A_r \cup \dots)^c\right| &= a - |A_p \cup A_q \cup A_r \cup \dots| \\ &= a - \sum \frac{a}{p} + \sum \frac{a}{pq} - \sum \frac{a}{pqr} + \dots \\ &= a\left(1 - \sum \frac{1}{p} + \sum \frac{1}{pq} - \sum \frac{1}{pqr} - \dots\right).\end{aligned} \tag{5.5.2}$$

But, recalling the elementary symmetric polynomials and (2.5.5), we have (insert $t_1 = 1/p, t_2 = 1/q, \dots$)

$$\prod_p \left(1 - \frac{1}{p}\right) = 1 - \sum \frac{a}{p} + \sum \frac{a}{pq} - \sum \frac{a}{pqr} + \dots.$$

Hence we have shown

Theorem 5.5.3 *The number of integers b, $0 \le b < a$, relatively prime to a is*

$$\phi(a) = a \prod_p \left(1 - \frac{1}{p}\right),$$

where the product is over distinct prime factors of a.

For example,

$$\phi(pq) = pq\left(1 - \frac{1}{p}\right)\left(1 - \frac{1}{q}\right) = (p-1)(q-1),$$

and $a = 12$ implies $p = 2$ and $q = 3$, so

$$\phi(12) = 12\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right) = 4.$$

Exercises

Exercise 5.1 Use (4.7.1) to show $2^n - 1$ is composite when n is composite. Conclude primes of the form $2^n - 1$ must have n prime. Such primes are *Mersenne primes*..

Exercise 5.2 Use (4.7.1) to show $a^n + 1$ is composite when n is odd. Conclude if $2^n + 1$ is prime, then $n = 2^k$ is a power of two. Such primes are *Fermat primes*..

Exercise 5.3 Write a function `div(a,b)` that returns q and r.

Exercise 5.4 Write a function `gcd(a,b)` that returns $\gcd(a, b)$ without using `euclid(a,b)`.

Exercise 5.5 Write a function `exteuclid(a,b)` that returns `(s,t)` satisfying (5.4.1), in a manner similar to `gcd(a,b)`.

Exercise 5.6 If `a` is a list, `p = a.pop(n)` removes `a[n]` from `a` and assigns it to `p`. `a.remove(b)` removes `b` from `a`. The *sieve of Eratosthenes* is

```
0>>  def primes(n):
1>>     sieve = list(range(2,n))
1>>     sifted = [ ]
1>>     while sieve:
2>>        p = sieve.pop(0)
2>>        sifted.append(p)
2>>        for n in sieve:
3>>           if n%p == 0:
4>>              sieve.remove(n)
1>>     return sifted
```

Explain why this code returns all primes less than n.

Exercise 5.7 If p and q are primes and p divides q, then $p = q$.

Exercise 5.8 If p divides a product $a_1 a_2 \dots a_n$, then p divides at least one of the factors a_i (use induction on n and Theorem 5.4.3).

Exercise 5.9 If p and q are distinct primes both dividing an integer a, then the product pq divides a.

Exercise 5.10 If the fundamental theorem is true for n, then it's true for $-n$.

Exercise 5.11 If a and b are relatively prime, then $\phi(ab) = \phi(a)\phi(b)$. Conclude $\phi(pq) = (p-1)(q-1)$.

Exercise 5.12 Write a function `eulerphi(n)` that returns $\phi(n)$.

Chapter 6
Modular Integers $\mathbf{Z}_n$

Recall the axioms for the integers $\mathbf{Z}$ (Chapter 4). In this Chapter we study a different kind of arithmetic, *modular arithmetic*. Here the numbers satisfy the same axioms except for positivity (4.5.2), which is replaced by its negation.

To explain this, recall positivity states *zero is neither positive nor negative*, where (§4.5) the positive integers are by definition the sums of ones, $1 + 1 + \cdots + 1$. The *negation* of this is *zero is either positive or negative*. Since $-0 = 0$, this is the same as saying *zero is both positive and negative*.

Now 0 *is positive* means 0 is a sum of ones,

$$0 = 1 + 1 + \cdots + 1.$$

Modular arithmetic depends on the number of ones that add up to zero. If the least number of ones that add up to zero is $n \geq 1$, we call n the *modulus*, and we obtain the *modular integers* $\mathbf{Z}_n$. Therefore the properties that characterize $\mathbf{Z}_n$ are

1. additive commutativity (4.1.1)
2. additive associativity (4.1.2)
3. existence of zero (4.1.3)
4. existence of negatives (4.2.1)
5. multiplicative commutativity (4.3.1)
6. multiplicative associativity (4.3.2)
7. existence of one (4.3.3)
8. distributivity (4.4.1)
9. minimality (4.5.1)
10. zero is the sum of n ones, and not fewer

In §A.2, we show there is, for each $n \geq 1$, essentially one set $\mathbf{Z}_n$ equipped with addition and multiplication operations satisfying these axioms.

Our grade school intuition about integers assumes zero is not positive. Since positivity of zero is completely at odds with our grade school intuition, we have to adjust our expectations when studying modular integers, where things behave very differently.

By negating a single axiom, one is led to other kinds of arithmetics that are self-consistent and useful.[1]

In $\mathbf{Z}_1$, $1 = 0$, and the set $\mathbf{Z}_1$ consists of one element. We are only interested in situations where $1 \neq 0$, so we only look at $\mathbf{Z}_n$, $n \geq 2$.

In this chapter, we will see, for each $n \geq 2$, $\mathbf{Z}_n$ has its own distinct arithmetic that is self-consistent and useful.

6.1 Addition, Multiplication

Let's take modulus $n = 5$ for example. Then $\mathbf{Z}_5$ is the set of numbers satisfying axioms 1 through 9 together with $5 = 1 + 1 + 1 + 1 + 1 = 0$, the sum of five ones equals zero. So the set $\mathbf{Z}_5$ includes these six numbers 0, 1, $2 = 1 + 1$, $3 = 2 + 1$, $4 = 3 + 1$, and $5 = 4 + 1$. But $5 = 0$, because 5 is the sum of five ones, so these are five numbers, not six. In fact, the only numbers in $\mathbf{Z}_5$ are 0, 1, 2, 3, 4,

$$\mathbf{Z}_5 = \{0, 1, 2, 3, 4\}.$$

Indeed, if we add seven ones, we end up with 2 as an answer, since five ones add up to zero,

$$1 + 1 + 1 + 1 + 1 + 1 + 1 = 7 = 5 + 2 = 0 + 2 = 2.$$

Therefore $3 + 4 = 2$. Hence addition in $\mathbf{Z}_5$ is determined by the following rule: If a and b are in $\mathbf{Z}_5$, then $a + b$ is obtained by adding them the usual way (in $\mathbf{Z}$), then taking the remainder after division by 5.

Be careful, these are not integers, they are numbers of a different sort, since $5 = 0$. To emphasize this difference, these numbers are called *modular integers*.

If a is an integer,[2] the *residue of a mod n* is the remainder after division by n. With the above rule, we can build the addition table for $\mathbf{Z}_5$ (Figure 6.1).

There are only five numbers in $\mathbf{Z}_5$. It's not that there is no 5 in $\mathbf{Z}_5$. There is a 5 in $\mathbf{Z}_5$, it's the sum of five ones, and it equals 0. It's not that there is no -1 in $\mathbf{Z}_5$. There is a -1 in $\mathbf{Z}_5$, it's the sum of four ones, since $4 + 1 = 0$, so $-1 = 4$. So we could have written just as well

$$\mathbf{Z}_5 = \{5, 6, 7, 8, 9\},$$

[1] This situation is entirely analogous to that of Euclid's axioms for geometry. There, a single axiom, the *parallel postulate*, is singled out. By negating this axiom, one is led to other kinds of geometries that are self-consistent and useful.

[2] *Integer* is an element of $\mathbf{Z}$, and *modular integer* is an element of $\mathbf{Z}_2$ or $\mathbf{Z}_3$ or $\mathbf{Z}_4$, . . .

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Fig. 6.1 Addition in $\mathbf{Z}_5$.

or, for that matter,

$$\mathbf{Z}_5 = \{5, 11, 2, -2, -1\},$$

as long as we enforce $-\mathbf{J}_5$: $5 = 0$. Practically, however, we always write the residues

$$\mathbf{Z}_n = \{0, 1, 2, \dots, n-1\}$$

as that is simplest. It helps to visualize the numbers in $\mathbf{Z}_5$ as five points equally spaced on a circle, like the hours on a clock face. Then addition $3 + 4$ corresponds to going seven steps around the circle, which gets one back to 2.

Turning to multiplication, in $\mathbf{Z}_5$, $2 \cdot 3 = 6$, but $6 = 1$, so $2 \cdot 3 = 1$. Similarly, $4 \cdot 4 = 16 = 1$. Hence multiplication in $\mathbf{Z}_5$ is determined by the following rule: If a and b are in $\mathbf{Z}_5$, then ab is obtained by multiplying them the usual way (in $\mathbf{Z}$), then taking the remainder $(ab)\%5$ after division by 5. Here is the resulting multiplication table for $\mathbf{Z}_5$ (Figure 6.2).

×	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Fig. 6.2 Multiplication in $\mathbf{Z}_5$.

According to **D**, each a in $\mathbf{Z}_5$ has a negative $-a$, characterized by $a + (-a) = 0$. So $-0 = 0$, since $0+0 = 0$, $-1 = 4$, since $1+4 = 0$, $-2 = 3$, since $2+3 = 0$, $-3 = 2$ since $3+2 = 0$, and $-4 = 1$, since $4+1 = 0$. Here is the negative table for $\mathbf{Z}_5$ (Figure 6.3).

Turning now to positivity, recall $\mathbf{Z}^+$ was defined as the set of sums of ones. But in $\mathbf{Z}_5$, every number is positive. By the negative table, every number is also negative.

a	$-a$
0	0
1	4
2	3
3	2
4	1

Fig. 6.3 Negatives in $\mathbf{Z}_5$.

Thus $\mathbf{Z}_5 = \mathbf{Z}_5^+ = \mathbf{Z}_5^-$. Therefore we have $a < 0 < a$ for any a in $\mathbf{Z}_5$, so we cannot compare numbers in $\mathbf{Z}_5$: it is meaningless to say $a < b$.

Nevertheless, induction is still valid in $\mathbf{Z}_n$. It's not invalid, it's just not useful, because $\mathbf{Z}_n$ is a finite set. But anything that depended on positivity breaks down. For example, in $\mathbf{Z}_n$, Theorem 4.7.1 is not valid, and Theorem 4.6.2 is not valid.

To explain this, let a be in $\mathbf{Z}_n$. As before in $\mathbf{Z}$, the *reciprocal* of a modular integer a is a modular integer b in $\mathbf{Z}_n$ satisfying $ab = 1$ in $\mathbf{Z}_n$. We denote the reciprocal of a as $1/a$. For example, the reciprocal of 3 is 2 in $\mathbf{Z}_5$, since $3 \cdot 2 = 6 \equiv 1 \mod 5$. We write $1/3 = 2$, and we say 3 is a *unit* in $\mathbf{Z}_5$. Here is the reciprocal table for $\mathbf{Z}_5$ (Figure 6.4).

a	$1/a$
0	None
1	1
2	3
3	2
4	4

Fig. 6.4 Reciprocals in $\mathbf{Z}_5$.

This follows from the multiplication table. As we see, every nonzero modular integer has a reciprocal, and therefore is a unit in $\mathbf{Z}_5$. So the numbers in $\mathbf{Z}_5$ behave just like fractions, the rational numbers $\mathbf{Q}$: one can compute a/b for any a in $\mathbf{Z}_5$ and any nonzero b in $\mathbf{Z}_5$.

Since every nonzero number in $\mathbf{Z}_5$ is a unit, there are no primes in $\mathbf{Z}_5$, and there is no fundamental theorem of arithmetic as in $\mathbf{Z}$. We shall see, however, there is plenty of other stuff going on.

Division a/b is given by multiplication with the reciprocal, $a/b = a(1/b)$. For example, $2/3 = 2(1/3) = 2 \cdot 2 = 4$ in $\mathbf{Z}_5$. Here is the division table for $\mathbf{Z}_5$ (Figure 6.5).

Here a is along the top row, and b is along the left column.

÷	0	1	2	3	4
1	0	1	2	3	4
2	0	3	1	4	2
3	0	2	4	1	3
4	0	4	3	2	1

Fig. 6.5 Division in $\mathbf{Z}_5$.

Note our arithmetic examples here were in $\mathbf{Z}_5$. If the modulus is 6, in $\mathbf{Z}_6$, things are different: $2 \cdot 3 = 6 = 0$, since in $\mathbf{Z}_6$ zero is the sum of 6 ones. Thus each $\mathbf{Z}_n$, $n = 1, 2, 3, \dots$, is not only different from $\mathbf{Z}$, but also different from each other.

6.2 Congruence and Equality

Our first concern is comparing numbers in $\mathbf{Z}$ with numbers in $\mathbf{Z}_n$. For example, $6 \neq 11$ in $\mathbf{Z}$ but $6 = 11$ in $\mathbf{Z}_5$. Remember 6 means the sum of six ones, both in $\mathbf{Z}$ and in $\mathbf{Z}_5$. The only difference is that in $\mathbf{Z}_5$, $5 = 0$, so $6 = 1$. Similarly, $11 = 1$ in $\mathbf{Z}_5$, since $11 = 2 \cdot 5 + 1$. Hence $6 = 11$ in $\mathbf{Z}_5$. What is the general rule?

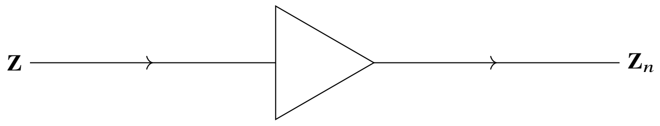

Fig. 6.6 The function $a \mapsto a_n$.

Let a be an integer in $\mathbf{Z}$. Then a is the sum of a ones. We use this to interpret a as a modular integer as follows. Write a as a sum of ones, or as the negative of a sum of ones, then interpret the addition in $\mathbf{Z}_n$. This then yields a modular integer in $\mathbf{Z}_n$, which we call a_n, or a when the context is clear. To be more explicit, we should write one in $\mathbf{Z}$ as 1, and one in $\mathbf{Z}_n$ as 1_n. Then the function (Figure 6.6) sends the integer a to the modular integer $a_n = a \cdot 1_n$, where $a \cdot 1_n$ is $1_n + 1_n + \cdots + 1_n$ summed a times. But we won't do this, we'll continue to write 1 for one in $\mathbf{Z}$ and one in $\mathbf{Z}_5$ and one in $\mathbf{Z}_7$, etc.

Since modular integers can be represented by residues, a_n can be interpreted also as a residue in

$$\{0, 1, 2, \dots, n-1\}.$$

Given integers a and b, we say *a is congruent to b mod n* if n divides $a - b$. We write

$$a \equiv b \mod n.$$

For example 5 divides $11 - 6$, so 11 is congruent to 6 mod 5, $11 \equiv 6 \mod 5$. Similarly $-23 - 47 = -70$ and $-70 = -14 \cdot 5$, so $-23 \equiv 47 \mod 5$. We show that congruence in $\mathbf{Z}$ corresponds to equality in $\mathbf{Z}_n$.

Theorem 6.2.1 *Let a and b be integers. Then $a \equiv b \mod n$ in $\mathbf{Z}$ if and only if $a = b$ in $\mathbf{Z}_n$.*

Proof Let r be the remainder after dividing a by n, and let s be the remainder after dividing b by n. Then $a = b$ in $\mathbf{Z}_n$ if and only if $r = s$, which happens if and only if there is no remainder after dividing $a - b$ by n. But $a - b$ is divisible by n if and only if $a \equiv b \mod n$. □

As a consequence,

Theorem 6.2.2 *Fix $n \geq 1$. Then $\mathbf{Z}_n$ is finite and in fact has n elements.*

Theorem 6.2.3 *For a, b, c, d in $\mathbf{Z}$, $a \equiv b \mod n$ and $c \equiv d \mod n$ implies $a + c \equiv b + d \mod n$ and $ac \equiv bd \mod n$.*

Proof $a \equiv b \mod n$ is the same as $a - b$ is divisible by n, and $c \equiv \mod n$ is the same as $c - d$ is divisible by n. Thus

$$(a - b) + (c - d) = (a + c) - (b + d)$$

is divisible by n, or $a + c \equiv b + d \mod n$. Since

$$ac - bd = (a - b)c + b(c - d),$$

we also have $ac - bd$ is divisible by n, hence $ac \equiv bd \mod n$. □

This shows congruence behaves like equality. For example, if $a \equiv b \mod n$, then

$$a^3 - 5a^2 + 17 \equiv b^3 - 5b^2 + 17 \mod n.$$

Above we called

$$\{0, 1, 2, \ldots, n - 1\}$$

the *residues* mod n. These are the same as `range(0,n)`. We call

$$\{1, 2, \ldots, n - 1\}$$

the *nonzero residues*. These are the same as `range(1,n)`.

The residue of -1 mod n is $n - 1$. This is consistent with backward indexing for lists, where `a[-1]` is the last item in the `list` `a`.

6.3 Classes

Even after the preceding sections, the reader may still have the nagging feeling: But what *are* modular integers? Are they integers or what? How can one have 4 in $\mathbf{Z}$ and 4 in $\mathbf{Z}_5$ and 4 in $\mathbf{Z}_6$ as three different *mathematical* objects?

In fact, we never explicitly said what the sets $\mathbf{Z}$, $\mathbf{Z}_5$, $\mathbf{Z}_6$ were, nor can we. All we can do is learn about $\mathbf{Z}$, $\mathbf{Z}_5$, $\mathbf{Z}_6$ through the axioms characterizing them.

To understand this better, we appeal to Python. Why? Because the situation there is completely analogous. We don't really know what an `int` or an `object` is. All we can do is access an `int` or `object` through its methods.

In Python, the integer 23 corresponds to the `object` `23` whose `type` is `int`. What Python `type` corresponds to a modular integer? Since there is no such built-in[3] `type`, we now build one ourselves.

A user-defined `type` is a `class`. We call the `class` we build `modint`. Recall a `type` is a factory that produces `objects` (§1.1). To specify a modular integer a mod n, we need the modulus n, a positive integer, and the value a, an integer. So the function `modint` has two arguments `a` and `n`, both `int`s, and returns an `object` `modint(a,n)`. Thus

```
>>>  m = modint(3,5)
```

assigns an `object` of type `modint` to the variable `a`. Then `a` points to an `object` of type `modint`,

```
>>>  type(m)
__main__.modint
```

An *instance* of the `class` `t` is an `object` of `type` `t`. So `23` is an instance of `int`, and `modint(3,5)` is an instance of `modint`. The assignment `a = 23` is a creation of an instance, an *instantiation* of `type` `int`, and `m = modint(3,5)` is an instantiation of `type` `modint`.

A `class`, while effectively a function, has different syntax,

```
0>>  class modint:
1>>     def __init__(self,value,modulus):
2>>        self.modulus = modulus
2>>        self.value = value
```

This says `modint` is a class, an object factory, that creates objects with two *attributes* `modulus` and `value`. Given this code, and the assignment above, attributes may be recovered using *dot notation*,

```
>>>  m.modulus
5
>>>  m.value
3
```

[3] There are built-in types for modular integers, but we won't use them, since the point here is to build our own from scratch.

In any class, the function __init__ executes upon instantiation of the class. Functions appearing inside a class are called methods. Here the method __init__ has only one job to do, which is to specify the arguments and how to retrieve them using dot notation. The zeroth argument of __init__ always refers to the instance to-be-created, that's why it makes sense, and is traditional, to call it self.

Any function can be inserted into the code body of a class. Then it is referred to as a method. If the code

```
>>>     def example(self,a,b,c):
>>>        ...
```

is inserted into the class modint, the method example is accessed via

```
>>>   m.example(a,b,c)
```

Functions whose names start and end with double underscores, like __init__, are *special* methods built into Python. For example, to print the modular integer as "a mod n", we use the special method __str__,

```
1>>     def __str__(self):
2>>        v = self.value % modulus
2>>        return str(v) + ' mod ' + str(modulus)
```

When this code is added to the class modint, we have

```
>>>   m = modint(7,5)
>>>   print(m)
2 mod 5
```

Step by step, the value 7 is reduced mod 5 to obtain the residue 2, then the residue and the modulus 5 are turned into strings and concatenated with the string ' mod '. Note for this to work, we have to use the print function.

We know to test for equality of ints by using ==. How do we test for equality of modint objects? To continue the usage of == for modint, add the the following code to the class modint,

```
1>>     def error(self,other):
2>>        if self.modulus != other.modulus:
3>>           raise ValueError('The moduli are different.')
>>>
1>>     def __eq__(self,other):
2>>        error(self,other)
2>>        a = self.value
2>>        b = other.value
2>>        return a == b
```

Then

```
>>>   m = modint(734,52)
>>>   n = modint(333,500)
>>>   m == n
```

```
Error: The moduli are different.
>>>  m = modint(734,500)
>>>  m == n
False
>>>  m = modint(833,500)
>>>  m == n
True
>>>  m != n
False
```

To summarize, we've built a `type` `modint` whose objects are modular integers. More exactly, the function that sends the integer a in $\mathbf{Z}$ to the modular integer a_n in $\mathbf{Z}_n$, discussed in the previous section, is precisely the same as the function that inputs the `int` a and returns `modint(a,n)`.

To complete the picture, we describe how to add and multiply `modint`s. Just add the following special methods to `modint`,

```
1>>     def __add__(self,other):
2>>        error(self,other)
2>>        mod = self.modulus
2>>        sum = self.value + other.value
>>>        return modint(sum % mod,mod)
>>>
1>>     def __mul__(self,other):
2>>        error(self,other)
2>>        mod = self.modulus
2>>        prod = self.value * other.value
2>>        return modint(prod % mod,mod)
```

Then

```
>>>  m = modint(734,500)
>>>  n = modint(323,500)
>>>  print(m + n)
57 mod 500
>>>  print(m*n)
82 mod 500
```

As mentioned in §1.3, the ability to customize operators such as + and * to work with non-`int` objects is an example of *operator overloading*.

To compute a power a^e mod n, where e may be a large integer, the simplest method is to multiply a by itself e times, each time reducing the result mod n. This takes $2n$ multiplications/divisions. A much faster method is to write e in binary and take successive squarings of a mod n. For example, with $e = 19 = 2^4 + 2 + 1$, $e//2 = 9 = 2^3 + 1$ and $e\%2 = 1$, hence

$$a^e = a^{19} = a^{2^4+2+1} = a^{2^4+2} \cdot a = (a^2)^{2^3+1} \cdot a = (a^2)^{e//2} \cdot a^{e\%2}.$$

Similarly, with $e = 18 = 2^4 + 2$, $e//2 = 9 = 2^3 + 1$ and $e\%2 = 0$, hence also

$$a^e = a^{18} = a^{2^4+2} = (a^2)^{2^3+1} \cdot a = (a^2)^{e//2} a^{e\%2}.$$

This procedure leads to a special method `__pow__(self,e)` yielding (Exercise 6.8)

```
>>>  m = modint(7,11)
>>>  m**23
2
```

This agrees with

$$7^{23} = 27368747340080916343 \equiv 2 \mod 11.$$

This takes roughly $\log(e)$ multiplications/divisions, because each time we divide e by 2, we get one multiplication and division. To reduce the number of multiplications even further, the exponent e is often taken in RSA cryptosystems (§6.8) to equal $2^{2^k} + 1$ for some k (Exercise 5.2).

`reciprocal(self)` and `legendre(self)` are other methods added to the class `modint` in the exercises.

6.4 Negatives and Reciprocals

Going back to negatives, the axioms guarantee that every a in $\mathbf{Z}_n$ has a negative $-a$. Is there a in $\mathbf{Z}_n$ satisfying $a = -a$? $a = 0$ works, but is there any other a?

Theorem 6.4.1 *If n is odd, $a = -a$ in $\mathbf{Z}_n$ only if $a = 0$ in $\mathbf{Z}_n$. If n is even, $a = -a$ in $\mathbf{Z}_n$ when $a = n/2$ or $a = 0$ in $\mathbf{Z}_n$.*

Proof Given $0 \le a < n$, let $b = n - a$. Then $a + b = n$, so $b = -a$ in $\mathbf{Z}_n$. Hence $a = -a$ happens when $a = n - a$, which in turn happens when $2a = n$. If n is even, this happens for $a = 0$ or $a = n/2$. If n is odd, this happens for $a = 0$. □

Let us go back to reciprocals, but this time with modulus 6, in $\mathbf{Z}_6 = \{0, 1, 2, 3, 4, 5\}$. What is the reciprocal of 3 in $\mathbf{Z}_6$? Checking $0 * 3 = 0$, $1 * 3 = 3$, $2 * 3 = 0$, $3 * 3 = 3$, $4 * 3 = 0$, $5 * 3 = 3$, none equal 1 in $\mathbf{Z}_6$, so there is no reciprocal of 3 in $\mathbf{Z}_6$ (remember this is mod 6 now). In fact the only modular integer in $\mathbf{Z}_6$ that is a unit[4] is 5, since $5 \cdot 5 = 1$ in $\mathbf{Z}_6$.

Given n and a, when does a have a reciprocal in $\mathbf{Z}_n$? Of course a can't be zero, because (as we saw in Chapter 4), anything times zero is zero.

Theorem 6.4.2 *A nonzero a has a reciprocal in $\mathbf{Z}_n$ if and only if* $\gcd(a, n) = 1$.

Proof If $\gcd(a, n) = 1$, by Theorem 5.4.2, there are s and t satisfying

[4] i.e. has a reciprocal

$$sa + tn = 1.$$

Then $sa - 1$ is divisible by n, so $sa \equiv 1 \mod n$, so $sa = 1$ in $\mathbf{Z}_n$. Conversely, if a has a reciprocal s in $\mathbf{Z}_n$, then $as = 1$ in $\mathbf{Z}_n$, which implies $as \equiv 1 \mod n$, which implies $as - 1$ is divisible by n, or there is a t with $as - 1 = -tn$. If $g = \gcd(a, n)$, then g is a common divisor of a and n, so g divides $as + tn$, which is 1. Thus $g = 1$. □

For example, $\gcd(2, 6) = 2 = \gcd(4, 6)$ and $\gcd(3, 6) = 3$, so 2, 3, and 4 are not units in $\mathbf{Z}_6$. But $\gcd(1, 6) = 1$ and $\gcd(5, 6) = 1$, so 1 and $5 = -1$ are units in $\mathbf{Z}_6$.

How does one compute $1/a$ given a and n, when $\gcd(a, n) = 1$ The simplest approach is to try all nonzero residues until one fits.

```
def reciprocal(a,n):
  for x in range(1,n):
    if (a*x)%n == 1:
      return x
  return 'None'
```

This takes around n multiplications/divisions. A faster approach is to use the extended Euclidean algorithm (5.4.2) to compute s and t satisfying

$$sa + tn = 1.$$

This takes around $\log(n)$ multiplications/divisions. Then $sa \equiv 1 \bmod n$, or s is the reciprocal of a mod n.

An important special case is

Theorem 6.4.3 *Let p be a prime. Then every nonzero modular integer in $\mathbf{Z}_p$ has a reciprocal. Conversely, if every nonzero modular integer in $\mathbf{Z}_n$ has a reciprocal, then n is prime.*

Proof If $a \neq 0$ in $\mathbf{Z}_p$, then p does not divide a in $\mathbf{Z}$, so $\gcd(a, p) = 1$, so a has a reciprocal in $\mathbf{Z}_p$. Now suppose every nonzero modular integer in $\mathbf{Z}_n$ has a reciprocal. If n is composite, $n = ab$, with $a \neq 0$, then $ab = 0$ in $\mathbf{Z}_n$, so $b = (1/a)ab = 0$, implying $n = ab = a0 = 0$, a contradiction to $n \geq 1$. Hence n is prime. □

As a consequence, we can cancel in $\mathbf{Z}_p$,

Theorem 6.4.4 *Let p be a prime. If $a \neq 0$ and $ab = ac$ in $\mathbf{Z}_p$, then $b = c$ in $\mathbf{Z}_p$.*

Proof We know $ab = ac$. let $1/a$ be the reciprocal of a. Then

$$b = 1b = ((1/a)a)b = (1/a)(ab) = (1/a)(ac) = ((1/a)a)c = 1c = c.$$

When p is not prime, this is false, for example $2 \cdot 3 = 0$ in $\mathbf{Z}_6$.

When $p = 2$, $a = 1$ is the only solution of $a = 1/a$. When $p > 2$, there are two solutions.

Theorem 6.4.5 *Let p be an odd prime. Then $a = 1/a$ in $\mathbf{Z}_n$ if and only if $a = \pm 1$ in $\mathbf{Z}_p$. For all other a in $\mathbf{Z}_p$, $a \neq 1/a$.*

Proof $x = 1/x$ in $\mathbf{Z}_p$ if and only if $x^2 = 1$ in $\mathbf{Z}_p$. Clearly $x = 1$ satisfies this equation. If $x \neq 1$ in $\mathbf{Z}_p$ satisfies this equation, then (difference of two squares)

$$(x-1)(x+1) = x^2 - 1 = 0.$$

But $x - 1 \neq 0$ and the modulus is prime, so $x - 1$ has a reciprocal. Multiplying this last equation by the reciprocal of $x - 1$ allows us to cancel $x - 1$, resulting in $x + 1 = 0$, or $x = -1$ in $\mathbf{Z}_p$. $\square$

Cancellation in $\mathbf{Z}_p$ can be used for

Theorem 6.4.6 *Let p be prime and let $f(x)$ be a polynomial of degree $n \geq 1$. Then*

$$f(x) = 0 \mod p$$

has at most n solutions.

Proof By induction on the the degree n. If $n = 1$, $f(x) = ax + b$ with $a \neq 0$, so the equation is $ax + b = 0$. Since this has the unique solution $x = -b/a$ ($1/a$ exists since p is prime), the base step is proved. Now assume the result is true for any polynomial of degree n, and let $f(x)$ be a polynomial of degree $n+1$. If a is a solution of $f(x) = 0 \mod p$, then $f(a) = 0 \mod p$. By Exercise 2.7, there is a polynomial $g(x)$ satisfying

$$f(x) = f(x) - f(a) = (x - a)g(x).$$

Since $f(x)$ has degree $n + 1$, $g(x)$ has degree n. If b is another solution of $f(x) = 0 \mod p$, then $f(b) = 0 \mod p$, so $0 = f(b) = (b - a)g(b) \mod p$. By cancelling $b - a$, b is a solution of $g(x) = 0 \mod p$. Since $g(x) = 0 \mod p$ has at most n solutions, it follows that $f(x) = 0 \mod p$ has at most $n+1$ solutions. This establishes the inductive step, and we are done. $\square$

Since $x = -x$ is the same as $2x = 0$, Theorem 6.4.1 deals with the equation $2x = 0 \mod n$. When n is even, this has two solutions. When n is odd, this has one solution.

Since $x = 1/x$ is the same as $x^2 = 1$, Theorem 6.4.5 deals with the equation $x^2 = 1 \mod n$. Here there are two solutions for n prime. If n is not prime, this may not be true. For example, $x \pm 1$ and $x = \pm 4$ are solutions of $x^2 = 1 \mod 15$.

Given a and b in $\mathbf{Z}_n$, $x = a/b$ is the solution of $bx = a$ mod n. We saw above reciprocals for general n may not exist, so in general a/b may not exist in $\mathbf{Z}_n$. In Exercise 6.12, you are asked to show a/b exists in $\mathbf{Z}_n$ if and only if $\gcd(b, n)$ divides a.

6.5 Wilson's Theorem

Let p be an odd prime. The nonzero residues mod p are the integers

$$1, 2, \ldots, p-1.$$

Each nonzero residue a, other than 1 and $p-1$, has a distinct reciprocal $a^* = 1/a$ (Theorem 6.4.5). Thus if we pair off each nonzero residue a with its reciprocal a^*, we obtain $(p-1)/2$ pairs (a, a^*), each satisfying $aa^* = 1 \mod p$. Multiplying $1 \cdot 2 \cdot \cdots \cdot (p-1) = (p-1)!$ in pairs, all the factors cancel, except for the first, which is 1, and the last, which is $p-1$. Hence

$$(p-1)! \equiv p-1 \equiv -1 \qquad \mod p.$$

We have shown

Theorem 6.5.1 (Wilson's Theorem) *Let p be an odd prime. Then*

$$(p-1)! \equiv -1 \mod p.$$

Let's go back to the binomial coefficients, given by (2.4.3) and Pascal's triangle (Figure 2.1). Starting with $n = 2$, take the n-th row and reduce it mod n. So `[1,2,1]` reduced mod 2 becomes `[1,0,1]`, `[1,3,3,1]` reduced mod 3 becomes `[1,0,0,1]`, `[1,4,6,4,1]` reduced mod 4 becomes `[1,0,2,0,1]`, and Pascal's triangle reduces to Figure 6.7.

You'll notice that the rows corresponding to n prime are zeroed out after reducing mod n, except for either end. But the residue of an integer is 0 mod n if and only if a is divisible by n, so this follows from

Theorem 6.5.2 *For p prime, $\binom{p}{k}$ is divisible by p for $1 \le k \le p-1$.*

Proof By (2.4.3), for each $1 \le k \le p-1$, p divides $\binom{p}{k} \cdot k!$. But p does not divide $k!$, as long as $1 \le k \le p-1$. By Theorem 5.4.3, p divides $\binom{p}{k}$. □

6.6 Fermat's Little Theorem

Continuing as in the previous section, let p be an odd prime.

Let a be any nonzero residue. Then by Theorem 6.4.4, $ak = am$ in $\mathbf{Z}_p$ implies $k = m$. Thus the numbers

$$a \cdot 1, a \cdot 2, \ldots, a \cdot (p-1)$$

are $p-1$ distinct numbers mod p, so they are the nonzero residues re-arranged in a different order. Hence, if we multiply them, we get

$$(a \cdot 1) \cdot (a \cdot 2) \cdot \cdots \cdot (a \cdot (p-1)) \equiv (p-1)! \mod p.$$

But we can pull out all the a's from the left side, and get $p-1$ of them, so

$$a^{p-1}(p-1)! \equiv (p-1)! \mod p.$$

```
 n = 0:                      1
 n = 1:                    1   1
 n = 2:                  1   0   1
 n = 3:                1   0   0   1
 n = 4:              1   0   2   0   1
 n = 5:            1   0   0   0   0   1
 n = 6:          1   0   3   2   3   0   1
 n = 7:        1   0   0   0   0   0   0   1
 n = 8:      1   0   4   0   6   0   4   0   1
 n = 9:    1   0   0   3   0   0   3   0   0   1
n = 10:  1   0   5   0   0   2   0   0   5   0   1
```

Fig. 6.7 The reduced Pascal's triangle.

Cancelling $(p-1)!$, we conclude

Theorem 6.6.1 (Fermat's Little Theorem) *Let p be an odd prime. If a is any integer not divisible by p, then*
$$a^{p-1} \equiv 1 \mod p.$$

What if the modulus n is not prime? The same argument works, but we multiply only over integers $0 \le b < n$ that are relatively prime to n. Since there are $\phi(n)$ such integers, we obtain

Theorem 6.6.2 (Euler's Theorem) *Let $n \geq 2$. If a is any integer relatively prime to n, then*
$$a^{\phi(n)} \equiv 1 \mod n.$$

6.7 Existence of $\sqrt{-1}$

Let p be an odd prime. In §6.4, we solved the equations $2x = 0 \mod p$ and $x^2 = 1 \mod p$. What other equations

$$f(x) = 0 \mod p$$

can we solve? By Theorem 6.4.6, we know this equation has at most n solutions, where n is the degree of the polynomial $f(x)$.

In this section, we solve

$$x^2 + 1 = 0 \mod p. \tag{6.7.1}$$

For example, for $p = 5$, $x = 2$ is a solution of (6.7.1), since $2^2 = 4 = -1 \mod 5$. Another solution is $x = 3 = -2$, Since $(-2)^2 = 4 = -1 \mod 5$. Thus for $p = 5$, (6.7.1) has two solutions, ± 2.

For $p = 7$, the squares x^2 of the nonzero residues x are $1^1 = 1$, $2^2 = 4$, $3^2 = 2$, $4^2 = 2$, $5^2 = 4$, $6^2 = 1$. Since none of these equals $-1 \mod 7$, (6.7.1) has no solutions when $p = 7$.

So, depending on p, (6.7.1) sometimes has solutions and sometimes does not. A solution x of (6.7.1) is $\sqrt{-1}$, *a square root of* -1 *mod* p, since x^2 equals -1 mod p.

Since $(-1)^2 = 1$, when (6.7.1) has a solution, it in fact has two solutions, since $a \neq -a \mod p$ when p is odd, except for $a = \pm 1$ (Theorem 6.4.1).

By checking all residues, the function `yes_or_no(p)` easily tells whether or not (6.7.1) has solutions mod p,

```
0>>  def yes_or_no(p):
1>>     for x in range(1,p):
2>>        if (x**2+1) % p == 0:
3>>           return x
1>>     return 'no'
```

yielding Figure 6.8.

p	x	p	x	p	x	p	x
3	no	19	no	43	no	71	no
5	2	23	no	47	no	73	27
7	no	29	12	53	23	79	no
11	no	31	no	59	no	83	no
13	5	37	6	61	11	89	34
17	4	41	9	67	no	97	22

Fig. 6.8 Existence of $\sqrt{-1}$ mod p.

Historically, attempting to solve $x^2 = -1$ in standard arithmetic led to the discovery of complex numbers. Because of this, figuring out the pattern in Figure 6.8, which amounts to solving (6.7.1) for all p prime, is important.

To figure out the pattern, we repeat the idea in the proof of Wilson's Theorem. There, we paired each nonzero residue a with its reciprocal $a^* = 1/a$. Then a equalled its partner a^* if and only if a was a solution of the equation $x = x^* = 1/x \mod p$, or $x^2 = 1 \mod p$.

Here, in the present proof, we pair each nonzero residue a with its negative reciprocal $a^* = -1/a$. Then a equals its partner a^* if and only if a is a solution of the equation $x = x^* = -1/x \mod p$, or $x^2 = -1 \mod p$. There are two cases.

The first case is when there are no solutions of (6.7.1) mod p. Then none of the $p-1$ nonzero residues a are equal to their partner a^*, so when we multiply

$$1 \cdot 2 \cdot \cdots \cdot (p-1) = (p-1)!, \tag{6.7.2}$$

we obtain $(p-1)/2$ pairs aa^*, each pair reducing to -1. By Wilson's theorem, this yields

$$-1 \equiv (p-1)! \equiv (-1)^{(p-1)/2} \mod p. \tag{6.7.3}$$

The second case is when there are two solutions of (6.7.1) mod p. In this case, two of the $p-1$ nonzero residues a are equal to their partner a^*, and the remaining $p-3$ are not. Call the two solutions z and $-z$, so $z^2 = -1 \mod p$. Then $z(-z) = -z^2 = 1 \mod p$. Hence when we multiply (6.7.2), we obtain $(p-3)/2$ pairs aa^*, each pair reducing to -1, together with $\pm z$. By Wilson's theorem, this yields

$$-1 \equiv (p-1)! \equiv (-1)^{(p-3)/2} z(-z) = (-1)^{(p-3)/2} \mod p. \tag{6.7.4}$$

Since p is odd, (6.7.3) implies $(-1)^{(p-1)/2} = -1$, and (6.7.4) implies $(-1)^{(p-3)/2} = -1$. But this happens if the powers of -1 are odd. Therefore if there are no solutions, $(p-1)/2$ is odd. If there are solutions, $(p-3)/2$ is odd.

Exercise 6.14 shows $(p-1)/2$ is odd if and only if $p \equiv 3 \mod 4$, and $(p-3)/2$ is odd if and only if $p \equiv 1 \mod 4$. We have proved

Theorem 6.7.1 (Euler's Theorem) *Let p be an odd prime number. Then (6.7.1) does or doesn't have solutions according to whether $p = 1 \mod 4$ or $p = 3 \mod 4$.*

6.8 RSA Encryption

Suppose you want to send somebody a message by publishing it in a newspaper, but you want only the recipient to understand the contents of the message. Nobody else should be able to read the message. It is possible, but highly improbable, that nobody will notice or read the published message.

A more reliable alternative is to *garble* or *hash* or *encode* the published message via some systematic procedure, where letters in the original message are replaced by other letters in a systematic fashion. This encoding process is called *encryption*.

The encryption procedure must be *reversible,* so that the recipient can recover the original message. The decoding process the recipient employs to extract the message is *decryption*. Together, these procedures are a *cryptosystem*. Since nobody else should be able to do this, the cryptosystem utilized must depend a secret key or password, without which it will not work. Because of the necessity of a simultaneous sharing of a secret key between sender and recipient, this is *symmetric key cryptography*.

The chief drawback of the above cryptosystem is the sharing of the key. How does the sender send the key to the recipient, or vice-versa? Publish it in a newspaper? Talk on the phone? It is possible, but highly improbable, that these methods are secure. Thus one needs a reliable alternative.

In the 1970's, three mathematicians with last name initials R, S, and A, came up with an *asymmetric* or a *public* key cryptosystem, leading to *public-key cryptography*.

In a public-key cryptosystem, there are two distinct keys, a *public key* used for encryption, and a *private key* used for decryption. Each person desiring to send or receive messages will have one of each. As the names imply, a person's private key is kept private, while a person's public key is shared with the world.

Going back to our scenario, with you wanting to send a message to a recipient, here's how it works. The *recipient* sends you their public key, which you use to encrypt the message. Then you send the encrypted message. Once received, the recipient decrypts it with their private key. That's it.

In RSA cryptography, the cryptosystem is based in modular arithmetic. Modular arithmetic provides us with a cryptosystem that uses two keys, as described above. Since then, other public-key cryptosystems have been devised, based on other mathematical techniques.

In RSA cryptography, the public and private keys are based on a pair (p, q) of two large primes p and q. The public key is the pair (n, e), where $n = pq$ is the product of p and q, and e is any integer less than and relatively prime to $(p-1)(q-1)$. The private key is the pair (n, d), where d is the reciprocal of e mod $(p-1)(q-1)$. We also assume the message to be sent is an integer m less than n. This last condition is not a constraint, since any electronic message is an integer.

For example, with $(p, q) = (5, 11)$, a corresponding public key is $(n, e) = (55, 3)$, since 3 is relatively prime to $(p-1)(q-1) = 40$. The corresponding private key is $(n, d) = (55, 27)$, since $3 \cdot 27 \equiv 1 \mod 40$. With these choices, you can encrypt any integer m up to size $pq - 1 = 54$.

These days the recipient is a web site that you're interacting with. Here's the private data from a website:

```
>>>  p = '''00:f1:63:a4:49:8c:bd:82:de:73:ca:fb:54:e1:7b:
       41:4f:14:6e:69:94:f3:c7:72:c7:69:ba:4a:ae:25:
       50:df:ce:c4:61:10:26:17:db:a4:fe:1c:4c:92:6c:
       c4:fb:16:d3:57:1e:4b:28:9f:b5:6d:2c:00:ec:2f:
       23:1f:a2:67:c4:d1:13:ad:b1:47:dc:79:51:b8:fe:
       39:41:11:bb:36:13:9d:61:58:e6:bd:02:1d:4b:ce:
       57:f5:32:7d:b6:9f:23:67:ff:2d:5e:51:dd:a8:50:
       44:a8:59:0b:9f:4d:e5:0c:15:bd:63:3e:77:2f:b2:
       c1:17:c1:f1:19:a0:e9:19:a5'''
>>>  q = '''00:df:12:aa:b2:4c:a2:b2:63:a3:d9:f6:1e:0d:a6:
       07:ee:fc:7a:4d:13:bc:29:0c:f9:b5:11:15:d8:ec:
       a8:3f:c5:d3:29:6d:75:63:6f:db:d1:c7:44:94:ab:
       cf:98:9a:72:89:cd:bd:1a:23:86:ab:ec:e1:e7:df:
       0e:bc:ee:5c:73:92:2b:32:bc:2b:bc:de:28:67:41:
       92:46:5e:e0:72:21:2d:31:9e:1c:2c:a5:9b:c6:56:
          88:5a:84:6f:22:f4:a5:60:64:5b:66:ee:88:db:54:
          39:55:ce:07:df:81:b0:f5:ba:b6:54:87:95:3b:e9:
          23:73:54:1e:34:a5:84:39:4b'''
>>>   e = 65537
```

and d is given by Exercise 6.17. Here p and q are in bytes, and e is in decimal. The exponent e is a prime of a particular form, a *Fermat prime* (Exercise 6.16), so e is automatically relatively prime to $(p-1)(q-1)$.

Encryption of the *cleartext* message m is carried out by raising m to the power $e \bmod n$, $c = m^e \mod n$. This results in the *cyphertext* message c. The RSA result says if you choose d to be the reciprocal of $e \bmod (p-1)(q-1)$, then $m \equiv c^d \mod n$. Thus decryption of the cyphertext message c is carried out by raising c to the power $d \bmod n$, $m \equiv c^d \mod n$.

To summarize, the map $m \mapsto m^e \mod n$ is encryption, and the map $c \mapsto c^d \bmod n$ is decryption. The RSA result is that these maps are inverses to each other, and thus we have a valid cryptosystem.

Since e is chosen relatively prime to $(p-1)(q-1)$, by Theorem 6.4.2, e does have a reciprocal d,

$$ed = 1 \mod (p-1)(q-1).$$

Here is the formal statement.

Theorem 6.8.1 (RSA algorithm) *Let p and q be primes and let $n = pq$. If $m < n$ and c is the residue of m^e mod n, then the residue of c^d mod n is m,*

$$c^d = m^{ed} \equiv m \mod n. \tag{6.8.1}$$

Proof Since $ed \equiv 1 \mod (p-1)(q-1)$, $ed-1$ is a multiple of $(p-1)(q-1)$, so $ed-1 = k(p-1)(q-1)$ for some integer k. If m is relatively prime to $n = pq$, then by Euler's theorem

$$m^{ed-1} = m^{k(p-1)(q-1)} = m^{k\phi(n)} \equiv 1^k = 1 \mod n.$$

Multiplying by m yields (6.8.1). If m is divisible by n, then both sides of (6.8.1) are zero, so (6.8.1) is valid. If m is divisible by p but not by q, then by Fermat's little theorem,

$$m^{ed-1} = m^{k(p-1)(q-1)} \equiv 1^{k(p-1)} = 1 \mod q.$$

Thus $m^{ed-1}-1$ is divisible by q. Since m is divisible by p, multiplying by m implies $m^{ed}-m$ is divisible by $n = pq$, which is (6.8.1). Similarly if m is divisible by q but not by p. □

If one knows the public key, one knows pq. Factoring pq will then lead to (p, q), and from there to d. How is such a system secure? This is secure only if p and q are chosen large enough, at least 200+ decimal digits long. Then, with current technology and current mathematics, the fastest factoring algorithms applied to pq take hundreds of years. As time goes on, however, both mathematics and technology get better, forcing one to use larger primes (p, q), or alternative cryptosystems. This has already happened over the last twenty years.

6.9 Quadratic Residues

Let p be an odd prime. After $ax = b$ and $x^2 = \pm 1$, the next equation to try is

$$x^2 \equiv a, \qquad \bmod p. \tag{6.9.1}$$

Here a is any integer. If x is a solution, then

$$(p-x)^2 = p^2 - 2px + x^2 \equiv a \mod p,$$

so $p - x$ is a solution. Since p is odd, $p - x \neq x$. Thus (6.9.1) either has no solutions, or has two solutions x and $p - x$.

Since the case $a = 0$ is trivial (0 is a quadratic residue), we deal only with nonzero residues $a \not\equiv 0 \mod p$. We call a a *quadratic residue* if (6.9.1) has two solutions. Otherwise, a is a *quadratic nonresidue*.

For example, -16 is a quadratic residue mod 5 since for $x = 2$, $x^2 = 4 \equiv -16 \mod 5$, and 17 is a quadratic nonresidue since $17 \not\equiv 0^2$, $17 \not\equiv 1^2$, $17 \not\equiv 2^2$, $17 \not\equiv 3^2$, $17 \not\equiv 4^2 \mod 5$.

For any p, 1 is a quadratic residue. By Euler's theorem, -1 is a quadratic residue if and only if $p \equiv 1 \mod 4$.

It is natural to denote the solutions of (6.9.1) as $\pm\sqrt{a} \mod p$. In §6.7, we studied $\sqrt{-1} \mod p$. Here we consider other a, for example $a = 5$, leading to Figure 6.9.

p	x	p	x	p	x	p	x
3	no	19	9	43	no	71	17
5	0	23	no	47	no	73	no
7	no	29	11	53	no	79	20
11	4	31	6	59	8	83	no
13	no	37	no	61	26	89	19
17	no	41	13	67	no	97	no

Fig. 6.9 Existence of $\sqrt{5} \bmod p$.

If a and b are quadratic residues, then there are x and y satisfying $x^2 \equiv a \mod p$ and $y^2 \equiv b \mod p$ respectively. Since $(xy)^2 = x^2y^2$ and $(x/y)^2 = x^2/y^2$, it follows that ab and a/b are quadratic residues. From this, it follows (Exercise 6.18) that there are an equal number of quadratic residues and nonresidues,

Theorem 6.9.1 *Let p be an odd prime. There are $(p-1)/2$ quadratic residues and $(p-1)/2$ quadratic nonresidues.*

For example, for $p = 5$, we have $1^2 = 1$, $2^2 = 4$, $3^2 = 4$, $4^2 = 1$, so the quadratic residues mod 5 are 1 and 4. Similarly, for $p = 7$, the quadratic residues are 1, 2, and 4.

If a is a quadratic residue and b is a quadratic nonresidue, it follows from above that ab is a quadratic nonresidue. We encapsulate this behavior by introducing the *Legendre symbol* $(a|p)$ or

$$\left(\frac{a}{p}\right),$$

by defining $(a|p) = +1$ if a is a quadratic residue, and $(a|p) = -1$ if a is a quadratic nonresidue. In this language, $(1|p) = 1$ for all p, and

$$\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$$

by Euler's theorem. We extend $(a|p)$ to any integer a by setting $(a|p) = (a_p|p)$, where a_p is the residue of a mod p.

The above discussion is summarized by

Theorem 6.9.2 *Let p be an odd prime. Then for integers a, b,*

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right).$$

Let a be a nonzero residue. If x is any nonzero residue, let $x^* = a/x$, so $xx^* = a$. Then, as before, we pair the nonzero residues

$$1, 2, \ldots, p-1$$

into pairs (x, x^*). There are two cases.

If a is a quadratic nonresidue mod p, then none of the nonzero residues x equals their partner x^*, so the product (6.7.2) pairs off into $a^{(p-1)/2}$, hence $(p-1)! = a^{(p-1)/2}$.

If a is a quadratic residue mod p, let z and $p - z$ be the solutions of $x^2 = a \mod p$. Then $z = z^* \mod p$, $(p-z) = (p-z)^* \mod p$, and none of the remaining $(p-3)/2$ nonzero residues x equals their partner x^*, so the product (6.7.2) pairs off into $a^{(p-3)/2}z(p-z)$, hence $(p-1)! = a^{(p-3)/2}(-a) = -a^{(p-1)/2}$. Appealing to Wilson's theorem and the Legendre symbol, we conclude

Theorem 6.9.3 *Let p be an odd prime, and suppose p does not divide a. Then*

$$a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) \mod p.$$

If a is an integer, let a_p be the residue of a mod p, and let $|a|_p$ be the residue mod p of $(-1)^{a_p}a_p$. Since p is odd, $|a|_p$ is always even. For example, with $p = 17$,

$|23|_p = |6|_p = 6$ and $|24|_p = |7|_p = -7 = 10$. Here is another formula for the Legendre symbol.

Theorem 6.9.4 (Eisenstein's Lemma) *Let p be an odd prime, and suppose p does not divide a. Let*

$$S(a,p) = (a2)_p + (a4)_p + \cdots + (a(p-1))_p.$$

Then

$$\left(\frac{a}{p}\right) = (-1)^{S(a,p)}. \tag{6.9.2}$$

Proof Look at the even residues $2, 4, \ldots, p-1$; there are $(p-1)/2$ of them. We claim the list

$$|2a|_p, |4a|_p, \ldots, |(p-1)a|_p \tag{6.9.3}$$

consists of $(p-1)/2$ distinct even residues: If $|xa|_p = |ya|_p$ with x and y even, then $xa \equiv \pm ya \mod p$. Cancelling a, $x \equiv \pm y \mod p$. But x and y are both even and p is odd, so we can't have $x = p - y$. Hence $|xa|_p = |ya|_p$ implies $x = y$, and we conclude the residues in (6.9.3) are distinct. Thus (6.9.3) is a rearrangement of the even residues. Multiplying, we have

$$2 \cdot 4 \cdot \cdots \cdot (p-1) = |2a|_p \cdot |4a|_p \cdot \cdots \cdot |(p-1)a|_p.$$

But

$$|2a|_p \cdot |4a|_p \cdot \cdots \cdot |(p-1)a|_p = 2a \cdot 4a \cdot \cdots \cdot (p-1)a \cdot (-1)^{S(a,p)},$$

so

$$2 \cdot 4 \cdot \cdots \cdot (p-1) \equiv (-1)^{S(a,p)} 2a \cdot 4a \cdot \cdots \cdot (p-1)a \mod p.$$

Cancelling $2 \cdot 4 \cdot \cdots \cdot (p-1)$ yields

$$1 \equiv (-1)^{S(a,p)} a^{(p-1)/2} \mod p.$$

By Theorem 6.9.3, this implies

$$\left(\frac{a}{p}\right) \equiv (-1)^{S(a,p)} \mod p.$$

Since p is odd, $(-1)^a \equiv (-1)^b \mod p$ if and only if $(-1)^a = (-1)^b$. This establishes (6.9.2). □

Since any a has a prime factorization, to compute $(a|p)$, by Theorem 6.9.2, it is enough to compute $(q|p)$ for the prime factors q of a. The main result in this section is

Theorem 6.9.5 (Gauss' Law of Quadratic Reciprocity) *Let p and q be odd primes. Then*

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)/4}.$$

Proof The proof is based on an alternate expression for $(q|p)$ involving Figure 6.10. Here, the dots and the card suits ♠, ♡, ♢, ♣ are centered at the points with integer coordinates, the *lattice points*, and the equation of the line is $yp = qx$. Because p and q are relatively prime and the slope of the line is q/p, none of the lattice points are on the line. We count card suits lying below the line. For example, with $p = 17$, $q = 11$, and $x = 14$, since $qx/p = 11 \cdot 14/17 = 9.05$, there are 9 ♠ above the point x.

We say a and b have the *same parity* if $(-1)^a = (-1)^b$. This happens if and only if a and b are both even or both odd. Then a and $-a$ have the same parity. Moreover, since p is odd, ap and a have the same parity. We show first

$$\left(\frac{q}{p}\right) = (-1)^{\text{total number of ♣ and ♠}}. \tag{6.9.4}$$

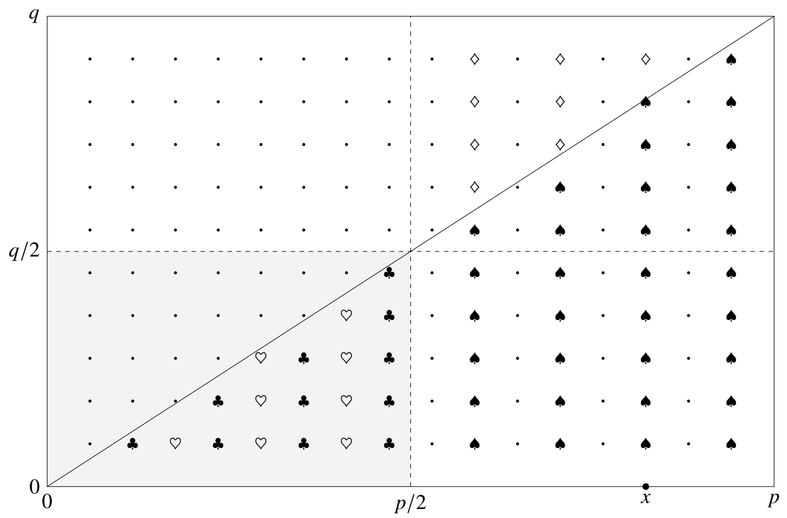

Fig. 6.10 Quadratic reciprocity with $p = 17$ and $q = 11$.

Let x be an even residue. Dividing qx by p, the remainder is $(qx)_p$, and the quotient Q is the number of integers between 0 and qx/p. Thus Q is the number of ♠ or ♣ above the point x. Since p is odd, Q has the same parity as Qp. Since x is even, p is odd, and $qx = Qp + (qx)_p$, $(qx)_p$ has the same parity as the number of ♠ or ♣ above the point x.

It follows that $S(q, p) = (q2)_p + (q4)_p + \cdots + (q(p-1))_p$ and the total number of ♣ and ♠ have the same parity. By Eisenstein's Lemma 6.9.2, this implies (6.9.4). Note (6.9.4) is valid for any positive integer q.

To complete the proof, note each column has $q-1$ dots, which is even. Hence the total number of ♠ has the same parity as the total number of ♢. By symmetry across the point $(p/2, q/2)$, the total number of ♢ equals the total number of ♡. Hence

$$\left(\frac{q}{p}\right) = (-1)^{\text{total number of ♣ and ♡}},$$

or

$$\left(\frac{q}{p}\right) = (-1)^{\text{total number of lattice points in shaded rectangle below } py = qx}.$$

Switching the roles of p and q,

$$\left(\frac{p}{q}\right) = (-1)^{\text{total number of lattice points in shaded rectangle above } py = qx}.$$

Since the total number of lattice points in the shaded rectangle is

$$\frac{p-1}{2} \cdot \frac{q-1}{2},$$

this completes the proof. □

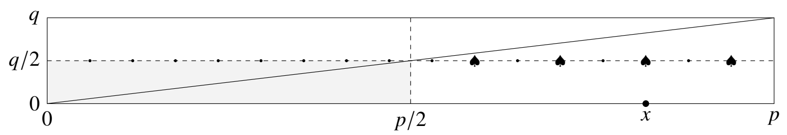

Fig. 6.11 Quadratic reciprocity with $p = 17$ and $q = 2$.

Turning to the special case $q = 2$, (6.9.4) remains valid. By Figure 6.11, above each even residue x, we have no ♣, if $x < p/2$, or one ♠, if $x > p/2$. Hence

$$\text{total number of ♣ and ♠} = |\{k : p/2 < 2k < p\}|.$$

There are four cases: primes p of the form $p = 8n \pm 1$ and $p = 8n \pm 3$. By Exercise 6.20, $|\{k : p/2 < 2k < p\}|$ is even for $p = 8n \pm 1$ and odd for $p = 8n \pm 3$. By (6.9.4), we conclude

Theorem 6.9.6 2 *is a quadratic residue mod p if and only if* $p = 8n \pm 1$.

Turning to the existence of $\sqrt{3}$ mod p, by the law we have

$$\left(\frac{3}{p}\right) = \left(\frac{p}{3}\right) \cdot (-1)^{(p-1)/2}.$$

Trying $p = 3n \pm 1$ doesn't lead anywhere, nor does $p = 6n \pm 1$. Trying $p = 12n \pm 1$ and $p = 12n \pm 5$ leads to Exercise 6.21.

p	x	p	x	p	x	p	x
3	no	19	no	43	no	71	12
5	no	23	5	47	7	73	32
7	3	29	no	53	no	79	9
11	no	31	8	59	no	83	no
13	no	37	no	61	no	89	25
17	6	41	17	67	no	97	14

Fig. 6.12 Existence of $\sqrt{2}$ mod p.

Given p, let $p^* = (-1)^{(p-1)/2}p$. Then quadratic reciprocity may be rephrased as

For any odd primes p and q,
$\sqrt{p}$ mod q *exists if and only if* $\sqrt{q^*}$ mod p *exists.*

Exercises

Exercise 6.1 Build the addition, multiplication, negative, reciprocal, and division tables for $\mathbf{Z}_6$ and $\mathbf{Z}_7$.

Exercise 6.2 Insert a `counter` and increment it after every multiplication and division in your code for Exercise 5.5, and show the number of multiplications and divisions in `exteuclid(a,n)` is roughly $\log(n)$, by returning `counter/math.log(n)`.

Exercise 6.3 Given an integer a in $\mathbf{Z}$, let a_n be the corresponding modular integer in $\mathbf{Z}_n$. Show $(a+b)_n = a_n + b_n$. Here the + on the left is in $\mathbf{Z}$, while the + on the right is in $\mathbf{Z}_n$.

Exercise 6.4 Continuing the previous exercise, show $(ab)_n = a_n b_n$. Here the multiplication on the left is in $\mathbf{Z}$, while the multiplication on the right is in $\mathbf{Z}_n$.

Exercise 6.5 Continuing the previous exercise, show $(a/b)_p = a_p/b_p$ if p is a prime and a/b is an integer. Here the division on the left is in $\mathbf{Z}$, while the division on the right is in $\mathbf{Z}_p$.

Exercise 6.6 Use Theorem 6.5.2 to show $(a+b)^p = a^p + b^p$ in $\mathbf{Z}_p$.

Exercise 6.7 Add the special method `__sub__(self,other)` to the class `modint` so `m1 - m2` returns $a - b$ mod n, where

```
>>>  (a,n) == (m1.value,m1.modulus)
>>>  (b,n) == (m2.value,m2.modulus)
```

Exercise 6.8 Continuing the previous exercise, add `__pow__(self,e)` to `modint` so that `m**e` returns $a^e \mod n$, computed by repeatedly dividing e by 2 (§6.3).

Exercise 6.9 Continuing the previous exercise, add a method `reciprocal(self)` to `modint` so that `m.reciprocal()` returns the reciprocal of $a \mod n$ (use Exercise 5.5).

Exercise 6.10 Continuing the previous exercise, add `__div__(self,other)` to `modint` so `m1/m2` returns $a/b \mod n$.

Exercise 6.11 Continuing the previous exercise, add a method `legendre(self)` to `modint` so that `m.legendre()` returns the Legendre symbol $(a|n)$.

Exercise 6.12 Let a and b be in $\mathbf{Z}_n$. Show a/b exists in $\mathbf{Z}_n$ if and only if $g = \gcd(b, n)$ divides a.

Exercise 6.13 Write out the proof of Euler's theorem in complete detail.

Exercise 6.14 Show $(p-1)/2$ is odd if and only if $p \equiv 3 \mod 4$, and $(p-3)/2$ is odd if and only if $p \equiv 1 \mod 4$.

Exercise 6.15 Write the web server private key (p, q) (§6.8) in decimal. How many decimal digits do p and q have? Compute $n = pq$.

Exercise 6.16 Continuing the previous exercise, show that e is prime and $e = 2^{2^k} + 1$ for some k. Hence e and $(p-1)(q-1)$ are relatively prime.

Exercise 6.17 Continuing the previous exercise, compute the reciprocal d of e mod $(p-1)(q-1)$ (Exercise 1.14).

Exercise 6.18 Let p be an odd prime. Show that there are an equal number of quadratic residues and quadratic nonresidues (0 is neither).

Exercise 6.19 Write a function `yes_or_no(a,p)` that returns $\sqrt{a} \mod p$ if it exists, and `'no'` otherwise.

Exercise 6.20 Count the number of even integers between p and $p/2$: Show that

$$|\{k : p/2 < 2k < p\}|$$

is even for $p = 8n \pm 1$ and odd for $p = 8n \pm 3$.

Exercise 6.21 Show 3 is a quadratic residue mod p if and only if $p = 12n \pm 1$, and a quadratic nonresidue if and only if $p = 12n \pm 5$.

Exercise 6.22 Show for p prime odd,

$$\left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}.$$

Exercise 6.23 Use

$$\left(\frac{-3}{p}\right) = \left(\frac{-1}{p}\right)\left(\frac{3}{p}\right)$$

to show -3 is a quadratic residue mod p if and only if $p = 6n + 1$, and a quadratic nonresidue if and only if $p = 6n - 1$.

Exercise 6.24 *Twin primes* are primes of the form p, $p + 2$. Show

$$\left(\frac{p}{p+2}\right) = \left(\frac{2}{p}\right).$$

Exercise 6.25 Write a function `legendre(a,p)` that computes $(a|p)$ by factoring a into prime factors, then using reciprocity to invert. For example, starting with `legendre(11,29)` leads to, up to ± signs,

```
legendre(29,11)  -> legendre(7,11) ->  legendre(11,7)
%:
->  legendre(4,7) -> legendre(2,11) times legendre(2,11)
```

Chapter 7
Rationals **Q**

Recall the only integers having reciprocals are ± 1 (§4.6). The rationals **Q** are obtained by enlarging **Z** so that all nonzero rationals have reciprocals. Therefore the properties that characterize the *rationals* **Q** are

1. additive commutativity (7.1.1)
2. additive associativity (7.1.2)
3. existence of zero (4.1.3)
4. existence of negatives (7.1.4)
5. multiplicative commutativity (7.1.5)
6. multiplicative associativity (7.1.6)
7. existence of one (7.1.7)
8. distributivity (7.1.8)
9. every nonzero rational has a reciprocal (7.1.9)
10. minimality (7.1.10)
11. zero is neither positive nor negative (7.1.11)

Every integer is a rational, but not every rational is an integer. Because of this, minimality here (7.1.10) is not the same as minimality (4.5.1) for integers. The latter is valid for integers, while the former is valid for rationals. Similarly for the other axioms.

In §A.3, we show there is essentially one set **Q** equipped with addition and multiplication operations satisfying these axioms.

7.1 Basic Properties

Rationals may be added. As for integers, addition of rationals satisfies *commutativity*,

$$x + y = y + x \qquad \text{for every } x \text{ and } y, \tag{7.1.1}$$

and *associativity*,

$$(x + y) + z = x + (y + z) \qquad \text{for every } x \text{ and } y \text{ and } z. \tag{7.1.2}$$

There is a special rational 0, called *zero*, satisfying

$$x + 0 = 0 + x = x \qquad \text{for every } x. \tag{7.1.3}$$

If a rational x is not zero, then we say x is *nonzero*. To every rational x corresponds its *negative* $-x$: This is the rational y which when added to x yields zero:

$$x + y = x + (-x) = 0. \tag{7.1.4}$$

Rationals may be multiplied. As for integers, multiplication of rationals satisfies *commutativity*,

$$xy = yx \qquad \text{for every } x \text{ and } y. \tag{7.1.5}$$

and *associativity*,

$$(xy)z = x(yz) \qquad \text{for every } x \text{ and } y \text{ and } z. \tag{7.1.6}$$

There is a special rational 1, called *one*, satisfying

$$x \cdot 1 = 1 \cdot x = x \qquad \text{for every } x, \tag{7.1.7}$$

and addition and multiplication are related by *distributivity*,

$$x(y + z) = xy + xz \qquad \text{for every } x \text{ and } y \text{ and } z. \tag{7.1.8}$$

Because the above properties are structurally the same as the corresponding properties of integers, all consequences of these properties, derived in Chapter 4, remain valid for rationals. For example, the rationals 0 and 1 are unique, $-(-x) = x$ for every rational, etc.

Moreover, the rationals that are sums of ones

$$\pm(1 + 1 + \cdots + 1)$$

behave exactly like integers, and so *every integer is a rational*.

What differentiates rationals from integers is

$$\textit{every nonzero rational has a reciprocal.} \tag{7.1.9}$$

This is the *division* property, as it allows us to *define* division as multiplication by the reciprocal,

$$\frac{x}{y} = x \cdot \frac{1}{y}, \qquad y \neq 0.$$

Then the reciprocal of x/y is y/x since

$$\frac{x}{y} \cdot \frac{y}{x} = x \cdot \frac{1}{y} \cdot y \cdot \frac{1}{x} = x \cdot \frac{1}{x} \cdot y \cdot \frac{1}{y} = 1, \qquad x \neq 0, y \neq 0.$$

Let $\mathbf{Q}^+$ be the set of all rationals x that are ratios a/b of positive integers a and b. These are the *positive* rationals. We write $x > 0$ to mean x is a positive rational.

Let $\mathbf{Q}^-$ be the set of all negatives of the positive rationals. These are the *negative* rationals. We write $x < 0$ to mean x is negative.

As for integers, the rationals are *minimal*, in the sense

every rational is positive, or negative, or zero, (7.1.10)

and

zero is neither positive nor negative. (7.1.11)

This is called *positivity*, because, as for integers, it forces $\mathbf{Q}^+$ and $\mathbf{Q}^-$ to be disjoint.

A rational $x = a/b$ is in *lowest form* if a and b are relatively prime, and $b > 0$. When $x = a/b$ and $y = c/d$ are in lowest form, $a/b > c/d$ if and only if $ad > bc$. It is easy to then check

the sum of positive rationals is positive, (7.1.12)

and

the product of positive rationals is positive, (7.1.13)

Then < and > follow the same rules we saw for integers, since these rules depend only on the above four properties (§4.5).

Here are some consequences of the above properties. By commutativity and associativity, the reciprocal of xy is the product of the reciprocals of x and y,

$$\frac{1}{xy} = \frac{1}{x} \cdot \frac{1}{y}, \qquad x \neq 0, y \neq 0,$$

since

$$(xy) \cdot \left(\frac{1}{x} \cdot \frac{1}{y} \right) = x \cdot \frac{1}{x} \cdot y \cdot \frac{1}{y} = 1 \cdot 1 = 1.$$

From this follows

$$\frac{x}{y} = \frac{xz}{yz}, \qquad y \neq 0, z \neq 0,$$

since

$$\frac{xz}{yz} = xz \cdot \frac{1}{yz} = x \cdot \frac{1}{y} \cdot z \cdot \frac{1}{z} = \frac{x}{y}.$$

It follows that we may always reduce a rational to lowest form. By distributivity,

$$\frac{a+b}{c} = \frac{a}{c} + \frac{b}{c}, \qquad c \neq 0. \tag{7.1.14}$$

From this follows

$$\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}, \qquad b \neq 0, d \neq 0, \tag{7.1.15}$$

and

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}, \qquad b \neq 0, d \neq 0. \tag{7.1.16}$$

These consequences are valid for a, b, c, d rationals, not just integers.

Rationals are built in Python as follows

```
>>>  from fractions import Fraction
>>>  a = Fraction(6,10)
>>>  type(a)
fractions.Fraction
>>>  a
Fraction(3,5)
>>>  print(a)
3/5
```

Rationals in Python are always converted into lowest form,

```
>>> b = Fraction(123456789,987654321)
>>> print(b)
13717421/109739369
>>> print(a+b)
27434842/109739369
>>> b.denominator
109739369
>>> b.numerator
13717421
```

If $x = a/b$ is a rational, there is a unique integer $q = \lfloor x \rfloor$, the *floor* of x, satisfying $q \le x < q+1$. This follows from the division algorithm (§5.2): Let q be the quotient upon dividing a by b, and let $r = a - bq$ be the remainder. Since $0 \le r < b$, q satisfies $q \le x < q+1$. The floor can also be characterized as

$$\lfloor x \rfloor = \max\{k \text{ in } \mathbf{Z} : k \le x\}.$$

For any integer n, $\lfloor n \rfloor = n$, and

$$\lfloor x + n \rfloor = \lfloor x \rfloor + n. \tag{7.1.17}$$

Moreover, when x is not an integer, a moment's thought shows

$$\lfloor -x \rfloor = -\lfloor x \rfloor - 1. \tag{7.1.18}$$

for any integer n.

7.2 Farey Series

farey series ford circles

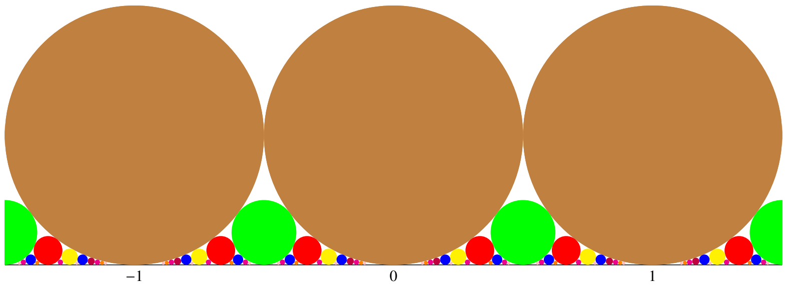

Fig. 7.1 Ford circles.

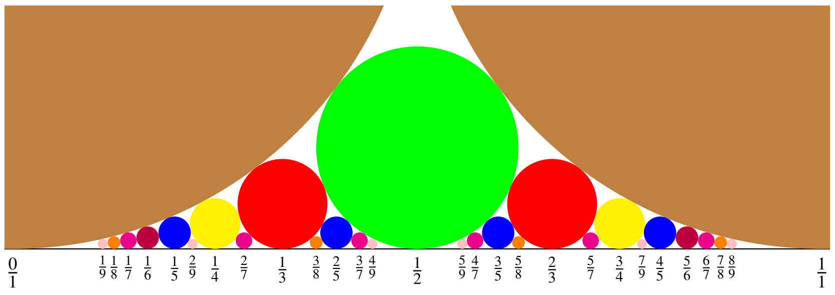

Fig. 7.2 Ford circles zoomed in.

7.3 Existence of $\sqrt{2}$

A *quadratic equation* is the degree two equation

$$f(x) = ax^2 + bx + c = 0, \tag{7.3.1}$$

where a, b, and c are integers. The simplest such equation is $x^2 = 2$, whose solution $x = \sqrt{2}$ is rightly called *the square root of* 2. The following result goes back at least to the Greeks.

Theorem 7.3.1 *There is no rational $x = a/b$ whose square is* 2.

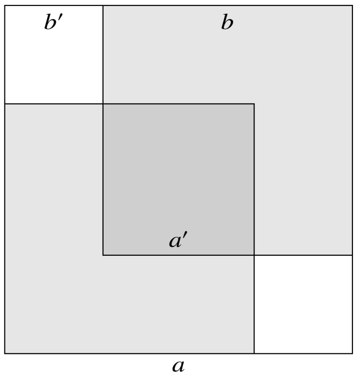

Fig. 7.3 $\sqrt{2}$ is irrational.

Proof There are many proofs of this. One standard proof follows Figure 7.3. If $x = a/b$ were a rational satisfying $x^2 = 2$, then $a^2 = 2b^2$. In Figure 7.3, the large square has sides a and the two shaded squares have sides b. The two shaded squares overlap and fail to cover the large square.

Let the sides of the overlap be a', and let the sides of the uncovered squares be b'. Since the area a^2 of the large square equals the sum $2b^2$ of the areas of the shaded squares, the overlap area a'^2 equals the sum $2b'^2$ of the the areas of the uncovered squares.

So starting with integers (a, b) satisfying $a^2 = 2b^2$, we found smaller integers (a', b') satisfying $a'^2 = 2b'^2$. But this argument can be repeated indefinitely without limit, contradicting the fact that the positive integers have a minimum at 1. Thus our hypothesis $a^2 = 2b^2$ must be impossible. □

This proof can be presented purely algebraically, by noting $b' = a - b$ and $a' = a - 2b' = a - 2(a - b) = 2b - a$. Then

$$\begin{aligned} a'^2 - 2b'^2 &= (2b - a)^2 - 2(a - b)^2 \\ &= 4b^2 - 4ab + a^2 - 2a^2 + 4ab - 2b^2 = 2b^2 - a^2 = 0. \end{aligned}$$

The above proof technique is called *infinite descent*, since it produces an infinite descending sequence of positive integers $a > a' > a'' > \dots$, which shouldn't happen, leading to a contradiction.

Given this, how do we compute the square root of 2? Since there is no rational square root, the best we can do is compute rational approximations a/b to $\sqrt{2}$, in the sense $(a/b)^2$ differs from 2 by smaller and smaller amounts.

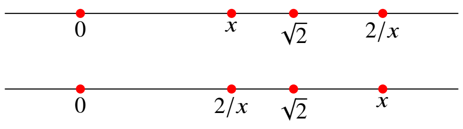

Fig. 7.4 The Babylonian algorithm.

The method we describe, used by the Babylonians thousands of years ago, starts with a guess x. If the guess is wrong, it's either greater than $\sqrt{2}$ or less than $\sqrt{2}$. If $x > \sqrt{2}$, then $2/x$ is less than $\sqrt{2}$, since

$$x > \sqrt{2} \qquad \implies \frac{2}{x} < \frac{2}{\sqrt{2}} = \sqrt{2}.$$

If $x < \sqrt{2}$, then $2/x > \sqrt{2}$, since

$$x < \sqrt{2} \qquad \implies \frac{2}{x} > \frac{2}{\sqrt{2}} = \sqrt{2}.$$

Hence, in either case, x and $2/x$ are on opposite sides of $\sqrt{2}$ (try this with $x = 1, 2, 3, \dots$), and their average,

$$x' = \frac{1}{2}\left(x + \frac{2}{x}\right),$$

will be a closer and hence better guess for $\sqrt{2}$. Starting with $x = 1$, and repeating this procedure, we get $x, x', x'', \dots,$

$$\frac{1}{1}, \frac{3}{2}, \frac{17}{12}, \frac{577}{408}, \frac{665857}{470832}, \frac{886731088897}{627013566048}, \dots$$

The number 577/408 appears on a Babylonian clay tablet, see [1]. This sequence is the output of `sqrt2(7)`, where

```
0>>  def sqrt2(n):
1>>     x = Fraction(1,1);
1>>     for i in range(1,n):
2>>        print(x,end=', ')
2>>        x = (x+2/x)/2
```

We show as x runs through this sequence, x^2 approaches 2. By the binomial theorem,

$$x'^2 - 2 = \frac{x^2}{4} - 1 + \frac{1}{x^2} = \frac{1}{4x^2}(x^2-2)^2,$$

so $x'^2 \geq 2$. Since all the approximations are at least 1,

$$0 \leq x'^2 - 2 \leq \frac{1}{4}(x^2 - 2)^2.$$

Hence (Exercise 7.13) starting with $x_0 = 1$, after n steps,

$$0 \leq x_n^2 - 2 \leq \frac{4}{4^{2^n}}, \qquad n \geq 1. \tag{7.3.2}$$

Thus x_n^2 approaches 2.

A nonzero integer D is *squarefree* if it is not divisible by a square a^2. For example, $6 = 2 \cdot 3$ and -1 are squarefree, but $12 = 3 \cdot 2^2$ is not. By the fundamental theorem of arithmetic, this is the same as saying $\pm D$ is a product of distinct positive primes, or $D = -1$.

A *quadratic irrational* is a number of the form

$$x = a + b\sqrt{D}, \tag{7.3.3}$$

where a and b are rationals, and D is a squarefree integer. The *rational part* of x is a, the *irrational part* of x is b, and the *discriminant* is D. When $b \neq 0$, quadratic irrationals are not rationals (Exercise 7.5).

Since (7.3.1) is solved by the *quadratic formula*

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{7.3.4}$$

the solutions of (7.3.1) are quadratic irrationals. Conversely, every quadratic irrational x satisfies (7.3.1) for some integers a, b, c (Exercise 7.4).

Since D is squarefree,

$$a + b\sqrt{D} = c + d\sqrt{D} \qquad \text{if and only if} \qquad a = c \text{ and } b = d.$$

In particular, $a + b\sqrt{D} = 0$ if and only if $a = 0$ and $b = 0$.

Sums and products of quadratic irrationals are given by

$$(a + b\sqrt{D}) + (c + d\sqrt{D}) = (a + c) + (b + d)\sqrt{D},$$

and

$$(a + b\sqrt{D}) \cdot (c + d\sqrt{D}) = (ac + bdD) + (bc + ad)\sqrt{D}.$$

Since D is not the square of a rational, $a^2 - b^2D$ is never zero, unless $a + b\sqrt{D} = 0$, and reciprocals are given by

$$\frac{1}{a + b\sqrt{D}} = \frac{a}{a^2 - b^2D} - \frac{b}{a^2 - b^2D}\sqrt{D}.$$

Strictly speaking, we have not defined $\sqrt{D}$ for D squarefree. For example, above we have shown there is no rational x satisfying $x^2 = 2$, but we haven't discussed the existence of such a number. Similarly, Exercise 7.5 shows there is no rational x satisfying $x^2 = D$, but we haven't discussed the existence of $\sqrt{D}$. This issue can be avoided by considering quadratic irrationals as ordered pairs $x = (a, b)$ of rationals subject to the above addition and multiplication rules (§B.1).

Quadratic irrationals may be manipulated in Python by creating the class

```
0>> class quadirr:
1>>   def __init__(self,discriminant,rational,irrational):
2>>      self.discriminant = discriminant
2>>      self.rational = rational
2>>      self.irrational = irrational
>>>
>>>   a = Fraction(3,5)
>>>   b = Fraction(11,17)
>>>   D = 21
>>>   x = quadirr(D,a,b)
>>>   x.rational
3/5
>>>   x.irrational
11/17
>>>   x.discriminant
21
```

Here `discriminant` is a squarefree `int`, and `rational` and `irrational` are `Fractions`.

When the discriminant D is positive, quadratic irrationals are ordered, $x < y$, just like **Z** and **Q**, by considering $\sqrt{D}$ to be positive. Then there are three cases when $a + b\sqrt{D}$ is positive.

The first case is when a and b are both nonnegative, with at least one of a or b positive. The second case is when a is positive and b is negative. In this case, $a + b\sqrt{D} > 0$ implies $a > -b\sqrt{D} > 0$ which implies $a^2 > b^2 D$. The third case is when a is negative and b is positive. In this case, $a+b\sqrt{D} > 0$ implies $b\sqrt{D} > -a > 0$ which implies $a^2 < b^2 D$.

Carrying out a similar analysis for $a + b\sqrt{D} < 0$, we end up with Figure 7.5, where the entries $\pm(a^2 - Db^2)$ indicate the sign of $a + b\sqrt{D}$ is the product of the sign of a with the sign of $a^2 - Db^2$.

The key point is that Figure 7.5 allows us to define the sign of $a + b\sqrt{D}$ without any reference to $\sqrt{D}$. We now turn the argument around and *define* $x = a + b\sqrt{D}$ as *positive* or *negative* following Figure 7.5. Then $\sqrt{D} = 0 + 1 \cdot \sqrt{D} > 0$ follows from Figure 7.5, and $x > 0$ if and only if $-x < 0$ (Exercise 7.7). Moreover, the following consequences hold, when $D > 0$:

1. every quadratic irrational is positive, or negative, or zero,
2. zero is neither a positive nor a negative quadratic irrational,

3. the sum of positive quadratic rationals is positive,
4. the product of positive quadratic rationals is positive.

The first two follow immediately from Figure 7.5. The derivation of the last two is quite involved, because there are many cases to check. We establish them in §B.1.

Defining $x < y$ to mean $y - x$ is positive, $<$ and $>$ follow the same rules we saw for integers, since these rules depend only on the above four properties (§4.5).

Why do this at all? Why not just define $\sqrt{D}$ to be positive and work from there? Because (Exercise 7.5) there is no rational square root of D, and we cannot take such an appoach and stay within the realm of algebra.[1]

a b	$+$	0	$-$
$+$	$+$	$+$	$Db^2 - a^2$
0	$+$	0	$-$
$-$	$a^2 - Db^2$	$-$	$-$

Fig. 7.5 Sign of $x = a + b\sqrt{D}$.

In §7.4, we will need the floor of a quadratic irrational x. The *floor* $\lfloor x \rfloor$ of a quadratic irrational x is the unique integer q satisfying

$$q \le x < q + 1.$$

Theorem 7.3.2 *Let D be positive. The floor $\lfloor x \rfloor$ of a quadratic irrational $x = a+b\sqrt{D}$ exists.*

Proof By Exercise 7.8, $\sqrt{D} < D$. Let

$$S = \{k \text{ in } \mathbf{Z} : k \le x\}$$

S is nonempty, since $k = -|a| - |b|D < a - |b|\sqrt{D} \le a + b\sqrt{D} = x$. Also, S is bounded above, since any k in S satisfies $k \le |a| + |b|D$. By Exercise 4.8, $q = \max S$ exists. By definition of S, $q \le x$. By definition of max, $q+1$ is not in S, so $q+1 > x$.□

Note the definition here of $\lfloor x \rfloor$ is consistent with that in §7.1.

[1] $\sqrt{D}$ is a well-defined real number only after we've defined real numbers, which we do not discuss.

7.4 Continued Fractions

Another method of approximating $x = \sqrt{2}$ by rationals is to write $x^2 = 2$ as $x^2 - 1 = 1$. Factoring $x^2 - 1 = (x - 1)(x + 1)$ yields

$$x - 1 = \frac{1}{x + 1},$$

or

$$x + 1 = 2 + \frac{1}{x + 1}.$$

Looking at this equation recursively, and inserting the left side into the denominator,

$$x + 1 = 2 + \cfrac{1}{2 + \cfrac{1}{x + 1}} = 2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{x + 1}}} = 2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{x + 1}}}}.$$

These are *continued fractions*. Since we may repeat this indefinitely, we end up expressing

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \dots}}}}}.$$

as an infinite continued fraction.

We are interested in continued fractions of the form

$$x = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cfrac{1}{q_4 + \cfrac{1}{q_5 + \dots}}}}}, \tag{7.4.1}$$

where $q_0, q_1, q_2, q_3, \dots$ are integers. Such continued fractions are *periodic* if the sequence $q_1, q_2, q_3, \dots$ eventually repeats. For example, the continued fraction for $\sqrt{2}$ repeats, since $q_n = 2$, $n \geq 1$, and $q_0 = 1$.

Conversely, we may use the repeating pattern in

$$x = 1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2 + \dots}}}}}, \tag{7.4.2}$$

to find x. Thinking of the continued fraction as infinite descending stairs, walking down the stairs two steps results in the same stairs, so

$$x = 1 + \cfrac{1}{2 + \cfrac{1}{x}}.$$

Clearing denominators leads to the quadratic equation

$$2x^2 - 2x - 1 = 0.$$

Applying (7.3.4), we obtain

$$x = \frac{1}{2} + \frac{\sqrt{3}}{2},$$

where we take the plus sign since x is positive. We want to generate continued fractions for numbers x of the form (7.3.4).

Given a rational x, how are continued fractions such as (7.4.1) generated? They are generated as follows. Let $q_0 = \lfloor x \rfloor$ (§7.1) and let $x_0 = x$. If x is not an integer, then

$$x_0 = q_0 + \frac{1}{x_1} \tag{7.4.3}$$

defines a rational $x_1 > 1$. If x_1 is not an integer, we may repeat this and

$$x_0 = q_0 + \cfrac{1}{q_1 + \cfrac{1}{x_2}} \tag{7.4.4}$$

defines a rational $x_2 > 1$. Continuing, we obtain rationals $x_0, x_1, x_2, \dots$ satisfying $x_0 = x$ and

$$x_n = q_n + \frac{1}{x_{n+1}}, \qquad n \geq 0. \tag{7.4.5}$$

Since the denominators of $x_0, x_1, x_2, \dots$ are a strictly decreasing sequence of positive integers, at some point x_n must be an integer q_n, and this process must terminate. We obtain a finite continued fraction

$$x = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cfrac{1}{\ddots + \cfrac{1}{q_n}}}}},$$

with $q_1 \geq 1$, $q_2 \geq 1$, $\dots$, $q_n \geq 1$. Thus a finite continued fraction is equivalent to the euclidean algorithm (§5.3).

We wish to repeat this process starting with a quadratic irrational x. We assume $D > 0$, so $\lfloor x \rfloor$ is defined as in §7.3. Given any quadratic irrational x that is not a rational, let $x_0 = x$, $q_0 = \lfloor x_0 \rfloor$, and define x_1 by (7.4.3). Since $0 < x_0 - q_0 < 1$, $x_1 = 1/(x_0 - q_0) > 1$. Thus x_1 is a quadratic irrational that is not a rational, and $q_1 = \lfloor x_1 \rfloor \geq 1$.

Repeating, we define x_2 by (7.4.4). Then x_2 is a quadratic irrational that is not a rational, and $q_2 = \lfloor x_2 \rfloor \geq 1$. Continuing, we obtain quadratic irrationals $x_1, x_2, \dots$, none of which are rationals, satisfying (7.4.5). Since none of $x_1, x_2, \dots$ are rationals, this process does not end, and $q_n \geq 1$ for $n \geq 1$. This is what is meant by (7.4.1).

Note that $q_0 = \lfloor x_0 \rfloor$ depends only on x_0, hence if $x_n = x_0$ for some n, then $q_n = q_0$, and the sequence of quotients repeats after n steps, $q_n = q_0$, $q_{n+1} = q_1$, $q_{n+2} = q_2, \dots$.

Theorem 7.4.1 (Lagrange's Theorem) *If x is a quadratic irrational that is not a rational, the quotients $q_0, q_1, q_2, \dots$ in the continued fraction* (7.4.1) *eventually repeat: There are positive integers n and m with $q_{n+k} = q_{n+m+k}$, $k \geq 0$. Conversely, if* (7.4.1) *eventually repeats, then x is a quadratic irrational that is not a rational.*

Proof Assume x satisfies (7.3.1) with $D = b^2 - 4ac$. Then D is not a square of a rational. By (7.4.5) (Exercise 7.14), $x_0 = x, x_1, x_2, \dots$ satisfy

$$f_n(x_n) = a_n x_n^2 + b_n x_n + c_n = 0, \qquad n \geq 0,$$

where $f_0(x) = f(x)$ and a_n, b_n, c_n satisfy

$$\begin{aligned} a_{n+1} &= a_n q_n^2 + b_n q_n + c_n \\ b_{n+1} &= 2a_n q_n + b_n \\ c_{n+1} &= a_n. \end{aligned} \tag{7.4.6}$$

The discriminant of $f_n(x)$ doesn't depend on n (Exercise 7.15),

$$b_n^2 - 4a_n c_n = D, \qquad n \geq 1. \tag{7.4.7}$$

Since $a_{n+1} = f_n(q_n)$, it follows that $a_n \neq 0$, hence $|a_n| \geq 1$ and $q_n \geq 1$, and x_n, $n \geq 1$, are quadratic irrationals that are not rational.

We show a_n does not eventually become positive: It is not the case that $a_n > 0$ for all n beyond some point. If this were so, then by (7.4.6), since $q_n \geq 1$, $b_{n+1} \geq 2 + b_n$, so b_n eventually becomes positive, and by (7.4.6) again, so does c_n. But this is impossible, since $x_n > 0$ and $f_n(x_n) = 0$. Thus a_n does not eventually become positive.

Similarly, a_n does not eventually become negative: If this were so, then by (7.4.6), since $q_n \geq 1$, $b_{n+1} \leq -2 + b_n$, so b_n eventually becomes negative, and by (7.4.6) again, so does c_n. But this is impossible, since $x_n > 0$ and $f_n(x_n) = 0$. Thus a_n does not eventually become negative.

Thus the sequence $a_0, a_1, a_2, \dots$ switches sign infinitely often, so there are infinitely many n with $a_n a_{n-1} < 0$, or, by (7.4.6), with $a_n c_n < 0$. Let E be the set of n's for which $a_n c_n < 0$. For n in E,

$$b_n^2 \leq b_n^2 - 4a_n c_n = D.$$

Also $4|a_n c_n| = -4a_n c_n \leq D$, so $4|a_n| \leq 4|a_n c_n| \leq D$ and $4|c_n| \leq 4|a_n c_n| \leq D$, hence for n in E,

$$4|a_n| + 4|c_n| + b_n^2 \leq 3D.$$

Since there are only finitely many integers a, b, c satisfying this last inequality, we conclude there are only finitely many distinct polynomials $f_n(x)$ for n in E. Since each polynomial has at most two roots, and E is infinite, there are at least two integers n and $n + m$ with $x_n = x_{n+m}$. As we saw above, this forces the q's to eventually repeat.

We now turn to the proof of the converse. A *linear fractional transformation* is a map of the form

$$y = \frac{ax + b}{cx + d}, \tag{7.4.8}$$

for some integers a, b, c, and d. By Exercise 7.17, if x and y are related by

$$y = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cfrac{1}{\ddots + \cfrac{1}{q_m + \cfrac{1}{x}}}}}}, \tag{7.4.9}$$

then they are related by (7.4.8), for some integers a, b, c, and d.

To prove the converse of the Theorem, if the continued fraction for x repeats eventually, then for some m and n,

$$x_n = q_n + \cfrac{1}{q_{n+1} + \cfrac{1}{q_{n+2} + \cfrac{1}{q_{n+3} + \cfrac{1}{\ddots + \cfrac{1}{q_{n+m-1} + \cfrac{1}{x_n}}}}}}.$$

But this implies

$$x_n = \frac{ax_n + b}{cx_n + d},$$

for integer coefficients a, b, c, d. Multiplying out leads to a quadratic, hence x_n is a quadratic irrational, which implies x is a quadratic irrational. This completes the proof. □

Exercises

Exercise 7.1 Prove (7.1.14), (7.1.15), (7.1.16).

Exercise 7.2 Prove (4.5.3), (4.5.4), and Theorems 4.5.3 and 4.5.4 for a, b, c rational.

Exercise 7.3 Show that x given by the quadratic formula (7.3.4) solves the quadratic equation (7.3.1).

Exercise 7.4 Show x is a quadratic irrational if and only if x satisfies (7.3.1) for some integers a, b, c with $b^2 - 4ac \neq 0$.

Exercise 7.5 If D is a squarefree integer, then D is not the square of a rational.

Exercise 7.6 The *norm* of a quadratic irrational $x = a + b\sqrt{D}$ is $N(x) = a^2 - b^2 D$. Show that $N(xy) = N(x)N(y)$.

Exercise 7.7 Let x be a quadratic irrational with positive discriminant. Show that $x > 0$ if and only if $-x < 0$.

Exercise 7.8 Let $D > 1$. Using only Figure 7.5, show $\sqrt{D} < D$.

Exercise 7.9 Add the special methods `__eq__`, `__str__`, `__add__`, `__sub__`, `__mul__` to `quadirr`. These methods should raise an error when the discriminants are not the same.

Exercise 7.10 Add the special method `__pow__` to `quadirr` that returns `x**e` (Exercise 6.8).

Exercise 7.11 Add the special methods `__lt__`, `__gt__`, `__le__`, `__ge__` to `quadirr` to return the order operators $<$, $>$, $\leq$, $\geq$. These methods should raise an error when the discriminants are not the same or not positive.

Exercise 7.12 Write a recursive function `sqrt(a,n)` that computes the square root of $a > 0$ by applying

$$x' = \frac{1}{2}\left(x + \frac{a}{x}\right)$$

n times.

Exercise 7.13 Derive (7.3.2) by induction.

Exercise 7.14 Derive (7.4.6) by induction.

Exercise 7.15 Derive (7.4.7) by induction.

Exercise 7.16 If $y = f(x)$ and $y = g(x)$ are linear fractional transformations (7.4.8), so is their composition (§3.5). Show that $y = q + x$ and $y = 1/x$ are linear fractional transformations.

Exercise 7.17 If x and y are related by (7.4.9), then x and y are related by (7.4.8) (use the previous exercise and induction).

Appendix A

In this appendix, we establish the uniqueness of $\mathbf{Z}$, $\mathbf{Z}_n$, and $\mathbf{Q}$. We introduce *rings* and *fields*, objects that are at the basis of many areas of mathematics, and we show

1. $\mathbf{Z}$ is the unique minimal ring in which zero is neither positive nor negative.
2. $\mathbf{Z}_n$ is the unique minimal ring in which zero is either positive or negative.
3. $\mathbf{Q}$ is the unique minimal field in which zero is neither positive nor negative.
4. $\mathbf{Z}_p$ is the unique minimal field in which zero is either positive or negative.

A *ring* is a set equipped with addition and multiplication operations satisfying

1. additive commutativity (4.1.1)
2. additive associativity (4.1.2)
3. existence of zero (4.1.3)
4. existence of negatives (4.2.1)
5. multiplicative commutativity (4.3.1)
6. multiplicative associativity (4.3.2)
7. existence of one (4.3.3)
8. distributivity (4.4.1)

Thus a ring is a set equipped with addition and multiplication operations that agree with (most of) our intuitive understanding of how numbers behave.

We only consider rings where *zero is not one*, to rule out the uninteresting case $\mathbf{Z}_1$ (Exercise A.1).

A ring X is *totally ordered* if there is a subset X^+, the *positive numbers* in X, such that with the negative numbers X^- being the negatives $-X^+$ of the positive numbers, we have

1. every number is either positive, negative, or zero,
2. zero is neither positive nor negative,
3. the sum of positive numbers is positive,
4. the product of positive numbers is positive.

(A.0.1)

A ring is *minimal* if every number in the ring is a sum of ones (this is defined precisely in the next section), or the negative of a sum of ones, or zero. Thus a minimal ring is in some sense a smallest possible ring, as every number in the ring is then the result of adding or subtracting ones (§4.5).

If a sum of ones is never zero, we say the ring has *modulus zero*. In §A.1, we show there is essentially a unique minimal ring with modulus zero, the *integers* $\mathbf{Z}$.

In a minimal ring X with modulus zero, if we define the positive numbers X^+ to be the sums of ones, then it follows immediately from the definitions that X is totally ordered. Thus $\mathbf{Z}$ is totally ordered.

The *negation* of (4.5.2) is[1]

zero is either positive or negative

If a ring satisfies this, then $n1 = 0$ for some positive integer n. The least such n is the *modulus* of the ring.

In §A.2, we show there is essentially one minimal ring with modulus n, for each $n \geq 1$, the *modular integers* $\mathbf{Z}_n$.

Note, because of the presence of the modulus n, we can't even *define* $\mathbf{Z}_n$ without first studying and agreeing upon what $\mathbf{Z}$ is.

A *field* is a ring where

every nonzero number has a reciprocal.

A field is *minimal* if every number in the field is a ratio of sums of ones, or the negative of such a ratio, or zero. Thus a minimal field is in some sense a smallest possible field. In §A.3, we show there is essentially a unique minimal field with modulus zero, the *rationals* $\mathbf{Q}$.

In a minimal field X with modulus zero, if we define the positive numbers X^+ to be the ratios of sums of ones, then it follows immediately from the definitions that X^+ is totally ordered. Thus $\mathbf{Q}$ is totally ordered.

We also show there is essentially a unique minimal field with modulus p, $\mathbf{Z}_p$, if p is prime, and no such field otherwise. In §B.1, we study the field $\mathbf{Q}(\sqrt{D})$, which is not minimal.

[1] Since $0 = -0$, this is the same as saying zero is both positive and negative.

A.1 Uniqueness of Z

All the results derived in Chapter 4 depend only the axioms listed there. As such, the results in Chapter 4 are valid for any minimal ring X with modulus zero.

Let X and Y be rings and suppose f is a map from X to Y (§3.5). We say f is a *ring map* if $f(1) = 1$ and

$$f(x + x') = f(x) + f(x'), \tag{A.1.1}$$

and

$$f(xx') = f(x)f(x'), \tag{A.1.2}$$

for all x, x' in X. Here the addition and multiplication on the left are taken in X, while the addition and multiplication on the right are taken in Y. In other words, a map f is a ring map if f maps numbers in X to numbers in Y in a manner consistent with the addition and multiplication operations in X and Y. If f is also bijective (§3.5), then f is an *isomorphism*. If rings X and Y share an isomorphism, we say X and Y are *isomorphic*. Isomorphic rings are the same ring except for the relabelling $x \mapsto f(x)$.

If X and Y are isomorphic rings, then they have the same modulus (Exercise A.8).

If two rings have the same modulus, they need not be isomorphic; they may be vastly different. However, if the two rings are both minimal, Theorems A.1.2 and A.2.1 guarantee they are isomorphic.

If f is a ring map from X to Y, by (A.1.1), $f(x) = f(x + 0) = f(x) + f(0)$, so $f(0) = 0$. Moreover, by choosing $x' = -x$ in (A.1.1), a ring map satisfies

$$f(-x) = -f(x), \qquad x \text{ in } X. \tag{A.1.3}$$

A subset S of a ring X is *inductive* if S contains 1 and contains $x + 1$ whenever it contains x.

If X is any ring, the set of *positive numbers* X^+ is the smallest inductive subset of X. The set of *negative numbers* X^- is the set of negatives of the positive numbers in X. X is *minimal* if X is the union of X^+, X^- and zero, and X *has modulus zero* if zero is neither positive nor negative. As in Chapter 4, having modulus zero is equivalent to X^+, X^- and zero being disjoint.

The main result is that, up to isomorphism, there is only one minimal ring with modulus zero. The key step towards the main result, that there is a unique ring map f from **Z** to any ring X, is almost obvious, since we have no choice but to set $f(1) = 1$, $f(1 + 1) = f(1) + f(1)$, etc. Minimality and modulus zero then guarantee f is well-defined.

For the actual proof of the key step, we follow our nose through the definition of a map (§3.5). Here is the key step.

Theorem A.1.1 *Let X be a minimal ring with modulus zero and let Y be any ring. Then there is a unique ring map f from X to Y.*

Proof The goal of the proof is to exhibit a ring map f between X and Y. Since X^+, X^- and zero are disjoint and their union is X, we may define f on each of X^+ and X^- separately, and set $f(0) = 0$. By (A.1.3), f is determined on X^- by what it does on X^+. Therefore we begin by constructing a map f from X^+ to Y.

We say a relation f (§3.5) between X^+ and Y is *inductive* if

1. f contains $(1, 1)$, and
2. f contains $(x + 1, y + 1)$ whenever f contains (x, y).

Here $(1, 1)$ is the ordered pair consisting of 1 in X together with 1 in Y. Similarly, in the ordered pair $(x + 1, y + 1)$, the + on the left is in X, and the + on the right is in Y.

Let f be the *smallest* inductive relation between X^+ and Y. The plan is to show f is a map from X^+ to Y satisfying (A.1.1) and (A.1.2) for all x, x' in X^+.

Step 1

The source of f is X^+: To see this, we show the source of f is inductive. Since $(1, 1)$ is in f, 1 is in the source of g. If x is in the source of f, pick y in Y such that (x, y) is in f. Since f is inductive, $(x + 1, y + 1)$ is in f, so $x + 1$ is in the source of f, hence the source of f is inductive. Since X^+ is the smallest inductive subset of X, the source of f equals X^+.

Step 2

f is a map from X^+ to Y: For each x in X, let A_x be the set of y in Y related to x under f. For f to be a map, corresponding to each x in X^+, there must be only one y in Y related to x under f, in other words, we must show $|A_x| = 1$ for all x in X.

Let A be the set of x in X^+ for which this is not so, so let A be the set of x in X^+ such that $|A_x| > 1$. We claim A is empty. If not, by the well-ordering principle (Theorem 4.7.1), let $a = \min A$. There are two cases, $a = 1$ or $a > 1$.

If $a = 1$, there is a $c \neq 1$ with (a, c) in f. If $a > 1$, then $a - 1$ is not in A, so there is a unique number in Y, call it b, with $(a - 1, b)$ in f. Since f is inductive, $(a, b + 1)$ is in f. Since a is in A, there is a c with (a, c) in f and $c \neq b + 1$. Thus, in either of the cases $a = 1$ or $a > 1$, we made a specific choice of c with (a, c) in f.

Let g be f with (a, c) removed, $g = f - \{(a, c)\}$. Then g is a relation between X^+ and Y that is strictly smaller than f. We show g is an inductive relation. First, $(1, 1)$ is in g, since we didn't remove it. Next, if (x, y) is in g, then $(x + 1, y + 1)$ is in f, because f is inductive and $g \subset f$. If $x + 1 \neq a$, then $(x + 1, y + 1)$ is in g. If $x + 1 = a$, then $a > 1$ and $x = a - 1$ and y must equal b, hence $y + 1 = b + 1 \neq c$, so $(x + 1, y + 1) = (a, b + 1)$ is in g.

This shows $(x + 1, y + 1)$ is in g in either case, so g is inductive. But f was chosen to be the smallest inductive relation between X^+ and Y. This contradiction shows A is empty, hence f is a map from X^+ to Y.

Step 3

$f(1) = 1$ and $f(x+1) = f(x) + 1$ for x in X^+: This follows from the definition of f in Step 0, since f is a map and $y = f(x)$ is shorthand for (x, y) in f.

Step 4

(A.1.1) *holds for all x, x' in X^+:* Let S be the set of x in X^+ such that (A.1.1) holds for all x' in X^+. By the previous step, 1 is in S. If x is in S, then (A.1.1) holds. If x' is in X^+, then $x' + 1 = 1 + x'$ is in X^+, so

$$\begin{aligned} f((x+1)+x') &= f(x+(1+x')) = f(x) + f(1+x') \\ &= f(x) + f(x') + f(1) = f(x+1) + f(x'), \end{aligned}$$

so $x + 1$ is in S. Hence S is inductive. Thus $S = X^+$, which is the same as saying (A.1.1) holds for all x, x' in X^+.

Step 5

(A.1.2) *holds for all x, x' in X^+:* Let S be the set of x in X^+ such that (A.1.2) holds for all x' in X^+. Since $f(1) = 1$, 1 is in S. If x is in S, then (A.1.2) holds. By Step 4,

$$\begin{aligned} f((x+1)x') &= f(xx' + x') = f(xx') + f(x') \\ &= f(x)f(x') + f(x') = (f(x)+1)f(x') = f(x+1)f(x') \end{aligned}$$

so $x + 1$ is in S. Hence S is inductive. Thus $S = X^+$, which is the same as saying (A.1.2) holds for all x, x' in X^+.

Step 6

Summarizing, *f is a map from X^+ to Y satisfying* (A.1.1) *and* (A.1.2) *for all x, x' in X^+*. Now enlarge f to a map between X and Y by defining $f(0) = 0$ and $f(x) = -f(-x)$ for x in X^-.

Step 7

(A.1.1) *holds for all x, x' in X:* If $x > 0$ and $x' > 0$, this is Step 4. If $x > 0$ and $x' < 0$, there are three cases. First, if $x + x' > 0$, then by Step 4,

$$f(x) = f(x+x') + f(-x')$$

Hence

$$f(x+x') = f(x) - f(-x') = f(x) + f(x').$$

Second, if $x + x' < 0$,

$$f(x+x') = -f(-x-x') = -f(-x) - f(-x') = f(x) + f(x').$$

Third, if $x + x' = 0$,

$$f(x+x') = 0 = f(x) - f(x) = f(x) + f(-x) = f(x) + f(x').$$

Thus (A.1.1) is valid if $x > 0$ and $x' < 0$. If $x < 0$ and $x' < 0$, by Step 4,

$$f(x+x') = -f(-x-x') = -(f(-x) + f(-x')) = f(x) + f(x').$$

If either of x or x' is 0, (A.1.1) is immediate. This establishes (A.1.1) for all x and x' in X.

Step 8

(A.1.2) *holds for all x, x' in X:* If $x > 0$ and $x' > 0$, this is Step 5. If $x > 0$ and $x' < 0$, then $xx' < 0$, so by Step 5, $f(xx')$ equals

$$-f(-xx') = -f(x(-x')) = -f(x)f(-x') = f(x)f(x').$$

If $x < 0$ and $x' < 0$, then $xx' = (-x)(-x') > 0$, so by Step 5, $f(xx')$ equals

$$f((-x)(-x')) = f(-x)f(-x') = (-f(x))(-f(x')) = f(x)f(x').$$

When either of x or x' is 0, (A.1.2) is immediate. This establishes (A.1.2) for all x and x' in X.

Step 9

f is the unique ring map from X to Y. If g is another ring map from X to Y, let S be the set of x in X^+ satisfying $f(x) = g(x)$. Since f, g are ring maps, $f(1) = 1 = g(1)$. Since f, g are ring maps, S is inductive, hence $S = X^+$. Thus $f(x) = g(x)$ for all x in X^+. If x is in X^-, then $f(x) = -f(-x) = -g(-x) = g(x)$, so $f(x) = g(x)$ for all x in X^-. Since $f(0) = g(0)$, we conclude $f = g$. □

Now we can prove

Theorem A.1.2 *Let X and Y be minimal rings with modulus zero. Then X and Y are isomorphic.*

Proof By Theorem A.1.1, there is a unique ring map f from X to Y and a unique ring map g from Y to X. Let $h(x) = g(f(x))$ be their *composition*. Then h is a ring map from X to X, since

$$\begin{aligned} h(x+x') &= g(f(x+x')) = g(f(x)+f(x')) \\ &= g(f(x)) + g(f(x')) = h(x) + h(x') \end{aligned}$$

and

$$h(xx') = g(f(xx')) = g(f(x)f(x')) = g(f(x))g(f(x')) = h(x)h(x').$$

But the identity $k(x) = x$ is a ring map from X to X. By uniqueness, $h = k$, so $g(f(x)) = x$ for all x in X. Similarly, by switching the roles of f and g, $f(g(x)) = x$ for all x in X. Thus g is the inverse of f, hence f is bijective. We conclude X and Y are isomorphic. □

By this theorem, it is meaningful to define $\mathbf{Z}$ as the unique minimal ring with modulus zero. Since all minimal rings with modulus zero are isomorphic, we may refer to any of them as $\mathbf{Z}$. With this understood, Theorem A.1.1 can be rephrased as

Theorem A.1.3 *If X is any ring, there is a unique ring map f from $\mathbf{Z}$ to X.*

If X is any ring, then $f(n)$ is what is meant by "the sum of n ones in X". If we denote the sum of n ones in X by $n1$, since f is a ring map, we have

$$(m \pm n)1 = m1 \pm n1, \qquad (mn)1 = (m1)(n1), \qquad m, n \text{ in } \mathbf{Z}. \tag{A.1.4}$$

Let X be any ring and let f be the unique ring map from $\mathbf{Z}$ to X. Then f is not injective if and only if $m1 = k1$ for some integers k and m, which implies $(k - m)1 = 0$, or zero in X is positive. Conversely, if zero in X is positive, then f is not injective. Thus f is not injective if and only if X has modulus n for some $n \neq 0$.

On the other hand, if f is injective, then the target $f(\mathbf{Z})$ is an isomorphic copy of $\mathbf{Z}$. We conclude *every ring with modulus zero contains a copy of* $\mathbf{Z}$, given by

$$f(\mathbf{Z}) = \{n1 : n \text{ in } \mathbf{Z}\}.$$

The technique in the proof of Theorem A.1.1 can be used for many other cases, as follows.

Theorem A.1.4 *Let X be any set, let a be an element of X, and let h be a map from $X \times X$ to X. If f is a map from $\mathbf{Z}^+$ to X, then there is a unique map $g : \mathbf{Z}^+ \to X$ satisfying*

$$g(1) = a, \qquad g(n+1) = h(g(n), f(n+1)), \qquad n \geq 1. \tag{A.1.5}$$

Proof Mimicking the proof of Theorem A.1.1, we say a relation g between $\mathbf{Z}^+$ and X is *inductive* if $(1, a)$ is in g, and $(n+1, h(b, f(n+1)))$ is in g whenever (n, b) is in g. Then exactly as in that proof, one shows the smallest inductive relation between $\mathbf{Z}^+$ and X is a map g from $\mathbf{Z}^+$ to X satisfying (A.1.5), and that such a map is unique. □

If we choose $X = \mathbf{Z}$ and $h(x, y) = xa$ for x, y in $\mathbf{Z}$ (here f doesn't enter), then g is the power map,

$$g(1) = a, \qquad g(n+1) = g(n)a, \quad n \geq 1.$$

Recall (§4.7) we denote the power map as $g(n) = a^n$.

If we choose $X = \mathbf{Z}$, f the power map, and $h(x, y) = x + y$, then g is the geometric sum map,

$$g(1) = 1 + a, \qquad g(n+1) = g(n) + a^{n+1}, \quad n \geq 1.$$

Recall (§4.7) we denote the geometric sum map as $g(n) = a^n + \cdots + a + 1$.

A.2 Uniqueness of $\mathbf{Z}_n$

In this section, we establish

Theorem A.2.1 *Let $n \geq 1$. If X and Y are minimal rings with modulus n, then X and Y are isomorphic.*

Proof Define a relation f between X and Y by

$$f = \{(k1, k1) : k \text{ in } \mathbf{Z}\}.$$

Here $(k1, k1)$ is the ordered pair consisting of k ones in X together with k ones in Y. If $k1 = m1$ in X, then by (A.1.4), $(k - m)1 = 0$ in X. By Exercise A.2, n divides $k - m$, so $(k - m) = nt$. Hence

$$k1 - m1 = (k - m)1 = (nt)1 = (k1)(n1) = 0$$

in Y. Thus f is a map from X to Y satisfying $f(k1) = k1$ for all k in $\mathbf{Z}$. By (A.1.4), f is a ring map. Repeating the same argument with X and Y switched, we have a ring map g from Y to X satisfying $g(k1) = k1$ for k in $\mathbf{Z}$. Thus g is the inverse of f, and we conclude f is bijective. Hence X and Y are isomorphic. □

By this theorem, it is meaningful to define $\mathbf{Z}_n$ as the unique minimal ring with modulus n. Since all minimal rings with modulus n are isomorphic, we refer to any of them as $\mathbf{Z}_n$.

Let X be any ring and let f be the unique ring map from $\mathbf{Z}$ to X. If X has modulus n, then f is not injective, and the target has n elements.

We conclude *every ring with modulus n contains a copy of* $\mathbf{Z}_n$, given by

$$f(\mathbf{Z}) = \{k1 : k \text{ in } \mathbf{Z}\}.$$

A.3 Uniqueness of Q

Let X be a field with modulus zero. From §A.1, X contains a copy of the integers $\mathbf{Z}$. Analogously to what was done in Chapter 4, let X^+ be the set of ratios a/b of positive integers a, b. These are the *positive* numbers in X. Let X^- be the *negative* numbers, the negatives of the positive numbers. With this understood, a field is *minimal* if every number is either positive, negative, or zero.

If f is a ring map from a field X to a field Y, then by choosing $x' = 1/x$ in (A.1.2), we have $f(1/x) = 1/f(x)$ whenever $x \neq 0$ and $f(x) \neq 0$. Thus there is no need for a separate definition of "field map".

A minimal field is not a minimal ring. Nevertheless, we have

Theorem A.3.1 *Let X be a minimal field with modulus zero and let Y be any field with modulus zero. Then there is a unique ring map f from X to Y.*

Proof Since X is a field, X is a ring. Since X has modulus zero, there is a copy of $\mathbf{Z}$ in X. By Theorem A.1.3, there is a unique ring map f from $\mathbf{Z}$ to Y. We want to extend f from $\mathbf{Z}$ to all of X. Since Y has modulus zero, $f(m) \neq 0$ when $m \neq 0$. Define the relation F between X and Y by

$$F = \{(n/m, f(n)/f(m)) : n, m \text{ in } \mathbf{Z}, m \neq 0\}.$$

Since X is a minimal field, the source of F is X. If a, $b \neq 0$, c, $d \neq 0$ are in $\mathbf{Z}$ with $a/b = c/d$, then $ad = bc$, so $f(a)f(d) = f(ad) = f(bc) = f(b)f(c)$, hence $f(a)/f(b) = f(c)/f(d)$. Thus F is a map from X to Y, and $F(a/b) = f(a)/f(b)$ for a, b in $\mathbf{Z}$. Now for a, b, c, d in $\mathbf{Z}$,

$$\begin{aligned} F\left(\frac{a}{b} + \frac{c}{d}\right) &= F\left(\frac{ad+bc}{bd}\right) = \frac{f(ad+bc)}{f(bd)} \\ &= \frac{f(a)f(d) + f(b)f(c)}{f(b)f(d)} = \frac{f(a)}{f(b)} + \frac{f(c)}{f(d)} = F\left(\frac{a}{b}\right) + F\left(\frac{c}{d}\right) \end{aligned}$$

and

$$\begin{aligned} F\left(\frac{a}{b} \cdot \frac{c}{d}\right) &= F\left(\frac{ac}{bd}\right) = \frac{f(ac)}{f(bd)} \\ &= \frac{f(a)f(c)}{f(b)f(d)} = \frac{f(a)}{f(b)} \cdot \frac{f(c)}{f(d)} = F\left(\frac{a}{b}\right) \cdot F\left(\frac{c}{d}\right), \end{aligned}$$

so F is a ring map. Since $F(a/b) = f(a)/f(b)$ for a, b in $\mathbf{Z}$ and f is uniquely determined on $\mathbf{Z}$, F is uniquely determined. □

As a consequence,

Theorem A.3.2 *Let X and Y be minimal fields with modulus zero. Then X and Y are isomorphic.*

Proof The proof is identical to that of Theorem A.1.2. □

By this theorem, it is meaningful to define $\mathbf{Q}$ as the unique minimal field with modulus zero. Since all minimal fields with modulus zero are isomorphic, we refer to any of them as $\mathbf{Q}$. With this understood, Theorem A.3.1 can be rephrased as

Theorem A.3.3 *If X is any field with modulus zero, there is a unique ring map f from* $\mathbf{Q}$ *to X.*

Let X be any field with modulus zero and let f be the unique ring map from $\mathbf{Q}$ to X. By Exercise A.9, f is injective.

We conclude *every field with modulus zero contains a copy of* $\mathbf{Q}$, given by

$$f(\mathbf{Q}) = \{n1/m1 : n, m \text{ in } \mathbf{Z}\}.$$

We also have

Theorem A.3.4 *Let X be a minimal field with modulus p. Then p is prime and X is isomorphic to* $\mathbf{Z}_p$.

Proof If p is not prime, let $p = ab$ be a nontrivial decomposition. Then $(a1)(b1) = (ab)1 = p1 = 0$, but $a1 \neq 0 \neq b1$. Since X is a field, $1/a1$ exists, hence $b1 = (1/a1)a1b1 = 0$ contradicting $b1 \neq 0$. Thus p is prime. From the previous section, X contains $\mathbf{Z}_p$. By Theorem 6.4.3, $\mathbf{Z}_p$ is a field. Since X is a minimal field $X = \mathbf{Z}_p$.

If X is a field with modulus p, then the unique ring map $\mathbf{Z} \longrightarrow X$ does not extend to $\mathbf{Q}$. It extends only to the portion $\mathbf{Z}_{(p)}$ of $\mathbf{Q}$ consisting of rationals whose denominator is not divisible by p (Exercise A.11).

Exercises

Exercise A.1 If X is a ring where $0 = 1$, then the set X consists of one number.

Exercise A.2 Let X be any ring and suppose $n1 = 0$ for some positive integer n. If n is the least positive integer satisfying $n1 = 0$, and k is any integer satisfying $k1 = 0$, use the division algorithm to show n divides k.

Exercise A.3 Show that the modulus of a ring is either zero or is prime.

Exercise A.4 If a ring X has 2 or 3 numbers, then X is isomorphic to $\mathbf{Z}_2$ or $\mathbf{Z}_3$ respectively.

Exercise A.5 If X and Y are rings, then defining $(x, y) + (x', y') = (x + x', y + y')$ and $(x, y)(x', y') = (xx', yy')$ makes $X \times Y$ a ring.

Exercise A.6 Show that $\mathbf{Z}_2 \times \mathbf{Z}_2$ is not a minimal ring. Show the modulus of $\mathbf{Z}_2 \times \mathbf{Z}_2$ is 2. While $\mathbf{Z}_2 \times \mathbf{Z}_2$ and $\mathbf{Z}_4$ both have 4 numbers each, conclude $\mathbf{Z}_2 \times \mathbf{Z}_2$ and $\mathbf{Z}_4$ are not isomorphic rings.

Exercise A.7 Let X be a ring with modulus n, and let f be the unique ring map from $\mathbf{Z}$ to X. Show that the target $f(\mathbf{Z})$ is isomorphic to $\mathbf{Z}_n$. Conclude *every ring with modulus n contains a copy of* $\mathbf{Z}_n$.

Exercise A.8 If X and Y are isomorphic rings, then they have the same modulus.

Exercise A.9 If X is a field with modulus zero and f is the unique ring map from **Q** to X, then f is injective.

Exercise A.10 Let p be a prime and let $\mathbf{Z}_{(p)}$ denote the rationals m/n where p does not divide n. Show that the sum and product of rationals in $\mathbf{Z}_{(p)}$ are in $\mathbf{Z}_{(p)}$. Which rationals in $\mathbf{Z}_{(p)}$ have reciprocals in $\mathbf{Z}_{(p)}$?

Exercise A.11 Let p be a prime and let $\mathbf{Z}_{(p)}$ denote the rationals m/n where p does not divide n. If X is any field with modulus p, there is a unique ring map f from $\mathbf{Z}_{(p)}$ to X.

Appendix B

B.1 Quadratic Irrationals

In this section we build the set $\mathbf{Q}(\sqrt{D})$ of quadratic irrationals as a field with modulus zero. As such, $\mathbf{Q}(\sqrt{D})$ contains $\mathbf{Q}$, as it should, but it is not a minimal field. We also show, when $D > 0$, $\mathbf{Q}(\sqrt{D})$ is totally ordered (A.0.1).

In §7.3, we defined a quadratic irrational as a number of the form $a + b\sqrt{D}$, where D is a squarefree integer. Strictly speaking, this definition uses the number $\sqrt{D}$, whose existence lies outside the realm of $\mathbf{Q}$. How then do we approach quadratic irrationals, while remaining within the algebra of $\mathbf{Q}$?

The answer is to define $\mathbf{Q}(\sqrt{D})$ to be the set $\mathbf{Q} \times \mathbf{Q}$ (§3.5) of *ordered pairs* $x = (a, b)$ of rationals, and define, for $y = (c, d)$, addition and multiplication in $\mathbf{Q}(\sqrt{D})$ by

$$x + y = (a, b) + (c, d) = (a + b, c + d),$$

and

$$xy = (a, b) \cdot (c, d) = (ac + bdD, ad + bc).$$

Since these definitions involve only rationals, one can check that with these operations and $0 = (0, 0)$ and $1 = (1, 0)$, $\mathbf{Q}(\sqrt{D})$ is a ring. Moreover, since $a^2 - b^2 D$ is not zero for any (a, b) except $(0, 0)$, reciprocals are defined by

$$\frac{1}{x} = \frac{1}{(a, b)} = \left(\frac{a}{a^2 - b^2 D}, \frac{-b}{a^2 - b^2 D}\right), \qquad (a, b) \neq (0, 0),$$

turning $\mathbf{Q}(\sqrt{D})$ into a field (Exercise B.1).

This is valid for any squarefree integer D. The quadratic irrationals corresponding to $D = -1$ are the *Gaussian rationals*. The quadratic irrationals corresponding to $D = -3$ are the *Eisenstein rationals*. The quadratic irrationals corresponding to $D = 5$ are the *Dirichlet rationals*.

With these definitions, the computations in §7.3 are valid. Moreover sums and products of quadratic irrationals of the form $(a, 0)$ behave the same as sums and products of rationals, so $\mathbf{Q}(\sqrt{D})$ has modulus zero.

Given $x = (a, b)$, let $N(x) = a^2 - b^2 D$ be the *norm* of x. Then (Exercise 7.6)

$$N(xy) = N(x)N(y). \tag{B.1.1}$$

A quadratic irrational $x = (a, b)$ is *positive* if (a, b) is of the form $(+, +)$, $(+, 0)$, $(0, +)$, or $a > 0$ and $N(x) > 0$, or $a < 0$ and $N(x) < 0$. The *negative* quadratic irrationals are the negatives of the positive quadratic irrationals. Then positive and negative quadratic irrationals are defined following the table in Figure B.1, where the entries $\pm N(x)$ in the table means $x = (a, b)$ has the same sign as $N(x)$, when $a > 0$ and $b < 0$, and the same sign as $-N(x)$, when $a < 0$ and $b > 0$. (Recall $N(x) = a^2 - Db^2 \neq 0$ for x nonzero since the discriminant D is squarefree.)

With $D > 0$ and the above definition of positivity, the goal is to show $\mathbf{Q}(\sqrt{D})$ is totally ordered (A.0.1).

It follows immediately from Figure B.1 that every number is positive, or negative, or zero, and zero is neither positive nor negative.

$b \backslash a$	$+$	0	$-$
$+$	$+$	$+$	$-N(x)$
0	$+$	0	$-$
$-$	$N(x)$	$-$	$-$

Fig. B.1 Sign of $x = (a, b)$.

It remains to establish the sum and product of positive quadratic irrationals are positive. If these were to hold, then $D = (\pm\sqrt{D})^2$ would have to be positive. Thus it is necessary that $D > 0$ for $\mathbf{Q}(\sqrt{D})$ to be totally ordered.

Let $x = (a, b)$ and $y = (c, d)$ be positive quadratic irrationals. We show

$$x + y = (a + c, b + d)$$

is a positive quadratic irrational.

From Figure B.1, there are five possibilities for positive $x = (a, b)$, and five for positive $y = (c, d)$. Hence there are 25 possibilities to consider. Since addition is commutative, the number of possibilities is cut to 15. These we tabulate in Figure B.2.

The positivity of $x + y$ in each of the blue cases follows immediately from Figure B.1.

For the top left green case, $x = (a, b) = (+, -)$, $y = (c, d) = (+, +)$, and $x + y = (a + c, b + d) = (+, \pm)$, where $\pm$ is the sign of $b + d$. If $b + d \geq 0$, then $x + y$ is positive. If $b + d < 0$, we must show $N(x + y) > 0$. To check this, note $b < 0$ implies

(c, d) (a, b)	$(+, +)$	$(+, 0)$	$(0, +)$	$(+, -)$	$(-, +)$
$(+, +)$	$(+, +)$	$*$	$*$	$*$	$*$
$(+, 0)$	$(+, +)$	$(+, 0)$	$*$	$*$	$*$
$(0, +)$	$(+, +)$	$(+, +)$	$(0, +)$	$*$	$*$
$(+, -)$	$(+, \pm)$	$(+, -)$	$(+, \pm)$	$(+, -)$	$*$
$(-, +)$	$(\pm, +)$	$(\pm, +)$	$(-, +)$	$(\pm, \pm)$	$(-, +)$

Fig. B.2 Addition of positive quadratic irrationals.

$0 > b + d \geq b$ so $b^2 \geq (b + d)^2$. Also $a + c \geq a > 0$ implies $(a + c)^2 \geq a^2$. Hence

$$N(x + y) = (a + c)^2 - (b + d)^2 D \geq a^2 - b^2 D = N(x) > 0,$$

establishing the positivity of $x + y$. The same argument is valid for the other two green cases in the same row. For the bottom green cases, it's the same argument with the inequalities reversed.

For the yellow cases, $bd > 0$, $ac > 0$, and $N(x)$, $N(y)$ share the same sign. We must show $N(x + y)$ has the same sign as $N(x)$. If $N(x) > 0$ and $N(y) > 0$, then $(ac)^2 = a^2c^2 > Db^2Dd^2 = D^2(bd)^2$, or $ac > Dbd$. Similarly, if $N(x) < 0$ and $N(y) < 0$, then $ac < Dbd$. Since

$$N(x + y) = N(x) + 2(ac - bdD) + N(y),$$

the sign of $N(x + y)$ is positive in the first case and negative in the second case, establishing the positivity of $x + y$.

For the red case, $a < 0$, $b > 0$, $c > 0$, $d < 0$, and $a^2 < Db^2$, $c^2 > Dd^2$. These imply $ad > 0$, $bc > 0$, and

$$a^2d^2 < Db^2d^2 < c^2b^2,$$

hence

$$0 < ad < bc. \tag{B.1.2}$$

For the red case, there are nine possibilities, arranged in Figure B.3. The three positive red cases in Figure B.3 follow from Figure B.2.

If $a + c \leq 0$ and $b + d \leq 0$, then $0 < c \leq -a$ and $0 < b \leq -d$ so $c^2 \leq a^2$ and $b^2 \leq d^2$. But this implies $a^2d^2 \geq b^2c^2$, contradicting (B.1.2). This rules out four red cases in Figure B.3.

$a+c$ \ $b+d$	$+$	0	$-$
$+$	$+$	$+$	?
0	$+$	✕	✕
$-$	?	✕	✕

Fig. B.3 Red case in Figure B.2.

If $x + y = (a + c, b + d) = (+, -)$, then by (B.1.2),

$$1 + \frac{a}{c} > 1 + \frac{b}{d} > 0,$$

which leads to

$$(a + c)^2 = c^2 \left(1 + \frac{a}{c}\right)^2 > c^2 \left(1 + \frac{b}{d}\right)^2 > Dd^2 \left(1 + \frac{b}{d}\right)^2 = D(b + d)^2.$$

If $x + y = (a + c, b + d) = (-, +)$, then by (B.1.2),

$$1 + \frac{d}{b} > 1 + \frac{c}{a} > 0,$$

which leads to

$$(a + c)^2 = a^2 \left(1 + \frac{c}{a}\right)^2 < a^2 \left(1 + \frac{d}{b}\right)^2 < Db^2 \left(1 + \frac{d}{b}\right)^2 = D(b + d)^2.$$

This completes the ? cases in Figure B.3, which completes the analysis of the red case in Figure B.2.

Thus the sum of positive quadratic irrationals is positive. We turn now to multiplication.

Let $x = (a, b)$ and $y = (c, d)$ be positive quadratic irrationals. We show

$$xy = (ac + bdD, ad + bc)$$

is a positive quadratic irrational.

From Figure B.1, there are five possibilities for positive $x = (a, b)$, and five for positive $y = (c, d)$. Hence there are 25 possibilities to consider. Since multiplication is commutative, the number of possibilities is cut to 15. These we tabulate in Figure B.4.

By Figure B.1, xy is positive for each of the blue cases.

(c, d) / (a, b)	$(+, +)$	$(+, 0)$	$(0, +)$	$(+, -)$	$(-, +)$
$(+, +)$	$(+, +)$	$*$	$*$	$*$	$*$
$(+, 0)$	$(+, +)$	$(+, 0)$	$*$	$*$	$*$
$(0, +)$	$(+, +)$	$(0, +)$	$(+, 0)$	$*$	$*$
$(+, -)$	$(\pm, \pm)$	$(+, -)$	$(-, +)$	$(+, -)$	$*$
$(-, +)$	$(\pm, \pm)$	$(-, +)$	$(+, -)$	$(-, +)$	$(+, -)$

Fig. B.4 Multiplication of positive quadratic irrationals.

For each of the green cases, $N(x)$ is positive if $a > 0$, and negative otherwise, and $N(y)$ is positive if $c > 0$, and negative otherwise. By (B.1.1), we know the sign of $N(xy)$, and we know the sign of the rational part of xy. Since the sign of xy is the product of the sign of $ac + bdD$ and the sign of $N(xy)$, by Figure B.5, xy is positive.

For the red cases, we have x (not a) is positive, and

$$y = (c, d) = (+, +) = (+, 0) + (0, +) = y_1 + y_2$$

is the sum of positive numbers y_1 and y_2. By distributivity,

$$xy = xy_1 + xy_2,$$

and each term on the right side is positive by one of the green cases. Since the sum of positive numbers is positive, xy is positive.

Exercises

Exercise B.1 Let D be a squarefree integer. Show that $\mathbf{Q}(\sqrt{D})$ is a field.

$x = (a, b)$	$y = (c, d)$	$N(x)$	$N(y)$	$N(xy)$	$ac + bdD$
$(+, -)$	$(+, 0)$	$+$	$+$	$+$	$+$
$(+, -)$	$(0, +)$	$+$	$-$	$-$	$-$
$(+, -)$	$(+, -)$	$+$	$+$	$+$	$+$
$(-, +)$	$(+, 0)$	$-$	$+$	$-$	$-$
$(-, +)$	$(0, +)$	$-$	$-$	$+$	$+$
$(-, +)$	$(+, -)$	$-$	$+$	$-$	$-$
$(-, +)$	$(-, +)$	$-$	$-$	$+$	$+$

Fig. B.5 Green cases in Figure B.4.

Appendix C

In this appendix, Morley's Theorem is presented following the recent proof in [5].
NOT COMPLETE

C.1 Morley's Theorem

```
class point:
    # determined by its x and y coordinates

    def __init__(P,x,y):
        P.x = x
        P.y = y

    def __add__(P,v):
        # returns the translate P + v of point P by vector v
        return point(P.x + v.x,P.y + v.y)

    def __sub__(P,Q):
        # returns v satisfying P + v = Q, v = Q - P
        return vector(P.x - Q.x,P.y - Q.y)

    def rotate(P,Q,angle):
        # rotates P about Q
        return P + (Q-P).rotate(angle)

class vector:
    # determined by x and y coordinates
    # vectors based at origin
    def __init__(v,x,y):
        v.x = x
        v.y = y

    def __mul__(u,v):
        # dot product
        return u.x*v.x + u.y*v.y

    def rotate(v,angle):
        #rotation of vector about origin
        x = v.x
        y = v.y
        s = math.sin(angle)
        c = math.cos(angle)
        return vector(x*c - y*s,x*s + y*c)

def angle(u,v):
        return math.acos(u*v/math.sqrt((u*u)*(v*v)))

def draw(P,Q):
    # draws segment joining points P, Q
    Ptuple = (P.x,P.y)
    Qtuple = (Q.x,Q.y)
    return line([Ptuple,Qtuple],axes=False)

def intersect(line1,line2):
    #intersection of lines
    (P,u) = line1 # line1 passing through P, parallel to u
    (R,v) = line2
    Q = P + u   # another point on line1
    S = R + v
    import numpy as np
    A = np.array([[Q.x-P.x,R.x-S.x],[Q.y-P.y,R.y-S.y]])
    v = np.array([R.x-P.x,R.y-P.y])
    t,s = np.linalg.solve(A,v)
    return point((1-t)*P.x+t*Q.x,(1-t)*P.y+t*Q.y)

def triangle(P,Q,R):
    Ptuple = (P.x,P.y)
    Qtuple = (Q.x,Q.y)
    Rtuple = (R.x,R.y)
    return polygon([Ptuple,Qtuple,Rtuple])

@interact         # assumes SageMath
```

```
def morley(x=7,y=5): # coordinates of the third vertex
    A = point(0,0) # the first two vertices are fixed
    B = point(10,0)
    C = point(x,y)
    alpha = angle(C-A,B-A)/3
    beta = angle(A-B,C-B)/3
    gamma = angle(A-C,B-C)/3
    u = (B-A).rotate(alpha)
    v = (A-B).rotate(-beta)
    P = intersect((A,u),(B,v))
    u = (C-B).rotate(beta)
    v = (B-C).rotate(-gamma)
    Q = intersect((B,u),(C,v))
    u = (A-C).rotate(gamma)
    v = (C-A).rotate(-alpha)
    R = intersect((C,u),(A,v))
    show(
        draw(A,P) + draw(B,P) +
        draw(B,Q) + draw(C,Q) +
        draw(C,R) + draw(A,R) +
        draw(A,B) + draw(B,C) + draw(C,A)
        + triangle(P,Q,R)
    )
```

References

[1] John C. Baez, *Babylon and the Square Root of* 2 (2011), available at http://bit.ly/babylon-sqrt2.

[2] David A. Cox, *Primes of the Form $x^2 + ny^2$, Second Edition*, Wily, 2013.

[3] Allen B. Downey, *Think Python: How to think Like a Computer Scientist, Second Edition*, O'Reilly Media, Reading, MA, 2015.

[4] David Finston and Patrick Morandi, *Abstract Algebra: Structure and Application*, Springer, 2014.

[5] Eric Grinberg and Mehmet Orhon, *Morley's Trisector Theorem and the Law of Sines* (2020).

[6] Allen Hatcher, *Topology of Numbers*, Cornell University, 2014.

[7] Victor Kac and Pokman Cheung, *Quantum Calculus*, Springer, 2002.

[8] Mark Lutz, *Programming Python*, O'Reilly, 2006.

[9] Barry Mazur and William Stein, *Prime Numbers and the Riemann Hypothesis*, Cambridge University Press, 2016.

[10] William Stein, *Elementary Number Theory: Primes, Congruences, and Secrets, A Computational Approach*, Springer, 2009.

[11] J. Steinig, *A proof of Lagrange's theorem on periodic continued fractions*, J. Arch. Math **59** (1992), 21-23.

Python

Symbols

Index