

Morten Misfeldt & Morten Schultz

**Ten Ideas in Programming**

* *A minimal introduction to programming with LLMs*

Text by Morten Misfeldt, examples by Morten Schultz

**INTRODUCTION**

Ten Ideas in Programming is designed to communicate a minimal set of concepts in programming and computer science that I believe everyone should encounter as part of their education. These ideas range from concrete and low-level to more abstract. They don’t attempt to cover a specific part of introductory computer science — but they represent a minimum set of ideas that serve two main purposes:

First, they offer a practical starting point for writing code and working with writing programs. Second, they provide 10 concepts that help read and understand simple programs. The goal is for the list to be understandable in a few minutes, and for the whole material to be readable within a few hours, including examples. I am hoping that the list will also serve as an anchor to go back so you can refine your conceptual understanding as you improve your programming skills.

I focus on just ten ideas to make the material approachable and contained. I believe that a simple “idea-oriented” introduction to programming will be beneficial when learning about programming in the age of AI, either as a stepping stone for further studies in computer science or simply to become more literate about the building blocks of the digital world we all live in. The order in which the ideas are presented in are not entirely random. But the intention is that all ten ideas are introduced at once and then revisited as you gain more experience.

After introducing the ten ideas, I have augmented the material with five key practices of a programmer, highlighting the essential aspects we consider when creating computer programs.

Where the ten ideas are concepts to be understood, the practices are focused on what we do when we program and why we do it. Before I go into exemplifying the ten ideas with coding examples, I have added a short section with some suggestions on how to work with LLMs when learning to program.

**HOW WE THINK ABOUT PROGRAMMING**

The first idea is **data**. Everything digital begins with data, which can take various forms, including numbers, text, images, and sensor readings. Data is usually stored in some sort of structure or sequence (say a list) that supports computational manipulation and /or human interpretation.

The second idea is **function**. A function in programming is almost like the function machine from our primary school math class. A function is a “machine” that do stuff when asked. It can take inputs and provide outputs as its mathematical sibling, but first and foremost, it performs actions when called.

The third idea is **variables** – which are used to store and access data. Variables are important for keeping track of values and enabling dynamic behavior in programs. A variable can be a tricky concept because, despite being used across mathematics, statistics, and computer science, its meaning varies in nuance. Most importantly, you almost always assign a value to a variable in programming, in contrast to, for example, solving equations, where you search for a value of the variable that solves the problem.

*You use variables to store and refer to data, and functions to process that data in meaningful ways*

The fourth idea is (**algorithms and**) **sequential processes**—computer programs perform their actions step by step, and these sequences of steps then transform data into something useful. Each step in the sequence can perform operations, such as adding or subtracting numbers, checking conditions, or calling functions. Typically, you will break down your program idea into a set of processes that work on data and variables.

The fifth idea is control structures and **conditionals**. These are structures that guide decision-making and distinguish between different cases in computing, such that certain processes only run under specific conditions.

The sixth idea is **loops** or **iteration**, the simple but powerful concept of repetition, which enables programs to perform tasks repeatedly a set number of times or until specific conditions are satisfied.

The seventh idea is **models** — representations of real-world or imagined systems, processes, or ideas, which we use in simulations, predictions, and reasoning. Models help us make sense of complexity and allow us to test, explore, and communicate ideas through computation.

The eighth idea is **abstraction and decomposition**. This involves identifying meaningful parts of a problem, chunking them into manageable pieces, and creating the right levels of abstraction to reason about complex systems.

Closely related is the ninth idea: **functional thinking**. This means organizing your specifications and/or code in functions, inputs, and outputs. This is when and under what conditions some element of the program runs, and what the inputs and outputs of that process are. Constructing (or reconstructing) programs in reusable, well-defined functions is a great way to organize your thinking and abstractions.

Last, we turn to **data structures**—ways of organizing data to make processing efficient, meaningful, and scalable, from simple sequencing in lists and arrays to graphs.

Later in the text there are examples and further explanation of these ten ideas.

**FIVE KEY PRACTICES – HOW WE DO PROGRAMMING**

We **specify**. Writing a computer program is first and foremost stating what you want the computer to do in an understandable way. Therefore, one of the most important practices when learning to program is the practice of *specification*. This means explaining and specifying what we want a program to do, and how we want it to do it. Sometimes, we focus mostly on *what* the program should do—what output or behavior we expect. Other times, we pay more attention to *how* to build that behavior—what steps the program should follow. In all cases, explaining *what* and *how* is at the heart of programming.

We **test and debug**. When we have an idea for part of a program, we usually try it out on its own before adding it to the larger project. This helps us check if it works as expected.

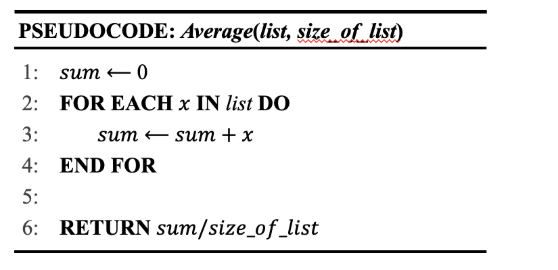
Writing code in a precise syntax is not easy, and dealing with complex logical structures is sometimes even harder. So, we make mistakes. Everyone does. Often. Making mistakes and fixing them is a normal and important part of programming. When trying to figure out what’s wrong, we localize the problem by isolating the parts we think might be causing the issue. By doing this, we can test, correct our mistakes, and improve our program step by step.

We **organize our thinking** about programs **in layers,** distinguishing **1) data**, **2) computation, and 3) interaction**. In a program that asks the user to type their name on the screen and then responds by generating a friendly message like “Hello, Morten!” The interaction layer is just this – you write your name and receive a personalized greeting. The data layer consists of the text string “Morten Misfeldt” that I provide in the input field and the computational layer is somehow able to fetch out my first name from this text string and provide it as input for the greeting that the program responds with.

These three layers—data, computation, and interaction—are distinct but closely connected. Keep these three layers in mind, and it will help you organize your thinking about the programs you are building.

**Pseudo code**

One common method for specifying programs is writing *pseudo code*—a description of your program in plain English (or whatever language you prefer).



We **document**, **modularize,** and **reuse** our code. When you're just starting out, you might write simple programs from scratch or edit examples your teacher gives you. But quickly, you’ll discover that programming is a cumulative process—programs grow over time when you work with them. In professional software development, programs often contain thousands of lines of code. Keeping track of changes and understanding what each part does is essential. But even in your first real project, you'll see the benefits of organizing and documenting your code.

If you’ve already written something that works for a particular task, then use it again for similar problems. When you do that, you will end up thinking about not only the specific problem you are engaged in (say, creating a red box with the text “No” on the screen) but rather thinking about the *type of problem* that you work on (e.g., creating a colored textbox on the screen). This way of thinking saves time, reduces errors, and helps you think more clearly about your program.

**Commenting Code**

Writing short explanations in your code helps you (and others) remember what you were trying to do. For example: What is this part supposed to do? Why are you doing it this way? This is called comments and is typically distinguished from the code by some special character like % this is a comment % in the file.

Finally, we **collaborate with our users**. Programs are developed for people — to help them do the activities they want to do in a more enriching, engaging, or more efficient way. An important part of programming is to think about use. If you’re trying to design a solution, it’s a good idea to understand the problem, and that often means understanding the practice that the program will support. This also means paying attention to how what you have developed is used by other people. There’s always a slight difference between what a designer or programmer has in mind and what users do. That’s why it’s important to consider both the imagined use during development and the actual use of the program after release or prototype.

There are many ways to do this. In modern software development, data about user behavior is often collected directly from the software. But you can also simply sit down with the users, talk to them, understand their needs, or, when your first version is ready, observe how users tweak it to fit their needs.

**CODING WITH AI/LLMs**

If you use **AI, such as ChatGPT, as a tool to write simple programs** you can experiment with generating and translating code based on the ten key concepts. Try to experiment with how these ideas manifest in actual code, and use this to solve small problems and get a sense of how the concepts work in practice.

MAKE PSEUDOCODE, SKETCHES AND FLOWCHARTS

Here it is a good idea to write a bit of simple **pseudocode** and to visualise what you want to create in order to help bridge the gap between informal ideas and real code. How will the solution look? What is the flow through the program?

*ANTICIPATE, PROMPT, AND REFLECT*

When you’re learning to program and using a large language model (like ChatGPT, Gemini or Copilot) to help you, it’s a good idea to follow a simple approach that we call the APR-approach. APR is short for *Anticipate*, *Prompt*, and *Reflect*.

Here’s how it works:

**Anticipate:** Before you even write your prompt, take a moment to think:

* What do I want my program to do?
* How should it work? What functions, variables, and data are involved?
* How are the problems broken down?

This step helps you clarify your intentions and focus your thinking. Here it sometimes makes sense to visualize your idea or write it in pseudocode.

**Prompt:** Now you can write your prompt for the language model. Be clear and specific. Tell the model what you want your program to do and how you want it to do it.

**Reflect: After** you get the response, take time to read and test the code.

* Did the program work the way you expected?
* Were there any surprises?
* What’s different between what you anticipated and what actually happened?

This reflection helps you understand both your own ideas and how the language model interprets them.

By repeating this cycle—anticipate, prompt, and reflect—you’ll develop a deeper understanding of programming and stay in control of your programming project. You’ll also learn how to work better with AI tools as a creative partner in your coding journey.

*CONCEPT MAPPING*

When you code with AI you can do more with less effort. This can potentially affect your learning negatively. Therefore you need to actively boost your conceptual understanding. We suggest that you **create mind maps** for example one that shows how the ten concepts relate to each other. You can also write short reflection or essays about how the concepts relate to one another. The goal is to encourage relational understanding and conceptual clarity—not just to know what each term means, but how they connect, overlap, and support one another.

**TWO CODING TASKS FOR A BEGINNER THAT USES AI/LLMs**

**Task 1**

Write a sentence and then make a program that changes the order of the words. Try reversing the order or making it random. Consider what the program needs to do and how you can utilize variables or specific data structures to achieve this. It is crucial that we think through how our program handles its data, specifically how it organizes data within the program.

**Task 2**

Make a program that simulates an object falling under gravity. You can think of gravity as a constant acceleration. How can you show how fast and how far it falls over time?

**FURTHER EXPLANATION AND EXAMPLES**

*DATA*

Data is the raw material of digital technology— data is the way in which information can be represented and processed by computers. Reading, storing, transforming, and acting on data — is the core of what programs do. Whether displaying a message, controlling a robot, or analyzing sensor readings, it all starts with data.

Data comes in different types, such as text (words, names), numbers (integers, decimals), or logical values (true/false). These types help computers understand how to handle the information.

Take 43: as text, it’s two characters—4 and 3; as a natural number, it follows 42 and comes just before 44; as a floating-point (a digital approximation to a real number) number, it might mean 43.0001. There are also practical concerns: How much memory does the data use? How fast can it be accessed? Precise numbers, for instance, require more space than simple values; 43 as a natural number is cheap, whereas 43.0001 is more expensive.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Figure 1. An example in a popular block-based programming environment (*Scratch*, in this case).

**PYTHON**

x\_position = -47

y\_position = 4

**print**(x\_position + y\_position) # -43

*FUNCTION*

In programming, a function is something you call to make something happen. Different languages implement functions in different ways, but they typically involve three key ideas.

Function *as a machine*: Like the input-output machines in school math, a function takes input, does something to it, and returns an output. But in programming, both input and output can be any kind of data—numbers, text, or even more complex things.

Function *as reuse and action*: A function doesn’t always need to take an input or return an output. It can simply perform an action—like printing a poem on the screen. This makes functions a useful way to organize and reuse code.

Function *as abstraction*: Functions help break a program into meaningful parts. By wrapping up a specific task into a function, you can use it in many places. This not only reduces repetition but also helps you think more clearly about how your program is structured. In short, functions are central to programming because they do things, can take input and give output, and help us organize and abstract our code.

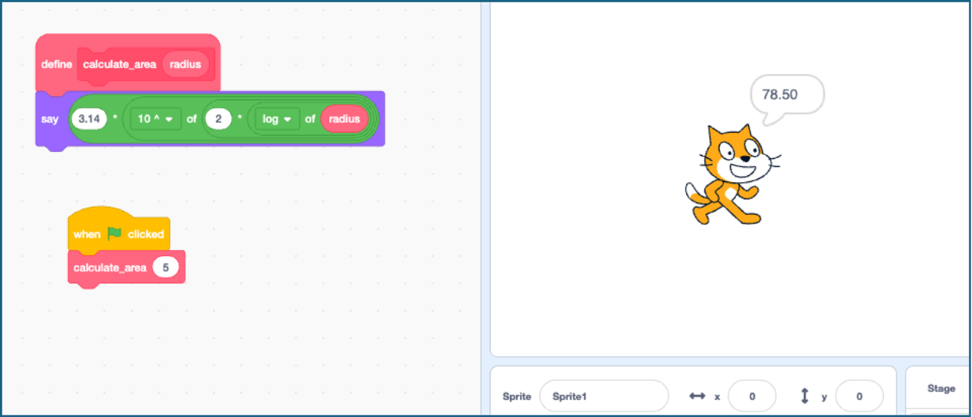


Figure 2. Creating – and using – a function in *Scratch*.

**PYTHON**

radius = 5

area = 3.14 \* radius \*\* 2

**print**(area) # 78.5

And as a function, making it reusable elsewhere (and with different inputs):

**PYTHON**

**def** **calculate\_area**(radius):

return 3.14 \* radius \*\* 2

**print**(**calculate\_area**(5)) # 78.5

**print**(**calculate\_area**(10)) # 314.0

**print**(**calculate\_area**(15)) # 706.5

*VARIABLES*

Variables are arguably my favorite concept in computer science. A variable stores information, which is what makes it possible to create programs that can handle a wide range of situations. You can read a new piece of data into a variable, and the program will respond accordingly.

Let’s say you have a variable that represents a name. Now you can write a program that says:

Hello, name. Goodbye, name. I hope this was fine, name.

And no matter what name you put in; the program will act as if it’s having a meaningful conversation with that person. Without changing the program. That’s powerful.

One quirky thing about variables, though, is that they have short-term memory. If you give them new information, they forget the old. That’s not a bug — it’s how they’re designed. A variable always stores just the latest value you gave it.

You might know the concept variable from mathematics. Variables in mathematics and programming share some important qualities. In both cases, a variable acts as a placeholder — a symbol that stands in for a value. And in both cases, that value typically belongs to a certain domain, like numbers or text. But beyond those basics, the way variables behave is quite different. In mathematics, a variable usually represents an unknown quantity that you're trying to solve for; it is like a puzzle to “*find x*”.

In programming, a variable is essentially a named box that can store a value, which can change as the program runs. That’s why something like x = x + 1, which seems nonsensical in math, is totally normal in programming: it means "*take the current value of x, add one, and store the result back into x.*" In other words, programming variables aren’t about solving equations — they’re about remembering things and updating them over time.

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype, logo

AI-genereret indhold kan være ukorrekt.

Figure 3. Defining/setting a variable in *Scratch*.

**PYTHON**

mortens\_variable = 34

**print**(mortens\_variable + 1) # 35

**print**(mortens\_variable) # 34

**print**(**type**(mortens\_variable)) # <class 'int'>

y = (mortens\_variable + 1) / 2

**print**(y) # 17.5

**print**(**type**(y)) # <class 'float'>

*SEQUENTIAL PROCESSES AND ALGORITHMS*

One key idea in programming is that computers do things *step by step*. Even when a program seems to be doing many things at once, what’s really happening is a fast series of *sequential* actions—one after the other.

That’s why breaking problems down into a clear sequence of steps is a critical skill in programming. This is how we make problems *computationally solvable*.

In some programming languages, you can set up *parallel* activities (things that run at the same time), but even these are built from smaller *sequential* instructions.

When we organize a sequence of steps in a smart way to solve a problem, we call it an **algorithm**. You might already know algorithms from everyday tasks like sorting a list, searching for something, or calculating a result.

There are two key takeaways:

**Sequential thinking**—breaking a problem down into steps—is the foundation of programming. Think of a sequence of events like a recipe things have to come in a specific order (add the rice to the risotto – before the wine and the bullion), and under certain conditions (stop adding bullion when the rice are cooked)

**Algorithmic thinking** is about doing such sequential processes in a structured, efficient way in order to solve problems.

A good example is the *Divide and Conquer*-algorithm. Imagine you're looking for a name in a long, alphabetically ordered list. You could check every name, one by one—but that would take time. Instead, you look at the name in the middle of the list and compare the first letter to your target. If your name comes before the middle one, you search the first half. If it comes after, you search the second half. Then you repeat the process. This smart way of narrowing down your search is much faster—and a great example of how algorithms can help us solve problems more efficiently.

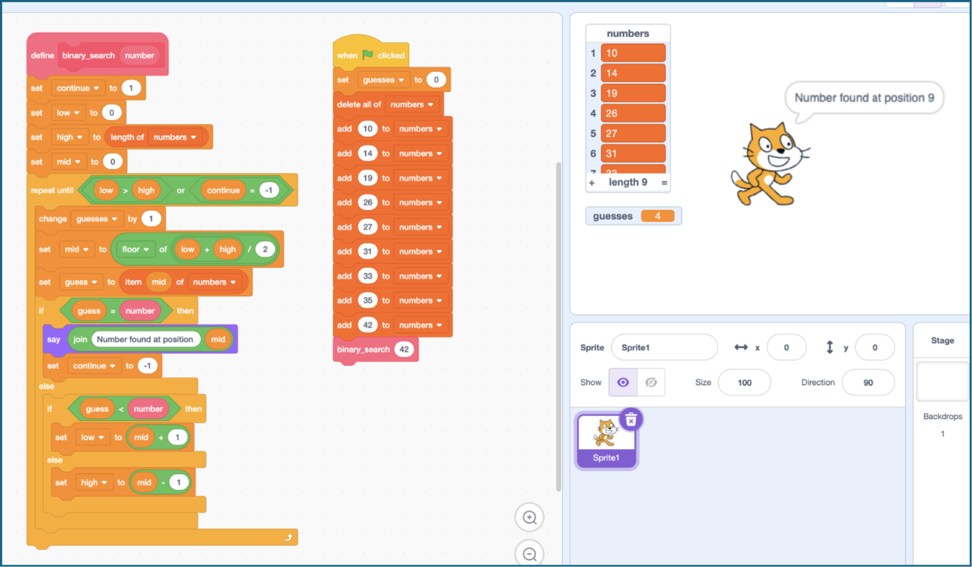


Figure 4. An example of Binary Search implemented in *Scratch* (mostly for the fun of it).

**PYTHON**

**def** **binary\_search**(numbers, number):

low = 0

high = **len**(numbers) - 1

**while** low <= high:

mid = (low + high) // 2

guess = numbers[mid]

**if** guess == number:

return f"Number found at position {mid}"

**elif** guess < number:

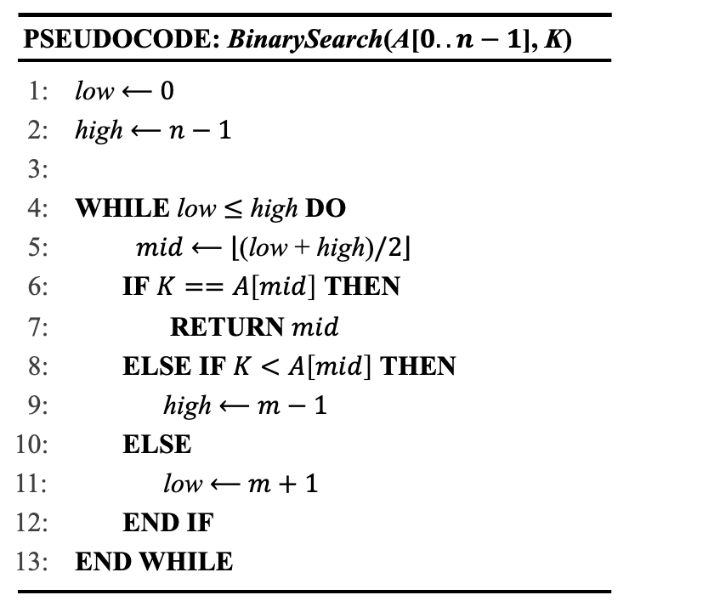
low = mid + 1

**else**:

high = mid - 1

list\_of\_numbers = [10, 14, 19, 26, 27, 31, 33, 35, 42]

**print**(**binary\_search**(list\_of\_numbers, 26)) # Number found at position 3



The two examples above are to be understood as concrete implementations of the pseudo code on the right, that is, a (somewhat) human-readable, informal description of program code or algorithms without regard for the particular syntax of specific programming languages, and thus not intended to be executed by a computer.

*CONDITIONALS/LOGICAL CONTROL STRUCTURES*

In programming, we often want our program to make decisions—to do one thing in one situation and something else in another. That’s where conditionals and control structures come in. They help us control the flow of the program.

A very common and important control structure is the if statement. An if statement tells the program:

"*Only run this part if a certain condition is true*."

You can also use an *if...else* statement. This lets your program choose between two paths:

"*If the condition is true, do this. Otherwise, do something else.*" There are different versions of control structures, but the basic idea is the same:

Use conditions to split the program into different routes. This is how your program can respond to different inputs, situations, or user actions. It’s one of the key ways to make your code dynamic and responsive.

Et billede, der indeholder tekst, skærmbillede, Font/skrifttype, logo

AI-genereret indhold kan være ukorrekt.

Figure 5. Checking for key-press – and then acting accordingly.

**PYTHON**

**if** temperature > 25:

**print**("It's warm outside!")

**else**:

**print**("Better bring a jacket.")

 Or even as follows (although not that common):

**PYTHON**

day = "Sunday"

**match** day:

**case** "Monday":

print("Time to get to work!")

**case** "Saturday" | "Sunday":

print("Weekend fun!")

**case** \_:

print("Just another regular day.")

*LOOPS AND ITERATION*

One of the most powerful ideas in programming is the loop — the ability to do something again and again and again. Loops allow your program to repeat an action many times without having to write the same code over and over.

In fact, loops work a lot like conditionals. They also use a condition — but instead of running once, they keep running as long as the condition is true.

There are different types of loops, but two common ones are:

A **for** loop is used when you know how many times you want the loop to run. For example, if you want to count from 0 to 100 (exactly 101 times—once for each number from 0 to 100).

A **while** loop keeps running as long as a condition is true. You use it when you don’t always know how many times something will repeat.

Loops are great for repetitive tasks—like going through all the items in a list (an array), checking values, or building something step by step. Learning how to use loops effectively is a big step toward writing powerful and efficient programs.

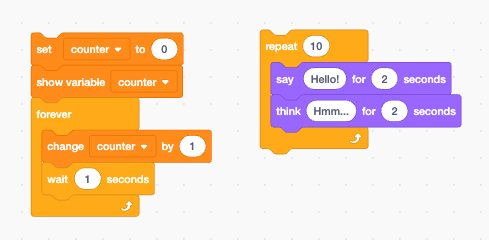


Figure 6. Examples of loops in *Scratch*.

**PYTHON**

**for** i **in** range(0, 100):

print(i) # 0, 1, 2, ... 99

 And a **while** loop:

**PYTHON**

number = 0

**while** number < 100:

**print**(number) # 0, 1, 2, ... 99

number = number + 1

*MODELS*

In computer science, the concept of a model is important. A model is a way to describe something so that a computer can work with it. This "something" could be part of the real world—like how a chain falls when you drop it—or it could be something imagined—like how a monster in a computer game behaves when you press a button.

When we create a model in computer science, we select the key components of a situation and describe them clearly, enabling the computer to utilize them for simulations, input responses, or to exhibit specific behaviors. For example, we might model what kind of sounds a robot should react to, or how a creature should move when it’s hungry.

These models help us learn about the world, test ideas, and build things that can interact with people or other systems.

*ABSTRACTION AND DECOMPOSITION*

Understanding a problem by breaking it down into meaningful, manageable components and structuring these at appropriate levels of abstraction cannot be overestimated. Not only will this process make the problem more manageable, but it will, in most cases, make the problem much easier to solve.

Consider, for instance, the task of writing a program that calculates the average score of a group of players and identifies which player, if any, scored above this average.

Applying the ideas of abstraction and decomposition to this problem, we can divide the task into distinct subcomponents. For example, one function might fetch the data, another would compute the average, another again might filter the player who exceeds it, and a fourth would format and display the result. This gives a clean, linear program structure, where each function handles a specific responsibility and passes its output to the next. Such an approach not only clarifies program logic but also enhances maintainability and reusability.

**PYTHON**

**def** **get\_scores**():

return [("Morten", 4), ("Jack", 2), ("Liv", 3)]

**def** **calculate\_average**(scores):

total = **sum**(score for \_, score in scores)

return total / len(scores)

**def** **print\_above\_average**(scores, average):

**for** name, score **in** scores:

if score > average:

**print**(f"{name} scored above average with his {score} points.")

scores = **get\_scores**() # our data

avg = **calculate\_average**(scores) # computation

**print\_above\_average**(scores, avg) # output

The above design demonstrates both decomposition, we split the program into well-defined tasks, and functional thinking, in that we express these tasks as functions that can be reused or modified independently. Notice how the program is quite easy to read from top to bottom, with each function performing one focused job. This makes it much easier to reason about, test, and maintain.

*FUNCTIONAL THINKING*

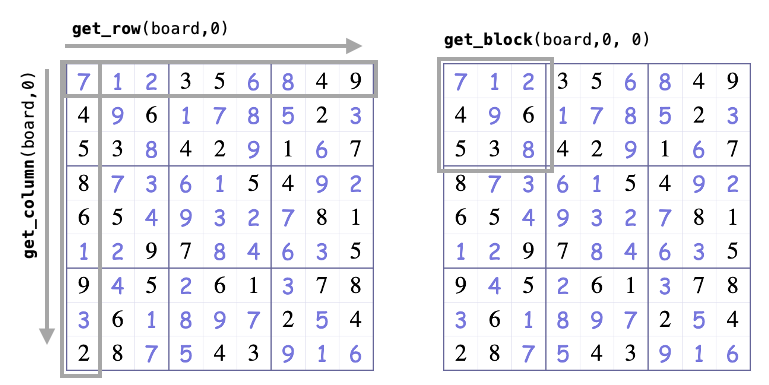
Functional thinking is related to decomposition. It is thinking about the interaction and the artifact you want to create in terms of inputs, outputs, and processes. What parts of the program depend on other parts? What is the flow of the program? Try to break your program down into inputs, outputs, and processes.

It also relates to generalization and abstraction. For every specific thing that needs to be handled, ask yourself: what is this task an example of? Can a complex computational task be broken down into a few powerful functions?

In the example below, the “task” of checking whether one has successfully solved a *Sudoku* (where the objective is to fill a 9x9 grid with digits, so that each row, column, and each of the nine 3x3 sub-grids contains all the digits from 1 to 9) has been broken down into a sequence of small, reusable functions, each handling a specific part of the process, and each taking some input and returning some output.

First, a few functions: one to parse the input string (the 81 numbers), and then a few others to handle querying into the rows, columns, and 3x3-blocks, respectively (shown in the figure below the code):

|  |
| --- |
| **PYTHON**  **def** **parse\_board**(puzzle\_str):  digits = [int(c) for c in puzzle\_str if c.isdigit()]  return [digits[i:i+9] for i in range(0, 81, 9)]    **def** **get\_row**(board, row):  return board[row]    **def** **get\_column**(board, col):  return [board[r][col] for r in range(9)]    **def** **get\_block**(board, block\_row, block\_col):  return [  board[r][c]  **for** r **in** **range**(block\_row \* 3, block\_row \* 3 + 3)  **for** c **in** **range**(block\_col \* 3, block\_col \* 3 + 3)  ] |



And some functions taking care of validation:

|  |
| --- |
| **PYTHON**  **def** **is\_valid\_group**(group):  nums = [n **for** n **in** group if n != 0]  return len(nums) == len(set(nums)) and all(1 <= n <= 9 for n in nums)    **def** **validate\_board**(board):  for i in range(9):  **if** not **is\_valid\_group**(get\_row(board, i)):  return False  **if** not **is\_valid\_group**(get\_column(board, i)):  return False  for br in range(3):  **for** bc **in** range(3):  **if** not **is\_valid\_group**(get\_block(board, br, bc)):  return False  return True |

And finally, tying everything up and trying out our main function:

|  |
| --- |
| **PYTHON**  sudoku\_str = (  "712" "356" "849"  "496" "178" "523"  "538" "429" "167"  "873" "615" "492"  "654" "932" "781"  "129" "784" "635"  "945" "261" "378"  "361" "897" "254"  "287" "543" "916"  )    board = **parse\_board**(sudoku\_str)  **print**(**validate\_board**(board)) # True |

*DATA STRUCTURES*

Data structures are about how you choose to organize the information your program needs. There are many standard ways to do this, and you may already know some from mathematics—like *variables*, *vectors*, or *coordinates*.

In programming, additional structures are useful. One common example is a list: a simple collection of data elements. In some programming languages, all items in a list must be of the same type (e.g., numbers). In others, you can mix types—like text, numbers, or even other lists—allowing for more flexible and expressive ways of organizing data.

**PYTHON**

# A simple list

fruits = ["apple", "banana", "cherry"]

fruits.**append**("strawberry")

**print**(fruits) # ['apple', 'banana', 'cherry', 'strawberry']

# A dictionary

capitals = {"France": "Paris", "Denmark": "Copenhagen"}

capitals["Italy"] = "Rome"

**print**(capitals["Denmark"]) # Copenhagen

# A set

colors = {"red", "green", "blue"}

colors.**add**("blue") # This is already in the set, so it is ignored

colors.**add**("yellow") # This, however, is not, so it is added

print(colors) # {'red', 'green', 'yellow','blue'} - returned unordered

The choice of data structure matters. It affects not only how easy it is for the computer to handle the data, but also how easy it is for you to think about what your program does. A well-chosen structure can make your code simpler, clearer, and more powerful.

However, what's efficient for the computer isn't always intuitive for the human, and vice versa—so choosing the right structure is often a balance between performance and clarity.



**Ten Ideas in Programming** is designed to communicate a minimal set of concepts in programming and computer science that I believe everyone should encounter as part of their education. These ideas range from concrete and low-level to more abstract. They don’t attempt to cover computer science—but they represent a minimum set of ideas that serves two main purposes:

* First, they offer **a practical starting point** for writing code and working with programs.
* Second, they provide 10 concepts that helps read and understand, simple programs. The goal is for the list to be understandable in a few minutes, and for the whole material to be readable within a few hours including examples.

**CONTACT US**

**Department of**

**Science Education**

Københavns Universitet

Niels Bohr Bygningen

Jagtvej 155A

DK-2200 København N.

WHAT IS THIS ABOUT?