Background

Computers

Computers are suited to solve problems that can be abstracted to be represented mathematically by *data*, such as measures, counts, facts, or labels, provided a method can be created to sove problems using that data. *Programs* encode methods to solve problems.

Computers process data exceptionally faster and more accurately than humans.

Certain problems are especially appropriate to solve using computers.

Python

Python is a programming language that simplifies writing programs. Python allows organizing data to make writing programs easier and writing instructions to process that data. It has several advantages over other programming languages.

- It is easier to learn than most programming languages.
- It is a widespread language, available on almost all types of computers.
- It has wide support for many common problems, and resources to learn how to solve them.
- It is easier to run and to develop programs than many languages.

Mathematical basics

You must understand various mathematical concepts to program. These sections include some programming specific information you may not have seen, you might take a look even if you are familiar with them. These include:

Concept	Description
Arithmetic	Basic operations and properties of numbers
Algebra	Solves problems using mathematical expressions
Number Systems	Representing numbers as numerals

Arithmetic

Arithmetic includes basic operations and properties of numbers.

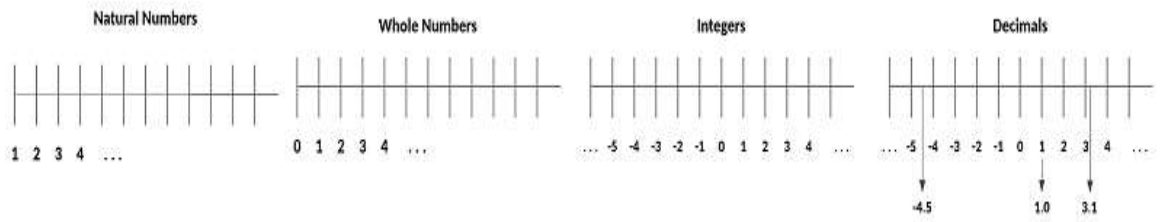
Types of numbers
Operations on numbers
Fractions
Factoring

Prime numbers
Factorials
Exponents

Types of numbers

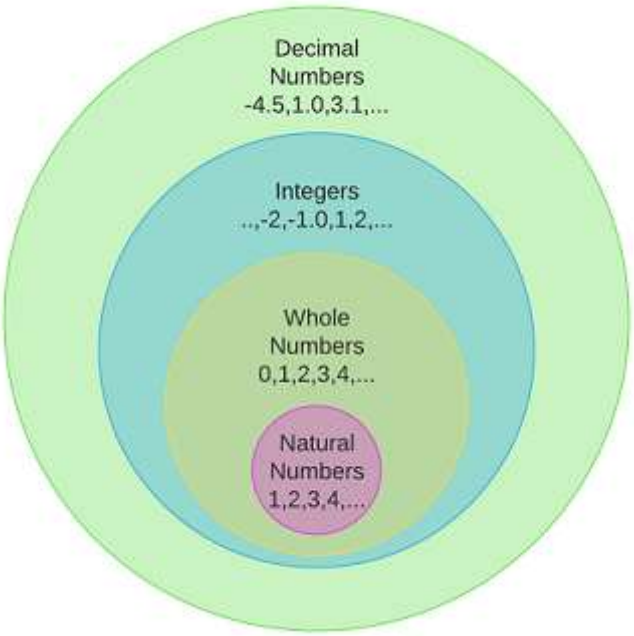
The types of numbers are listed here.

Number type	Description	Examples
Natural numbers	All positive numbers starting with 1	1, 2, 3
Whole numbers	All positive numbers starting with 0	0, 1, 2, 3, 4
Integers	All positive and negative numbers	-5, -4, -3, -2, -1, 0, 1, 2, 3, 4
Decimals	Numbers with a decimal point and fraction	-4.1, 1.0, 3.1



This shows the relationships between the types.

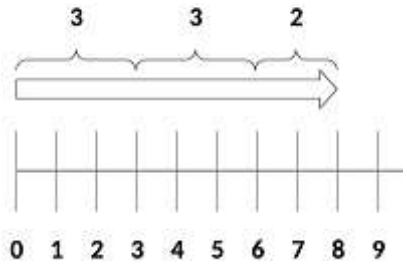
- All natural numbers are whole numbers
- All whole numbers are integers
- All integers are decimals



Operations on numbers

Kotlin allows the expected basic operations on numbers: Addition, Subtraction, Multiplication, Division, and Negation. Kotlin also supports the Modulo operation which gives the remainder of a division, after one number is divided by another. These operations use slightly different symbols in a program, for example because multiplying using x can be confused with the letter x, and trying to multiply using ' · ' is impossible because that is not a keyboard symbol. For example, `8 % 3` is 2 as shown here.

$$8 \% 3 = 2$$



These operations have specific priorities for the order in which operations are applied. Negation has the highest priority, then multiplication, division, and modulo have the next priority, then addition and subtraction are the lowest. Parentheses, `(` and `)`, are used to change the operation order.

Operation	Symbol	Program	Priority
Negation	-	-	highest
Multiplication	x or ·	*	next highest
Division	÷	/	next highest
Modulo	%	%	next highest
Addition	+	+	lowest
Subtraction	-	-	lowest

Operations of the same priority are applied left to right, so the order of operations for `3 + 6 - 2` is `(3 + 6) - 2`. For operations of different priorities, in programs it might sometimes be clearer and less error prone to use parentheses even if they not necessary, so `5 + 8 / 4 - 2` might be written as `5 + (8 / 4) - 2`.

Fractions

Fractions are a way to show a number as divisions of another number. The fraction $9/3$ shows the number of times 9 can be divided into groups of 3, which is 3. Fractions may not represent a whole number, such as $7/2$, where 2 does not divide into 7 evenly.

There are a number of rules to combine fractions.

Rule	Example
Addition	$\frac{a}{c} + \frac{b}{c} = \frac{a+b}{c}$

Rule	Example
Subtraction	$\frac{a}{c} - \frac{b}{c} = \frac{a-b}{c}$
Multiplication	$\frac{a}{c} \cdot \frac{b}{c} = \frac{a \cdot b}{c}$
Division	$\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \cdot \frac{d}{c}$

Factoring

A number N may be a multiple of whole numbers a, b, c, etc.

$$N = a \cdot b \cdot c \cdot \cdots$$

The numbers a, b, c, etc. are the *factors* of N. We say N is *evenly divisible* by a, b, c, etc.

Prime numbers

A number always at least has two factors, 1 and itself.

$$N = 1 \cdot N$$

A number that is only evenly divisible by 1 and itself is a *prime number*.

We say that when the factorization of N is

$$N = a \cdot b \cdot c \cdot \cdots$$

if all of a, b, c, etc. are prime, that is the *prime factorization* of N.

Factorials

A special case of factoring is when the factors of a number are all positive whole numbers less than or equal to n. This number is the *factorial* of n or $n!$.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 1$$

Exponents

Exponents are a way to represent repeated multiplication of the same number with fewer symbols. If a number is multiplied several times, the number of times it is multiplied can be represented with an *exponent*. For example, multiplying 2 three times can be represented using an exponent of 3:

$$2 \cdot 2 \cdot 2 = 2^3 = 8$$

We say that 2^3 is "two raised to the power 3". The value 2 is the *base* of the exponent. We can then do arithmetic on numbers with exponents. We notice that is the same as

$$(2 \cdot 2) \cdot 2 = 2^2 \cdot 2^1 = 2^3$$

where 2^1 is 2 "multiplied once".

We notice a pattern that

$$2^2 \cdot 2^1 = 2^3 = 2^{2+1}$$

so that when numbers with exponents are multiplied together, the exponents add. There is one interesting result from this. We can create a number with an exponent of 0, 2^0, or "two multiplied 0 times", where it is not obvious what that means. When we multiply that number by another number with an exponent we get

$$2^1 \cdot 2^0 = 2^{1+0} = 2$$

What value is 2^0? The only way to get that result is

$$2^1 \cdot 1 = 2$$

Therefore 2^0 must be 1. That fact will be used in the Hindu number system.

Exponent rules

This is a list of all exponent rules.

Rule	Formula
Product	$a^m \cdot a^n = a^{m+n}$
Quotient	$a^m \div a^n = a^{m-n}$
Power of a Power	$(a^m)^n = a^{mn}$
Power of a Product	$(ab)^m = a^m \cdot b^m$
Power of a Quotient	$(\frac{a}{b})^m = \frac{a^m}{b^m}$
Zero Exponent	$a^0 = 1$
Negative Exponent	$a^{-1} = \frac{1}{a^m}$
Fractional Exponent	$a^{\frac{m}{n}} = \sqrt[n]{a^m}$

Roots

The square *root* of a number n is the number which if multplied by itself will be n.

$$\sqrt{n} \cdot \sqrt{n} = (\sqrt{n})^2 = n$$

The cube root is the third root of n.

$$\sqrt[3]{n} \cdot \sqrt[3]{n} \cdot \sqrt[3]{n} = (\sqrt[3]{n})^3 = n$$

The fourth root is $\sqrt[4]{n}$, and so on. We say this following the Fractional exponent rule.

$$\sqrt[n]{a} = \sqrt[n]{a^1} = a^{\frac{1}{n}}$$

Logarithms

Logarithms are the reverse of exponents. If we say a number `N` has the value of a base `B` to the power `m`.

$$N = B^m$$

The logarithm of `N`, base `B`, is `m`.

$$logN_B = m$$

Logarithms are another convenient way to represent large numbers. For `N` = 1000000

$$log_{10}N = log_{10}1000000 = log_{10}\ 10^6 = 6$$

Algebra

Algebra finds unknown values using mathematical expressions.

Variables
Substitution rule
Expressions
Equations

Variables

Variables are symbols for unknown values. We can say the variable a has the value 1.

Substitution rule

Variables have the rule that they can be substituted by their value in any expresion. If the value of a is 1, we can say the expression

a + 2

is the same as

1 + 2

Expressions

Expressions are combinations of variables, values, and operators that give other values. An example of an expression is

a + (b · 4 - 1)

Equations

Equations say that two expressions separated by an equals sign = have the same value. We can say that the variable a has the value 1 with the equation

a = 1

There are a number of rules for equations.

Rule	Example	Note
Commutative	$a+b=b+a$	For addition
Commutative	$a \cdot b=b \cdot a$	For multiplication
Association	$a+(b+c)=(a+b)+c$	For addition
Association	$a \cdot (b \cdot c)=(a \cdot b) \cdot c$	For multiplication
Identity	$a + 0 = a$	for addition
Identity	$a - 0 = a$	for subtraction
Identity	$a - a = 0$	for subtraction
Identity	$-a + a = 0$	for subtraction
Identity	$a \cdot 1 = a$	for multiplication
Identity	$a \div 1 = a$	for division
Identity	$a \div a = 1$	for division

Note that the Commutative and Association rules do not apply for substraction and division.

These are rules for changing equations.

Rule	If	Then
Adding constants	$a = b$	$a + c = b + c$
Subtracting constants	$a = b$	$a - c = b - c$
Multiplying by constants	$a = b$	$a \cdot c=b \cdot c$
Dividing by constants	$a = b$	$a \div c=b \div c$
Inverse of adding	$a = b + c$	$a - c = b$
Inverse of subtracting	$a = b - c$	$a + c = b$
Inverse of multiplying	$a = b \cdot c$	$a \div c = b$
Inverse of dividing	$a = b \div c$	$a \cdot c = b$

We can use these rules to find the value of a variable in an equation. In the equation

$a - 2 = 1$

We can change the equation to solve for a.

Equation	Rule
$a - 2 + 2 = 1 + 2$	Adding constant 2
$a - 0 = 3$	Identity for subtraction (-2 + 2 = 0)
$a = 3$	Identity for subtraction

We can solve for equations with two variables. If we have

b - 1 = 2
a + b - 1 = 6

We can use these rules to solve for a and b.

Equation	Rule
a + 2 = 6	Substitution (2 for b - 1)
a = 6 - 2	Inverse of adding
a = 4	

Then we have

Equation	Rule
4 + b - 1 = 6	Substitution
b + 3 = 6	Subtraction
b + 3 - 3 = 6 - 3	Subtracting constant 3
b + 0 = 3	Identity for subtraction
b = 3	Identity for addition

Number systems

Number systems use symbols to represent numbers in a compact way.

Additive number systems
Exponents
Positional number systems
Binary numbers
Boolean logic

Additive number systems

The original way of counting, once the obvious limits of counting on fingers was reached, was using *tally marks* or notches on a stick, stone, or piece of clay as symbols each representing a single thing. These were added to together to create a *number*, which symbolizes a quantity of things. Though this was less limited than using fingers, these marks likewise reached a limit as numbers increased.

The Roman approach to this was to use symbols or *numerals* to count groups, then these numerals were added together to get numbers.

Number	1	2	3	4	5	9	10	50	100	500	1000
Roman	I	II	III	IV	V	IX	X	L	C	D	M

Larger numbers could be represented using fewer symbols. This is an example of adding these numberals together.

DCCXIV is 500 + 200 + 10 + 4 or 714.

Very large numbers still need quite a few symbols. Adding numbers is straightforward, but becomes cumbersome. Substracting numbers is a little harder, and multiplying and dividing numbers is harder yet.

In India, a new number system, the *Hindu* system, was created that used fewer symbols but could represent large numbers in a more compact way. Some background is needed to discuss this system.

Positional number systems

Positional number systems use fewer numerals than additive number systems, but the numerals have different values depending on their *position* in a number. The Hindu number system uses the numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. We inherited the number system from the Arabic cultures, and the number system uses Hindu-Arabic numerals or what we call Arabic numerals. The number system is called *decimal* because it uses 10 different numerals, each of which we call *digits*. We use *exponents* to show the different powers of 10 multiplied by each digit in different positions, so

$$123 = 100 + 20 + 3 = 1 \cdot 100 + 2 \cdot 10 + 3 \cdot 1 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0,$$

position	**3**	**2**	**1**
digit	1	2	3
multiplier	100	10	1
using exponents	10^2	10^1	10^0
value	100	20	3

To decide what digit goes in a position that has no value, the Hindu system had to invent the idea of a zero, the thing that is nothing and if multiplied by any number gives nothing. The concept of zero was new because it was not used in additive systems. As an example, the value that is 300 + 5 needs zero in the second position to work, so that we have

digit	**3**	**0**	**5**
position	3	2	1
multiplier	100	10	1
value	3 x 100 = 300	0 x 10 = 0	5 x 1 = 5

3 x 100 + 0 x 10 + 5 x 1 = 300 + 0 + 5 = 305

In the decimal system, adding numbers is simplified, you add each position separately. There are fewer numerals than in the Roman system, so addition is faster. For 305 + 262 = 567

position	1	2	3
	3	0	5
+	2	6	2
	---	---	---
	5	6	7

When adding numbers in a position gives more than 9, we introduce the idea of a *carry*. For 7 + 35 = 42

position	1	2
carry	1	
		7
+	3	5
	---	---
	4	2

Subtraction, multiplication, and division are similarly simplified.

Binary numbers

In the decimal system, we use a base of 10 for the exponents at each numeral position and the digits 0 through 9 for numerals. There are as many different digits as the number base. There is no reason why we can't use another number for the base, such as 2. We call a base 2 number system a *binary* system. There are two numerals in a binary system, 0 and 1. We can give the value for a binary number 10110, which we show as 10110_2 to indicate it is base 2, as

digit	1	0	1	1	0
position	5	4	3	2	1
multiplier	16	8	4	2	1
using exponents	2^4	2^3	2^2	2^1	2^0
position value	16	0	4	2	0

so the final number is 16 + 0 + 4 + 2 + 0 = 22. We can show decimal 22 as 22_{10} to show the number base.

We call each binary numeral a *binary digit* or *bit*. Computers are able to represent two values, 0 and 1, well. In electronic circuits 0 corresponds to "no power" or "no voltage" and 1 corresponds to "positive voltage". It turns out trying to electrically detect 10 voltage levels for 10 different decimal digits is unreliable.

We can add binary numbers like we do in the decimal system. We add each position separately.

$101_2 + 10_2 = 111_2$

position	1	2	3
	1	0	1
+		1	0
	---	---	---
	1	1	1

When adding binary numbers in a position gives more than 1, we introduce the idea of a *carry*.

$101_2 + 1_2 = 110_2$

position	1	2	3
carry		1	
	1	0	1
+			1
	---	---	---
	1	1	0

Subtraction, multiplication, and division are similarly simplified.

Hexadecimal numbers

Using binary numbers can also get tedious. A shorthand for binary numbers is using base 16 numbers, or *hexadecimal* numbers. Hexadecimal numbers use 16 numerals. We can use decimal digits for the first 10, 0 through 9, but we need different numberals for the last 6. We use the six letters A through F for them. We typically put `0x` in from on the number to show that it is hexademical.

decimal	binary	hexadecimal
0	0_2	0x0
1	1_2	0x1
2	10_2	0x2
3	11_2	0x3
4	100_2	0x4
5	101_2	0x5
6	110_2	0x6
7	111_2	0x7
8	1000_2	0x8

decimal	binary	hexadecimal
9	1001_2	0x9
10	1010_2	0xA
11	1011_2	0xB
12	1100_2	0xC
13	1101_2	0xD
14	1110_2	0xE
15	1111_2	0xF

The number 10110101_2 is converted to hexadecimal by breaking the number into chunks of 4 bits and converting each to a hexadecimal digit.

```
10110101 = 1011 0101 = 0xB5
```

Fixed binary number storage size

In computers, only a fixed number of bits are allocated for numbers. This limits the size of number that can be stored. Typically sizes for numbers are 8, 16, 32, or 64 bits. The term for an 8 bit number is a *byte*. The maximum value for a number for that size is the value of a binary number of that size with all 1 bits. The largest number that can be stored in 8 bits is

$$11111111_2 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

This value is $2^8 - 1$. The names and maximum values for different types of numbers of different sizes are given by

size	bits	bytes	maximum value
byte	8	1	$2^8 - 1$
short	16	1	$2^{16} - 1$
int	32	1	$2^{32} - 1$
long	64	1	$2^{64} - 1$

Numbers being limited by the number of bits requires that programs must anticipate the largest possible value and use a size large enough to store it. When a number too large for the number of bits is stored, we say an *overflow* occurs. In that case, the bits that cannot be stored are, for example, thrown away and the value is incorrect.

Boolean logic

The binary numerals 1 and 0 can conveniently represent truth values, true and false. A logic based on these values, *Boolean logic* (invented by mathematician George Boole), gives new operations on binary numbers. The basic operations are *AND*, *OR*, *NOT*, and *XOR*.

operation	description	example
AND	result is true if each operand is true	1 AND 1 = true
OR	result is true if either operand is true	0 AND 1 = true
NOT	result is the opposite of the operand	NOT 1 = false
XOR	result is false if both operands are the same, and true if both are different	1 AND 1 = false

The results of these operations can be shown in *truth tables*. This is the truth table for AND.

AND	1	0
1	1	0
0	0	0

This is the truth table for OR.

OR	1	0
1	1	1
0	1	0

This is the truth table for NOT.

NOT	
1	0
0	1

This is the truth table for XOR.

XOR	1	0
1	0	1
0	1	0

These operations are used in operations on binary numbers in programs.