Ten Ideas in Programming

a minimal introduction to programming with LLMs

Morten Misfeldt – Morten Aagaard Schultz – Karl Emil Kjær Bistrup

Working Paper July 7. 2025

**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. 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 steppingstone for further studies in computer science or simply to become more literate about the digital world we all live in[[1]](#footnote-1). The order that 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 encounter more experience.

After introducing the ten ideas I have augmented the material with **five key practices** of a programmer, communicating the key things we care for when we start to create 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. Towards the end of the text I have devoted a section to more details about the intentions of the text and about the specific conception of learning that it builds upon.

# Ten Ideas – how we think about programming

The first idea is **data**. Everything digital begins with data: it can be in different forms such as numbers, text, images and sensor readings. Data is usually stored in some sort of structure or sequence (say a list) that support 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**— that are used to store and access data. Variables are important for keeping track of values and enabling dynamic behavior in programs. Variable can be a tricky concept because even though the concept is used across mathematics, statistics and computer science the meaning has some differences 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 transforms 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 to a set of processes that works on data and variables.

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

The sixth idea is **loops or iteration**, the simple yet powerful idea of repetition, which allows programs to perform tasks repeatedly a specified number of times or until certain conditions are met.

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 eight 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.

In chapter X there are examples and further explanation of these ten ideas.

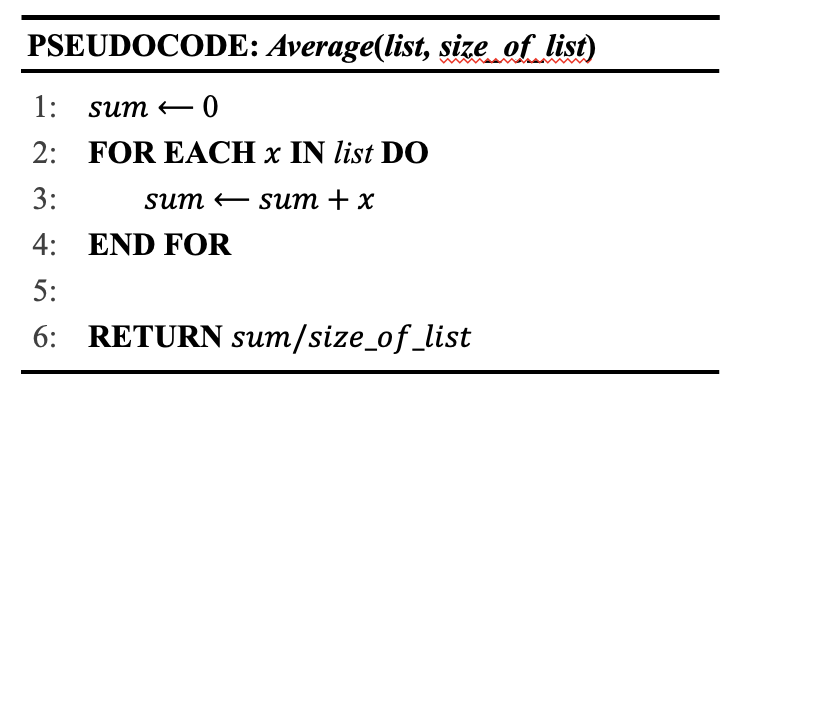
# Five key practices – how we do programming

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.

--

BOX:

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



--

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 into 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.

We **document, modulize 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 how your whole program fits together.

-----

Box: 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.

----

Finaly, 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 to think about **use**. If you’re trying to design a solution, it’s a good idea to understand the problem – and that is often 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 both consider imagined use during development and spend time understanding how the program is used 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.

….

# Learning programming with AI/LLMs

## *Anticipate, Prompt* and *Reflect*

When you’re learning to program and using a large language model (like ChatGPT 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 is the problems broken down

This step helps you clarify your intentions and focus your thinking. Here 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.

## Two activities: coding with AI and concept mapping

I envision that we do two types of learning activities that differ slightly from what is usually found in introductory programming courses. After engaging with a short introduction—such as reading the above overview of the ten key concepts (and possibly five practices) and watching a brief video or talk—the student is invited to complete two tasks.

The first activity involves **using AI, such as ChatGPT, as a tool to write simple programs**. The idea is to experiment with generating and translating code based on the ten key concepts. Students are encouraged to play with how these ideas show up in actual code and to begin forming an intuitive sense of how the concepts work in practice. This activity could optionally be supported by **pseudocode exercises** to help bridge the gap between informal ideas and real code.

BOX task 1: Change the Word Order[[2]](#endnote-1)

Write a sentence and then make a program that changes the order of the words. Try reversing the order or making it random. Think about what the program needs to do, and how you can use variables, or particular data structures, to make it work – it is, in other words, important that we think through how our program handles its data, i.e., how data is organized within our program.

BOX task 2 Falling object[[3]](#endnote-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?

The second activity is more reflective: **students create a mind map** that shows how the ten concepts relate to each other. This could later take the form of a short written reflection or essay. 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.

Together, these two activities—*coding with AI* and *concept mapping*—are meant to support both the practical and conceptual dimensions of learning to program.

# Further explanation and examples

## Data

**Data** is the raw material of digital technology— data is the way in which information that 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

**PYTHON**

x\_position = -47

y\_position = 4

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

## Function

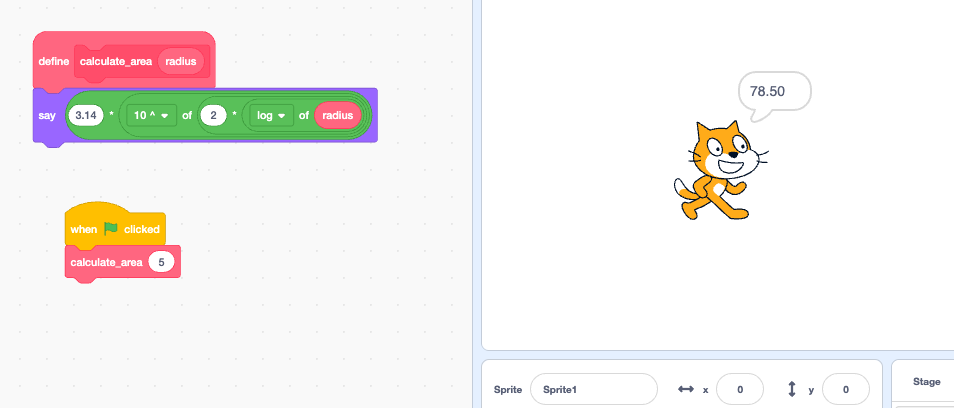
In programming, a function is something you call to make something happen. Different languages implement functions in different ways, but they typically involve three key ideas.

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.



**PYTHON**

radius = 5

area = 3.14 \* radius \*\* 2

**print**(area) *# 78.5*

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, and that is what makes it possible to create programs that can handle all kinds of different 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 more like a named box that can store a value — and that value 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.

**PYTHON**

mortens\_variabel = 34

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

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

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

y = (mortens\_variabel + 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*.

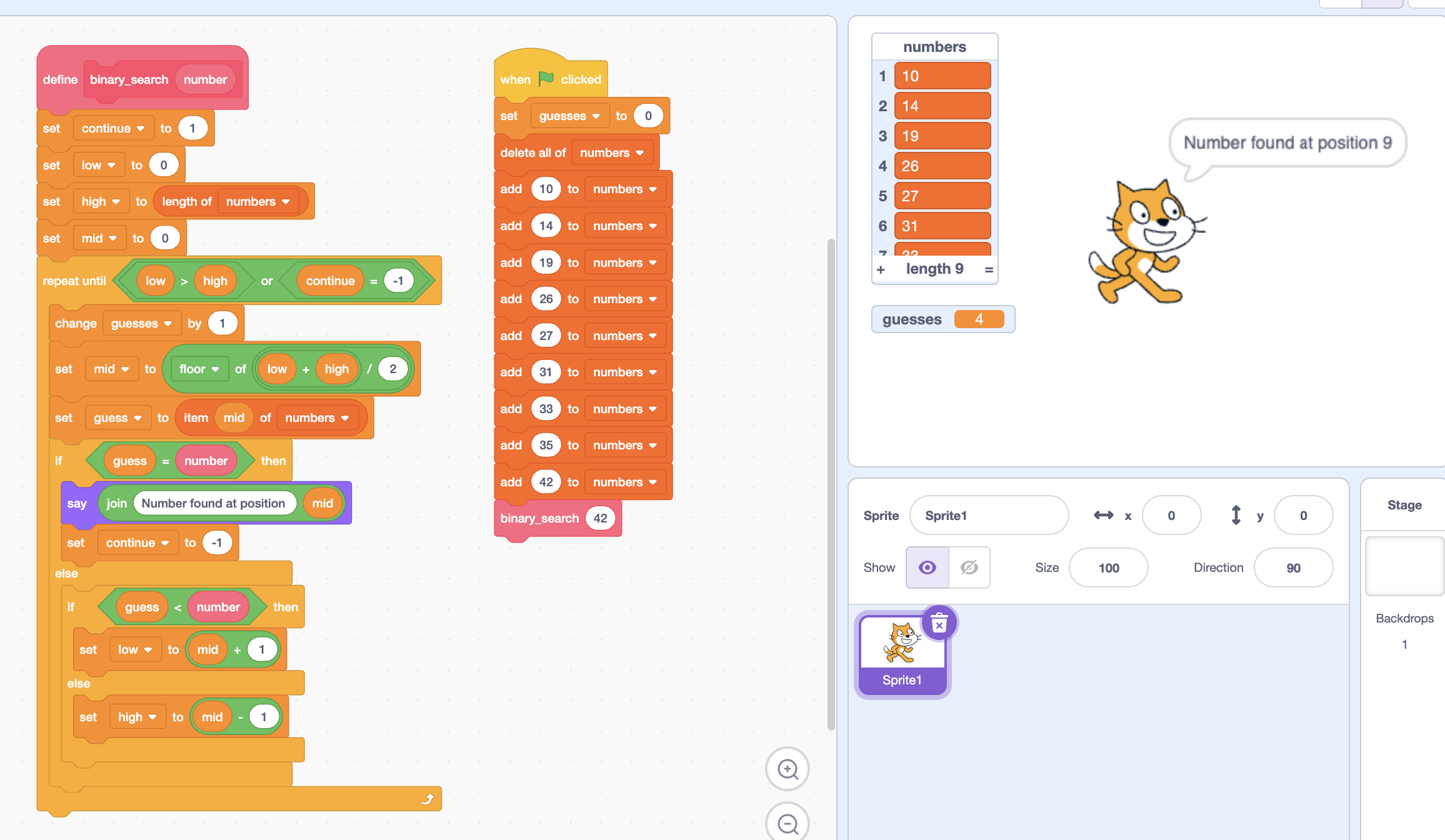
In some programming languages, you can set up *parallel* activities (things that run at the same time), but even these are built from smaller *sequential* instructions.

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.



**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

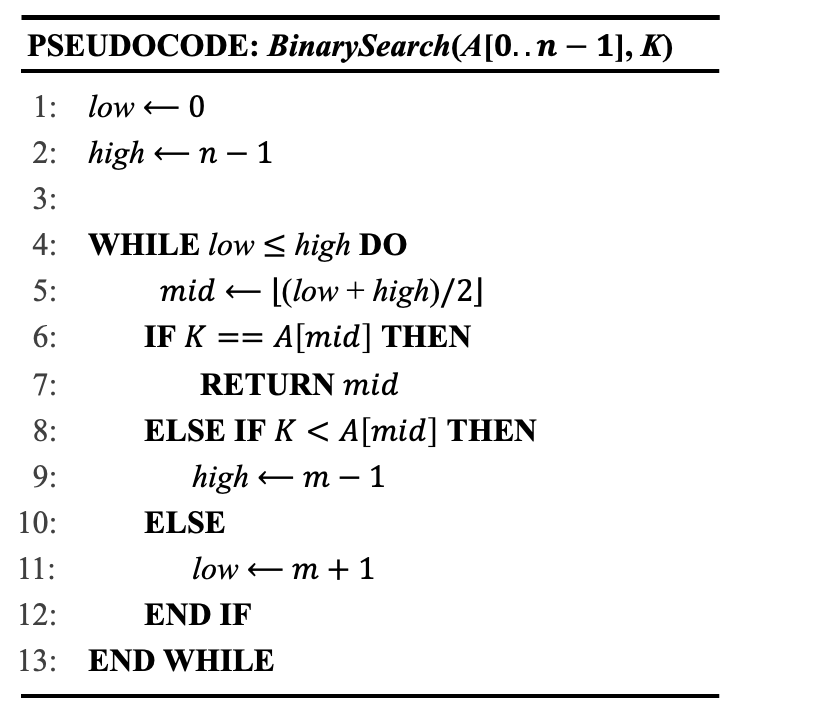
else:

high = mid - 1

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

**print**(**binary\_search**(list\_of\_numbers, 26))

The two examples above are to be understood as concrete implementations of the following pseudocode, a human-readable, informal description of program code or algorithms, i.e., without regard for the particular syntax of some specific programming language 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.

**PYTHON**

if temperature > 25:

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

else:

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

**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.

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

AI-genereret indhold kan være ukorrekt.

**PYTHON**

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

**print**(i)

number = 1

while number < 10:

**print**(number)

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 make a model in computer science, we choose the most important parts of a situation and describe them clearly, so the computer can use them to run a simulation, respond to input, or show certain 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

JEG HAR SVÆRT VED AT FORMULERE DENNE OG DEN NÆSTE

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, 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} scorede over gennemsnittet med sine {score} point.")

*# Knytter an til 3. praksis ("We organize our thinking ...")*

scores = **get\_scores**() *# abstraherer datakilden (DATA)*

avg = **calculate\_average**(scores) *# abstraherer matematikken/beregningerne (COMPUTATION)*

**print\_above\_average**(scores, avg) *# fokus på fortolkningen/outputtet (INTERACTION)*

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

functions as a way to do abstraction

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.

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.

# About the project

This part is more like meta text – not to be included in the final resource for the students – but to clarify and keep track of what I try to do, and also to invite people to further think with me…

.

This mini-project is about developing a relational and concept-focused way of introducing programming. The goal is to support intuitive understanding and concept formation by starting with a small set of key concepts that are central to almost any programming project. Instead of building up from low-level concepts, we aim to find cognitively “natural” starting points—concepts that are already present in most programming tasks. These are also concepts that are important in most curricular resources/textbooks. But here we *start with all ten.* I think of this as conceptually “starting from the middle”.

Th´ limited set of concepts we use can be grounded in a broader framework of ideas and relationships in programming (see the work of Benjamin Allsopp, 2016). The approach is inspired by research in mathematics education, especially early work on concept formation (from the 1980ties). This work has shown extremely relevant when conceptual development is supported by interactive tools—and I believe similar ideas can help us understand how people learn to program in a situation where cognitive tools can handle a lot of the syntax hassle.

Three ideas are especially important:

1. **Concept image vs. concept definition**  
   Drawing on the work of Shlomo Vinner and David Tall (1981), I focus on the difference between the formal definition of a concept and the mental image people have of it—their "concept image." A concept image includes examples, personal associations, and intuitive connections to other ideas. I believe programming education should actively support the development of these images, not just teach formal definitions.
2. **Relational understanding**  
   Inspired by ideas from mathematics education (e.g., Skemp 1978), I emphasize relational understanding—how one concept connects to others—rather than just procedural or instrumental use. Learning to program involves building a web of concepts, and helping learners see these connections is critical to developing real understanding.
3. **Processes and objects**  
   I also draw on theoretical work about the dual nature of mathematical objects—as both processes and things. This is well described in Anna Sfard’s work (1991) and in the APOS theory by Ed Dubinsky. In programming, functions and structures can be seen both as something you *do* (a process) and something you *use* (an object). Understanding how these two views interact is key to developing deeper conceptual insight.
4. **Computational thinking.** CT is… explain… Some people describe computational thinking as the ability to explain what you want in a way that a computer can understand. That’s not wrong—but it's also important to remember that programming languages always include some level of interpretation. They aren’t just mechanical instructions; they also involve communication—like explaining something to another person.

In fact, programming today often feels more like explaining your ideas to a peer than to a machine. This is especially true with the increasing help we can get from artificial intelligence when writing code.

1. **Maybe add algebraic thinking**

Kaput, Archavi

1. **Maybe add key ideas about enquiry, schemes and anticipation**
   1. **Dewey and Piaget**

In short, this project aims to build a foundation for programming that is intuitive, relational, and conceptually rich—rather than focused on syntax or formal logic.

# References

Allsopp, B. B., & Ejsing-Duun, S. (2016). Programming concepts in playful programming products. In 10th European Conference on Games Based Learning: ECGBL 2016 (p. 1). Academic Conferences and publishing limited.

Dubinsky, E., & McDonald, M. A. (2001). APOS: A constructivist theory of learning in undergraduate mathematics education research. In The teaching and learning of mathematics at university level: An ICMI study (pp. 275-282). Dordrecht: Springer Netherlands.

Sfard, A. (1991). On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. Educational studies in mathematics, 22(1), 1-36.

Skemp, R (1978). Relational understanding and instrumental understanding. The Arithmetic Teacher, 20-26.

Tall, D., & Vinner, S. (1981). Concept image and concept definition in mathematics with particular reference to limits and continuity. Educational studies in mathematics, 12(2), 151-169

.

1. This text is built on an attempt to outline a specific pathway into learning about programming that I currently call “conceptual, intuitive and from the middle”. It is a response to two trends: First, programming and basic computer science are no longer specialized skills, but something everyone should have some familiarity with. Second, the emergence of digital tools — especially those based on generative artificial intelligence — has made it much easier to get started creating programs.

   My aim is to develop an introduction to programming that emphasizes key concepts rather than syntactic details. I focus on what I consider to be the most important ideas in programming and the complex relations between these ideas, rather than on a hierarchical construction of knowledge from fundamental to more abstract and/or applied computational objects. My hope is to support meaning and intuition in introductory programming before getting into syntax and formal logic. [↑](#footnote-ref-1)
2. **Task 1: Playing with Word Order task in more detail.**

   **Objective:**  
   Explore how programming can help us manipulate text by changing the order of words in a sentence.

   **Instructions:**

   **Start with a Sentence:**  
   Think of a sentence — it can be anything you like. For example:  
   *"The quick brown fox jumps over the lazy dog."*  
   Write it down so you can refer to it.

   **Break It Down:**  
   Now think about how a program could work with that sentence. What would it need to do to change the order of the words?

   How can we **store the sentence** in a program?

   How do we **separate the words** in the sentence?

   What kind of **data structure** would be useful for this? (e.g., lists or arrays?)

   **Explore Word Order:**  
   Try to create a program that does one of the following:

   **Reverses** the order of the words.

   **Randomizes** the order of the words.

   (Bonus: Try both!)

   **Think and Plan First:**  
   Before you start coding, sketch out how the program will work. What steps are needed?  
   What tools (like variables, lists, or functions) do you think you’ll need?

   **Reflection Questions:**

   What does your program need to know about the sentence before it can change the word order?

   What was easy or hard about changing the order?

   What kind of word manipulations might be useful in other situations? [↑](#endnote-ref-1)
3. **Task 2: Simulate a Falling Object**

   **Objective:**  
   Create a simple simulation of an object falling straight down under the influence of gravity.

   **Instructions:**

   **Understand the Motion:**  
   Gravity causes a constant acceleration (e.g., 9.8 m/s²). Over time, the speed of the falling object increases, and it falls further and further.

   **Think About the Data:**  
   What information do we need to keep track of?

   Time (how long the object has been falling)

   Velocity (how fast it’s falling)

   Position (how far it has fallen)

   **Plan the Update Rule:**  
   For each step in time:

   Increase velocity using acceleration

   Increase position using velocity

   **Try It Out in Code:**  
   Write a loop that updates these values for each time step (e.g., every second or every 0.1 second). Print out or visualize the results.

   **Optional Extras:**

   Add a maximum time or a "ground" position where the object stops

   Try different accelerations (e.g., Moon gravity)

   Add air resistance for more realism

   **Reflection Questions:**

   How does the position change over time?

   What happens if you change the size of the time step? [↑](#endnote-ref-2)