{POWER.CODERS}

Intro to Programming

Contents

What is Programming?

What is Programming?

Learning to program means learning how to solve problems using code.

It consists of 2 parts:

- problem-solving
- learning programming syntax

Programming syntax

Learning a programming language and its **syntax** and semantics gives you the skills to read and **understand code** as well as to write your **own programs**.

It is an analytical activity.

Building blocks of Programming

- Data Types
- Variables
- Conditions
- Loops
- Functions

Most programming language has these building blocks. Only a different syntax.
We will use JavaScript to learn them.

What is JavaScript?

JavaScript is a **high-level** programming language. Nowadays you can (nearly) do anything with JavaScript.

JavaScript is not Java.

Java is to **Java**Script as **cat** is to **cat**fish.

Why do we learn JavaScript?

Is JavaScript a good choice as first programming language?

- It has a **low barrier to entry**
- It gives you **instant feedback**

JavaScript has also pitfalls and problems - as most programming languages do.

Problem solving

Solving a problem in programming can be defined as writing an original program that performs a particular **set of tasks** and meets all stated **constraints**.

This set of tasks is called an **algorithm**.

It is a creative activity.

That's what makes it difficult.

What is a constraint?

Think about creating a website:

- You need to use particular languages: **HTML** and **CSS**
- The website needs to be displayed on different devices: e.g. **Laptop**, **Tablet** or **Mobile phone**
- Its content needs to be searchable and readable by humans and machines: **SEO** and **Accessibility**

Constraints are the **limitiations** in how your program can perform the specific set of tasks.

How to think like a programmer?

"The biggest mistake I see new programmers make is focusing on learning syntax instead of learning how to solve problems.

—V. Anton Spraul

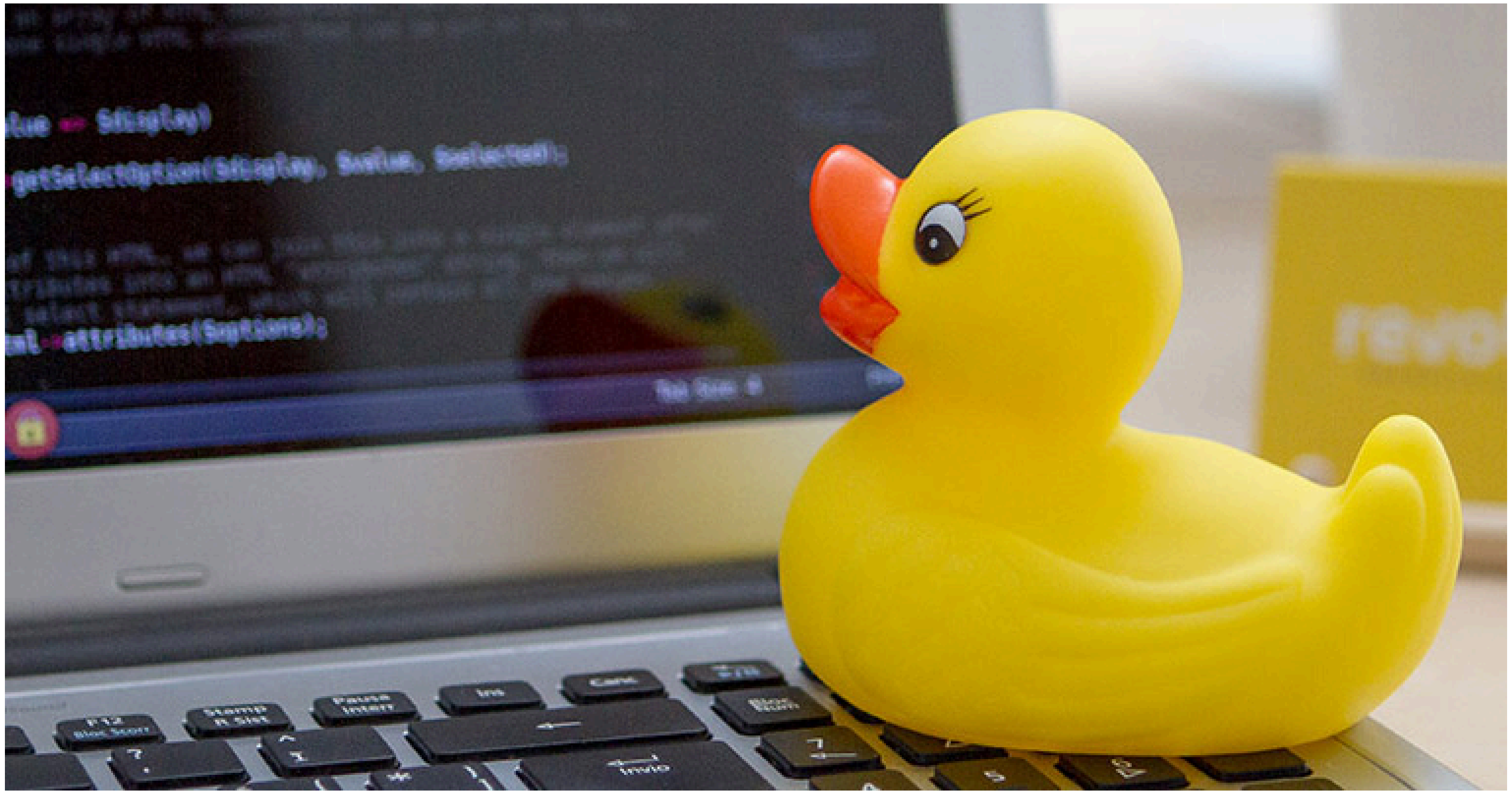
5 STEPS TO THINK LIKE A PROGRAMMER

1. Understand
2. Plan
3. Divide & conquer
4. Don't get frustrated
5. Practice

1. Understand

Read the problem several times until you can explain it to someone else in plain English.

> If you can't explain something in simple terms, you don't understand it.
>
> —Richard Feynman



Rubber duck debugging: Insights are often found by simply describing the problem aloud.

2. Plan

Take time to analyze the problem and process the information. Think **very precisely** about how you solve the problem.

Ask yourself: "Given input X, what are the steps necessary to return output Y?"

3. Divide & conquer

Do not try to solve **one big problem**. Break it down into **steps**. Steps that are so **simple** that a computer can execute them.

Once you solved every step (sub-problem), connect the dots and you will find the **solution to the original problem**. Congratulations!

Another definition of programming

Step 3 is so important that you can use it to define programming.

- Programming is the **art** of breaking a problem down into smaller problems.
- Finding the solutions to these smaller problems.
- Putting these pieces back together.

4. Stuck?

Don't get frustrated!

- **Debug:** Go step by step through your solution trying to find where you went wrong.
- **Reasses:** Sometimes we get so lost in the details of a problem that we overlook general principles. Take a step back. Look at the problem from another perspective.
- **Research:** Google is your friend. No matter what problem you have, someone has probably solved it. Find that person/ solution. In fact, do this even if you solved the problem! (You can learn a lot from other people's solutions).

How to google solutions

Don't look for a solution to the big problem, only look for solutions to sub-problems.

You will **learn and understand more**. You will need **less research and Google** next time.

Google is not about Copy & Paste, it is about learning.

BEST PRACTICE

on how to google

- Search in English
- Use more than one word
- Be precise
- Give context
- Use operators
- Check dates and up-to-dateness
- Use more than one source
- Search also images, scholar or books

5. Practice

Practice makes perfect.

The more problems you solve, the more research you do on other programmers solving the same problem, the more you think like a programmer.

"Demonstrating computational thinking or the ability to break down large, complex problems is just as valuable (if not more so) than the baseline technical skills required for a job.

—Hacker Rank (2018 Developer Skills Report)

How to solve a problem

How to cross the river?

A farmer with a fox, a goose and a sack of corn needs to cross a river. The farmer has a rowboat, but there is room for only the farmer and one of his three items. Unfortunately, both the fox and the goose are hungry. The fox cannot be left alone with the goose, or the fox will eat the goose. The goose cannot be left alone with the corn, or the goose will eat the corn.

How does the farmer get everything across the river?

Pair Exercise

Find a partner and talk through the puzzle.
You have 15 minutes to try solve the puzzle.
Describe the steps needed.

Find the constraints

- The farmer can take only one item at a time in the boat
- The fox and goose cannot be left alone on the shore
- The goose and corn cannot be left alone on the shore

List the possible actions (operations)

- Carry the fox to the far side of the river
- Carry the goose to the far side of the river
- Carry the sack of corn to the far side of the river

By enumerating all the possible operations, we can solve many problems by testing every combination of operations until we find one that works.

The more specific your operations are, the easier it is to miss possible operations.

Generalize the operations as much as possible

- Row the boat from one side to the other
- If the boat is empty, load an item from the shore
- If the boat is not empty, unload the item to the shore

This is also called **abstraction**.

Test every combination

Now we generate all possible sequences of moves, ending each sequence once it violates one of our contraints or reaches a configuration we've seen before.

Eventually we hit upon the only possible sequence, the solution of our problem.

Solution

1. Transport the goose to the other side
2. Row back alone
3. Transport the fox to the other side
4. Take goose back to original side
5. Transport the sack of corn to the other side (leave goose behind)
6. Row back alone
7. Transport the goose to the other side

How to solve a programming problem?

6 STEPS TO PROGRAM

1. Understand the problem
2. Solve the problem manually
3. Make your manual solution better
4. Write pseudocode
5. Replace pseudocode with real code
6. Simplify and optimize your code

1. Understand the problem

We had that before.

- Read the problem at least 3 times.
- Ask yourself always following questions: why? what? how?
- Explain the problem in your own words to someone else, remember the rubber duck.

You can use mind mapping tools or whiteboards to visualize your problem, e.g. a notepad or Miro.

2. Solve the problem manually

"Nothing can be automated that cannot be done manually!

Test your manual process with at least 3 different inputs. Think about what you did to find the solution.

3. Make your manual solution better

Simplify and optimize your steps. Look for patterns and see if there's anything you can generalize.

Can you reduce any steps? Are you repeating steps?

Try for brevity. The lines that you don’t write are the lines where you can be sure that they don’t have bugs.

4. Write pseudocode

Writing pseudocode line by line, before starting to code, helps you defining the structure of your code.

You can do that on a sheet of paper or as comments in your code editor.

Focus on logic and steps.

It is often easier to get started with flow charts.

What is a flow chart?

Flow charts help you visualize the different steps within a process or program.

There are different shapes for Process, Terminator, Decision, Data and On-Page Reference

Online editor to create flow charts

What is pseudocode?

Pseudocode is an informal way of writing a program. However, it is not a computer program.

It only represents the algorithm of the program in natural language and mathematical notations.

A JavaScript developer and a Cow developer would use pseudocode to communicate with each other.

Hello world in Cow

```
MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO MoO
```

Check it out here

The good thing about pseudocode

- no syntax rules
- helps you to think in steps

5. Replace pseudocode with real code

After your pseudocode is ready, translate each line into real code.

Test each step you translate as early and as thoroughly as possible. Finding a problem in a small and easy piece of code is much simpler than trying to spot it in a large program.

6. Simplify and optimize your code

This is also called **code refactoring** and we will hear more about that in the coming weeks.

> Programs must be written for people to read, and only incidentally for machines to execute.
>
> **—Gerald Jay Sussman and Hal Abelson**

Practice, practice, practice

Again, practice makes perfect.

- Debug every step
- Get feedback through code reviews
- Write useful comments

PRACTICE

The best way to learn coding is by coding as much as possible.

Books

- Think like a Programmer: An introduction to creative problem solving
- The pragmatic programmer

Programming 101

Darkness phobia

One family wants to get through a tunnel. Dad can make it in 1 minute, mama in 2 minutes, son in 4 and daughter in 5 minutes. Unfortunately, not more than two persons can go through the narrow tunnel at one time, moving at the speed of the slower one.

Can they all make it to the other side if they have a torch that lasts only 12 minutes and they are afraid of the dark?

Try it yourself first

Before you go to the next slide, try to solve the problem on your own first. Give yourself about 30 to 45 minutes.

- Understand: Visualize the problem. Ask the why?how?what?
- Plan: What are the constraints, the inputs, the steps?
- Divide: Break bigger problems into smaller ones.
- Conquer: Write pseudocode.

Possible solution

```
GetThroughTheTunnel
            Initialize maxTime as 12;
            Initialize time1 as 1, time2 as 2, time3 as 4, time4 as 5;
            Initialize maxPeople as 2;
            Initialize totalTime as 0;
            Initialize peopleAtStart as 4;
            Initialize ListOfPeople;

            Compare times with each other to sort them by speed
            => output is ordered list

            Select the 2FastestPeople (times)
            Select the 2SlowestPeople (times)

            Step 1: 2 fastest going
                IF 2FastestPeople <= maxPeople
                    totalTime = totalTime + (slowest of the 2 fastest);
                    peopleAtStart = peopleAtStart-2
                END IF
```

Add the sorting algorithm

Let's have a closer look to exercise 1 from before, finalize the pseudo-code and add how you would sort the times by speed.

Exercise 1

Allow the user to input digits, afterwards sort and print them in numerical order.

Understanding the problem

Understanding the problem is **key to solving** it.

Sometimes the first brief might not be enough.
Ask questions.

Possible questions

- Is there a limit of inputs? max. 10
- Are there several input fields (one per digit) or one? Only 1
- If there is only one, how is the correct input? Is there a seperator per digit (e.g. comma and/or space)? Official seperator is comma, space is optional

Can you think of more?

What are the constraints?

Try to find them yourself first. Take 10 minutes to find contraints and note them down.

- Numbers, positive, non-decimal
- Digits, not greater than 9
- Input is string, needs to be converted
- max 10 digits
- min 1 digit

Test input

1,2,7,3 => true
1, 2, 7, 3 => true
01226354 => false

Expected output

1237 => true
1,2,3,7 => false
2371 => false
[1,2,3,7] => false

Before you start coding...

... always define expected input and output and visualize the steps to get from input to output.

- Input: Input digits in one field, seperated by comma
- Process: Sorting algorithm
- Output: Print digits in numerical order

Pseudocode

Have a look at your pseudocode again. Can you now optimize it with inputs and outputs in mind?

Optimize your code now

POSSIBLE SOLUTION

```
SortNumbers
            INITIALIZE userInput AS String;
            INITIALIZE digitsList AS List;
            INITIALIZE output AS String;
            INITIALIZE counter;

            PROMPT "Please input minimum 2 digits and
                maximum 10 digits (0-9), separated by commas" AS userInput

            SPLIT userInput AT THE SEPERATOR "," TO digitsList
            // check for spaces, are there commas in the string

            FOR EACH item OF digitsList
                IF not a digit
                    PROMPT ("Please make sure that you only add single digits (
                END IF
            END FOR EACH

        END SortNumbers
```

Another Example

Build a tip calculator.

Do you have all the information you need to understand the problem?

Gather requirements

Ask questions to find out all you need to know, e.g.

- Can you explain how the tip should be calculated?
- What's the tip percentage? Is it a fixed value or should the user be able to modify it?
- What should the program display on the screen when it starts?
- What should the program display for its output? Do you want to see the total and the tip?

Problem statement

Once your questions are answered, summarize the problem in a statement:

> Create a simple tip calculator. The program should prompt for a bill amount and a tip rate. The program must compute the tip and then display both the tip and the total amount of the bill.

Example Output

For your understanding it is always helpful to have some example inputs and outputs in mind, before you start solving your problem.

- What is the bill? CHF 200
- What is the tip percentage? 15
- The tip is CHF 30.00
- The total is CHF 230.00

Don't start coding yet

This is the time where programmers often jump ahead and start coding.

Make sure your **program is thought through** properly, before you start.

PROCESS TO WRITE A PROGRAM

Discover inputs, processes and outputs

Even the simplest program has always inputs, processes and outputs. It is the **core definition** of a program.

Go back to the problem statement and check for nouns and verbs. **Nouns** are usually your **inputs** and **outputs**, while **verbs** describe your **processes**.

Nouns in the problem statement

- bill amount **= input**
- tip rate **= input**
- tip **= output**
- total amount **= output**

VERBS IN THE PROBLEM STATEMENT

- prompt
- compute
- display

Not every verb has to be a process. **prompt** and **display** define the GUI, how the user interacts with the program.

Result

- Inputs: **bill amount, tip rate**
- Processes: **calculate the tip**
- Outputs: **tip amount, total amount**

Driving Design with Tests

Test-driven development (TDD) is a very common practice in software developement and more and more web application development.

The essence of TDD is to think about what the **expected result** of the program is **ahead of time** and then work towards getting there.

Doing that even before you start coding, often enables you to think **beyond the initial requirements**.

Simple test plans

The easiest way to do that is creating simple test plans.

A test plan lists the program's inputs and its expected result.

TEMPLATE FOR TEST PLAN

- Inputs:
- Expected result:
- Actual result:

You start with writing down the program's inputs and expected output. After running your program you compare the expected result with the actual result.

Test plan for our tip calculator

- Inputs:
    - bill amount: 10
    - tip rate: 15
- Expected result:
    - tip: CHF 1.50
    - total: CHF 11.50

What does it tell us so far?

- We have **2** inputs and **2** outputs.
- We need to convert our tip rate from a whole number to a decimal.
- The output is formatted as a currency.

One test is not enough

Work with more than one test.

Try to think about corner and edge cases.

Corner case

A problem or situation that occurs outside of normal operating parameters.

Think of it as a hard to reach design state that happens only when **multiple conditions** happen in some combination.

Your program logic meets more than one boundary condition at once.

Edge case

A problem or situation that occurs only at an extreme (maximum or minimum) operating parameter, e.g. input.

Your program logic meets one boundary condition.

Is it very useful to add edge cases to your tests, as they are usually the place where bugs appear.

Try it yourself

Come up with several test cases on your own (15 min), then meet a peer and discuss them together (15 min).

Try to think about corner and edge cases.

For example: what happens if the bill amount is already a decimal? What happens if the tip rate is a decimal? What could be an edge case?

Constraints

Edge and corner cases can show you the constraints of your program. Also the program statement and test cases can visualize them.

- Enter the tip as a percentage. The input should be 15 for "15%", not 0.15.
- Both inputs have to be bigger than 0 and, of course, a number.
- Round fractions of a Rappen up to the next Rappen. Remember 5 Rappen is the smallest.

Algorithms

Algorithm

An **algorithm** is a step-by-step set of operations that need to be performed.

If you take an algorithm and write code to perform those operations, you end up with a **computer program**.

Start with pseudocode

Although there is no right or wrong way to write pseudocode, there are widely used terms.

- **Initialize:** set an initial value
- **Prompt:** for user input
- **Display:** for output on the screen

Try it yourself first

Before you go to the next slide, try to write the pseudocode on your own first. Start with very general descriptions of your steps. Give yourself about 30 to 45 minutes.

TIP CALCULATOR IN PSEUDOCODE

```
TipCalculator
            Initialize billAmount to 0
            Initialize tip to 0
            Initialize tipRate to 0
            Initialize total to 0

            Prompt for billAmount with "What is the bill amount?"
            Prompt for tipRate with "What is the tip rate?"

            convert billAmount to a number
            convert tipRate to a number

            tip = billAmount * (tipRate / 100)
            round tip up to nearest 5 rappen
            total = billAmount + tip

            Display "Tip : CHF" + tip
            Display "Total : CHF" + total
        End
```

What did we do?

- We set up **variables** with default values
- We asked for **user input**
- We did some **conversions** and math
- We displayed the calculated **output** on the screen

Improve the program

The next step would be to improve the code, e.g.

- Check for edge and corner cases
- Split the programs into functions
- Format the output as floats (just to be sure)

We will discuss data types, variables and functions in detail next week.

Writing the code

The final step would be to **write** the code, **comment** it properly, **test** and **debug** it right away and finally **revise** and **optimize** your code for quality and performance.

One word about comments

Never comment on **what** the code is doing, only write comments that explain **why**.

Bad example

No need to understand the code of the example yet, just have a look at the comments.

```
// This function checks whether a number is even
        let f(x) = function {
            // compute x modulo 2 and check whether it is zero
            if (x%2 == 0){
                // the number is even
                return true;
            } else {
                // the number is odd
                return false;
            }
        }
```

Good example

Notice that the name of the functions and the variables are self-explanatory. Comments are not needed.

```
let is_divisible = function(number, divisor){
            return number%divisor == 0;
        }

        let is_even = function(number){
            return is_divisible(number, 2);
        }
```

Optimizing code

Better naming and a better task breakdown make the comments from the bad example (what comments) obsolete.

Revise your code just as you would revise an essay: Sketch, write, delete, reformulate, ask others what they think. Repeat until only the crispest possible expression of your idea remains.

Important note

An algorithm needs to be presented at their level of understanding and in a language they understand.

"They" being **humans** as well as **machines**.

Real-life algorithms

- finding a word in a dictionary
- sorting a list of numbers
- operating a laundry machine
- playing a video game
- baking a cake

Like a recipe

- **Inputs** are the ingredients
- **Procedure** is the list of steps to be followed
- **Outputs** are the results of your recipe

Example

Let's play a guessing game

I am thinking of an integer between 1 and 100. Your task is to find this number by asking me questions of the form "Is your number higher, lower or equal to x" for different numbers x.

Let's do it together.

Possible ways to start

- Start with the minimum: 1 - and then go up
- Start with the maximum: 100 - and the go down
- Start in the middle: 50

In the first 2 cases, if you are unlucky, you have to ask 100 times, before you hit the right number.

What does that have to do with alogrithms?

Binary search algorithm

1. Find the midpoint of list of numbers (aka sorted array)
2. Compare the midpoint to the value of interest
3. If the midpoint is larger than the value, perform binary search on right half of the array.
4. If the midpoint is smaller than the value, perform binary search on left half of the array.
5. Repeat these steps until the midpoint value is equal to the value of interest.

This is an recursive algorithm

Another exercise

Validate a telephone number, as if written on an input form. Telephone numbers can be written as ten digits, or with dashes, spaces, or dots between the three segments, or with the area code parenthesized; both the area code and any white space between segments are optional.

Inputs, Processes and Outputs

- **Inputs:** telephone number
- **Processes:** validate the format of the phone number
- **Outputs:** true or false

Possible phone numbers

- **10 digits**, e.g. 0797899236
- With **dashes**, e.g. 079-879-9236
- With **spaces**, e.g. 079 879 9236
- With **dots**, e.g. 079.879.9236
- With **area code parenthesized**, e.g. (079) 879 9236 (optional)

Create test cases: Input and expected result.

What are the validation steps?

1. Remove the white space in the variable
2. Is the variable a number (=all digits)?
3. If yes, is the number of digits exactly 10? Then go to **step 7**
4. If no, is the variable empty? If yes, then go to **step 6**
5. If no, remove specific characters: dashes, spaces, dots and parentheses. Then go back to **step 2**
6. Not valid? Return error message
7. Valid? Return success message

Check your test cases against the validation.

Write your program

1. Start
2. Create a variable to receive the phone number
3. Clear the variable in case it is not empty
4. Prompt for the phone number
5. Store the response in the variable
6. Validate the stored response if it is a valid phone number
7. Not valid? Go back to step 3 plus error message
8. Valid? Return success message
9. End

Have all problems algorithmic solutions?

Many problems will require a combination of **algorithmic** and **heuristic** solutions.

Heuristic solutions emerge from trial and error based on knowledge and experience.

Real-life examples: Winning a chess game, making a speech at a ceremony or convention.
In programming typical problems are getting computers to speak a language or recognize patterns (Machine learning and artifical intelligence).

A QUICK WORD ON COMPLEXITY

When writing algorithms the complexety or performance is a **key factor to consider**: cost efficient, space efficient and fast. In computer science the **Big O notation** is used to describe the performance, or better the **worst-case scenario**.

GROUP EXERCISE

The GCD

Find the greatest common divisor of 2 positive integers.

In math, the GCD of 2 or more positive integers is the largest positive integer that divides each of the integers.

You have 45 min to do answer the points in the next slide.

Find the Greatest Common Divisor

- Understand the problem
- Describe input and output
- List the steps to find the GCD
- Can you find a repetitve way to get the solution?

1. Inputs: 2 positive integers, e.g. 40 (=number1) and 16 (=number2)
2. Procedure (=processes):
    1. Start
    2. Initalize variables for integers
    3. Initalize a variable for the output GCD
    4. Prompt for integer1 with "Put in your first positive number"
    5. Prompt for integer2 with "Put in your second positive number"
    6. Validate the variables to be positive integers
    7. Find the remainder: number1 % number2
    8. If the remainder == 0: then GCD = number2
    9. Else: then number1 = number2, number2 = Remainder, go back to step 3
    10. Display the output GCD of number1 and number2
    11. End

Programming principles

Manipulating data

Moving data and playing with it is the foundation of programming, e.g. send login credentials to a web sever.

Write for a human audience

You program in teams, not on your own. Be as clear as possible and as simple as possible. Clean code is more important than clever code.

Forced simplicity

Good programming is about keeping the underlying logic expressed simply, so that it can be readable to others and your future self.

Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.

—The Zen of Python

KISS PRINCIPLE

Keep **i**t **s**imple, **s**tupid.

Start small and simplify your program as much as possible, before you even start. The more complex the code, the more bugs and errors will turn up, maintaining the code as well as modifying it will be more difficult.

DRY PRINCIPLE

Don't **r**epeat **y**ourself.

Avoid duplicated and repeated code and data. Usually you can do that by abstracting and pulling code into functions, components and modules you only write once and re-use.

YAGNI PRINCIPLE

You **a**ren't **g**onna **n**eed **i**t.

Never code a functionality that you may need in the future. You might not need it after all and it makes your code more complex.

Refactor your code

There are many more principles, but the bottom line is: especially as an inexperienced programmer **code rarely comes out right the first time**.

Revisit, rewrite or even redesign parts of your code base based on programming principles.

Types of programming (programming paradigms)

Declarative programming

Declares **what** is being done rather than how it should be done.

Examples: SQL, HTML, CSS

Imperative programming

Focuses on **how** a task is done rather than what is being done.

Examples: Java, JavaScript, Ruby

PROCEDURAL PROGRAMMING

Uses **procedures**, also known as routines, subroutines or functions, to operate on data structures and carry out tasks.

Examples: Java, JavaScript, C, Pascal, Basic

Object-oriented programming

Involves **building objects** with data attributes and programming subroutines. Code reusability and inheritance are main concepts of today's dominant paradigm.

Examples: Java, JavaScript, Python, PHP, C, C++, Ruby

Multi-paradigm languages

Programming languages, like JavaScript, Java, and many more support **more than one** programming paradigm, in order to allow the **most suitable programming style** for a task.

Online resources

- Principles of algorithmic problem solving (book)
- A beginner's guide to Big O notation
- Fundamental programming principles
- 7 common programming principles
- 10 basic programming principles
- First principles of programming