COMP 1510
Winter 2022

Programming Methods

Week 02: Programming 101

Agenda for week 02

1. Compound operators
2. Basic input
3. Data conversion and str(), int(), float(), etc.
4. type()
5. Data types and why we need them: int, float, bool, and the rest (starting with str and None)
6. () [] {} and <>
7. Text editors
8. IDEs and PyCharm
9. Introduction to version control using git and GitHub
10. Four cornerstones of problem solving using computational thinking:
    1. Decomposition
    2. Abstraction
    3. Pattern matching
    4. Algorithms
11. Structured code using control statements for sequence, selection, repetition, and indirection
12. Flowcharts

COMPOUND OPERATORS

Assignment revisited

- We can use a variable name on both sides of an assignment statement
- The original value is overwritten
- (Actually, we overwrite the address of the original value with the address of the new value, but you already know that!)
- Sometimes we increment the value like this:

```
current_value = current_value + something_else
stored_value = stored_value / 3
```

We do this a lot in programming

- Programmers are notoriously lazy
- We love to reuse code, and we love "typing shortcuts"
- **Using a variable name on both sides of an assignment statement happens a LOT**
- Most programming languages have shortcuts for operations like this
- Python uses compound operators

Compound assignment operators

```
age_years = age_years + 1
age_years += 1

account_balance = account_balance – 50.00
account_balance -= 50.00

num_calls = num_calls * 2
num_calls *= 2

remainder = remainder / 2
remainder /= 2

offset = offset % base
offset %= base
```

BASIC INPUT

Input

- The *input( ) function* causes the interpreter to pause and wait until the user has entered something
- Input can be anything the user types including numbers, letters, and special characters
- Be careful though: *all input is treated as a string, even if the input is numerical*
- We can pass the input( ) function a string argument as a user prompt:

```
user_age = int(input())
user_age = int(input('Enter your age: '))
```

Whitespace and input

- Unless specified otherwise, *white space* is used to separate the elements of the input
- White space includes:
  1. space characters
  2. tabs
  3. new line characters
- Programmers call the collections of characters between white space words or, more generally, **tokens**.

DATA CONVERSION

How can we convert from a string to…

- Very easily!
- We call this **type conversion** or **data conversion**
- There are some conversion functions in Python that were written for this:

Function	Notes	Can convert:
int( )	Creates integers	int, float, strings that only contain integers
float( )	Creates floats	int, float, strings that contain integers or fractions
str( )	Creates strings	Anything! (this is helpful when printing)

Aside: a word about data conversion

- **Implicit conversion (coercion)** occurs during runtime
- This happens automatically when operands are converted so they can be used together
- For example, when adding an int and a float, the int will be converted to a float, added to the existing float, and the result will be a float
- A float would never be converted to an int in this case because we would have to **truncate** it (and lose data)

Explicit data conversion

We can use the built-in functions to explicitly convert types in Python, i.e., int( ), float( ), and str( ):

```
a_int = 5
b_int = 10
c_float_sum = float(a_int + b_int)

a_float = 1.5
b_float = 3.2
c_int_sum = int(a_float + b_float)
```

DATA TYPES, print( ), and type( )

So what kind(s) of data can we work with?

- In Python, we combine operators and operands to form expressions
- Those operands (numbers, Boolean values, strings of characters) are the core things that Python programmers manipulate
- We represent them in memory as discrete values inside objects
- Every object stores a value that has a type
- The type defines the kind of things we can do with it
- For example, we can add and subtract integers
- We cannot find the square root of a string. That doesn't make sense.
- We need rules about what kind of 'thing' each operand is!

We categorize our data into types

1. Integers aka whole numbers (-4 or 17 or 2018)
2. Floating point numbers have a decimal point (1.0000 or 3.14 or -19.73)
3. The Boolean values True and False
4. Strings like “a” and “ hello “ and “What is *5 ** 5*?”
5. Named collections of statements called functions
6. Collections like dictionary, list, set, tuple
7. A special type called None which means the absence of data.

What else can you tell me about this Chris?

- Every value we store in memory is encapsulated inside a special Python data structure we call an object
- Every object in Python has an address while it is in memory
- When we assign a value to a variable, we create an object in memory that contains the value and assigning its address to the variable!
- Each object has:
  1. *Value* like 'Python', '22', -0.34, True, etc.
  2. *Type* such as integer or float that determines the object's supported behavior, i.e., what we can do with it
  3. *Identity* aka a unique identifier that tells us where it resides in memory.

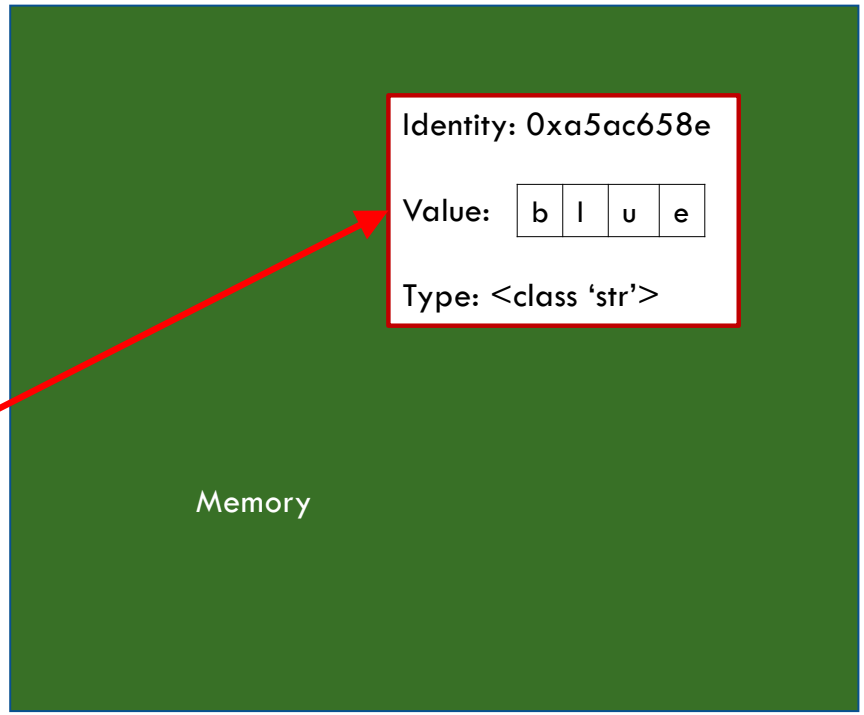
A simple model of an object in memory

When I type this:

```
colour_of_eyes = "blue"
```

This is what actually gets assigned:

```
colour _of_eyes =
                  0xa5ac658e
```



How can we determine a datum's type( )

- Suppose we store the address of an object in a variable
- How can we determine what kind of object is at that address?
- We can use a built-in function called type( )
- We can pass the variable as an *operand* to a *function*
- We say we are passing an *argument* to a *function*
- We say the type( ) function accepts a *parameter* which we put between the parentheses
- If we have a variable called radius, we can use the type( ) function like this: `type(radius)`

Aside: the built-in functions

- A named series of statements
- We use a function by invoking its name
- We say we are *invoking* the function
- This causes the function's statements to execute
- We've met two of Python's built in functions:
  - print( )
  - type( )
- Both functions accept a single parameter aka something to operate on
- We say we are *passing an argument* to the function

The built-in functions

- The Python interpreter provides some functions that are always available
- These are sequences of statements that are so useful, they have been encapsulated into a single group called a function
- These functions are described here: https://docs.python.org/3/library/functions.html#built-in-functions
- We have already started learning how to use them:
  - print( ) accepts some stuff and converts it to a string and prints it
  - type( ) accepts one thing and tells us what kind of thing it is
  - input( ) accepts user input from the keyboard and gives us a string with it!

( ) { } [ ] < >

Vocabulary lesson!

We don't actually call these ( ) brackets!

Here are what you should call these peculiar little things:

( ) these are parentheses

{ } these are curly braces

[ ] these are square brackets

< > these are angle brackets

This is the (not-so) perfect segue…

- The command line lets us travel through the file system of our computer
- The command python3 is what we use to execute the Python shell
- We can only type one instruction at a time
- This gets tedious very fast
- We want to store a collection of instructions in a file
- We have software to help us do this
- We use a **text editor** or an **IDE (integrated development environment)**

TEXT EDITORS

Text editors and executing the code

- *Don't use MS Word.*
- We need a plain text editor like VIM, Atom, Notepad++…*
- I suggest anything free. I use Sublime Text.
- Great for quick code snippets, class demos (!), working on little programs or proofs of concept:
  1. We type our instructions in the file
  2. Save the file with a .py extension (means python)
  3. Execute on the command line by navigating to the folder that contains our .py file, and typing python3 program.py

* Not Notepad on Win or TextEdit on macOS. Yuck.

But even Sublime isn't enough!

- We need more than colourful highlighting
- IDEs combine text editors, compilers, code completion tools, testing and debugging tools, integration with version control software, and so much more!
- We are going to learn how to use an industry standard IDE this term
- We will use PyCharm

PYCHARM

Install PyCharm

- ***It is important that you do this AFTER you install Python 3***
- Apply for a free Student License at JetBrains, the producer of PyCharm, at http://www.jetbrains.com/student/ and ***install PyCharm Professional (not Community)***
- We will use PyCharm to develop programs and systems that are written in Python 3 and compiled to Python bytecode
- The Python bytecode files are not human-readable; they will be read and executed quickly by the Python interpreter
- We say that Python is an interpreted language because of this!
- PyCharm takes care of all of this for us, plus more!

Hello world

- Traditionally the first program we always write in a new language is called hello world
- We are creating a new ‘thing’ that exists, a new kind of knowledge and power is at your fingertips
- This new ‘thing’ must greet the world
- “Hello world!”
- (Early programmers read a lot of SF. We still do.)

INTRODUCTION TO VERSION CONTROL

Version control, git, and GitHub

- Programmers use version control software like git and cloud-based repositories like GitHub
- Each time we add some code to our project, we document the change and save the difference (delta) to our version control system
- We can create branches that contain different code ideas and merge branches together
- We can revert changes, bring them back, label builds…
- We will use git and GitHub with PyCharm to create a toolchain and a workflow that maximizes our efficiency

Install git and open a GitHub account

- Install git from here https://git-scm.com/downloads
- Apply for a free Student Developer Pack from GitHub at http://education.github.com/pack
- After your GitHub account has been approved, create an avatar for your account
- Come to every future class:
  1. PyCharm open
  2. GitHub webpage open
  3. Ready to code.

What is the essence of version control?

- Maintain a list of changes we make to our files
- Different people can work on the same file and merge their changes
- Most version control systems let us:
    1. Create a repository of files
    2. Add files and folders of files to the repository
    3. Name and commit (save) each collection of additions we make and changes we make to existing files
    4. Review changes
    5. Revert changes

Exactly what is git?

- A popular (some say trendy) flavour of version control
- Software that must be installed locally (on your laptop)
- When we designate that a project directory (folder) is a git repository, the git software creates a hidden subfolder inside it called .git
- The .git subfolder contains git-specific files that track the changes we make to the files in the project folder
- We add files to a git repository, and git records the changes for us in the .git folder
- We commit changes made to files, and git records the changes
- The files in the folder will always be the most recent version.

What is GitHub?

- A website
- A web-based hosting service, aka a repository for remote git repositories
- Distributed version control: the complete codebase is stored in the cloud and mirrored on each developer's local rig
- We must do this for each new project we create:
  1. Clone our original, local repository to GitHub on the cloud
  2. Continue to commit changes to files in the local repository to our .git folder
  3. Occasionally push all committed changes from the local repository to the remote repository, which can be shared with others.

Demonstration*

* I'm going to do this many times, so if it confuses you the first time or even the first several times, that's okay! Don't worry, I will do it again. And again. And if you still don't understand, please tell me! It's important to me that you understand this.

Watch me:

1. Create a new PyCharm project
2. Import the new PyCharm project into version control by designating it a git repository
3. "Add" files to version control so git starts taking snapshots
4. Clone the git-monitored PyCharm project to the cloud (GitHub)
5. Write code and commit changes to local git repository
6. Then push changes to the git repository on GitHub in the cloud.

COMPUTATIONAL THINKING

Computational Thinking*

"the thought processes involved in formulating problems and their solutions so that the solutions are in a form that can be effectively carried out by an information-processing agent."

[Cuny, Snyder, Wing 2010]

*aka CT

Dr. Jeannette M. Wing (Columbia U)

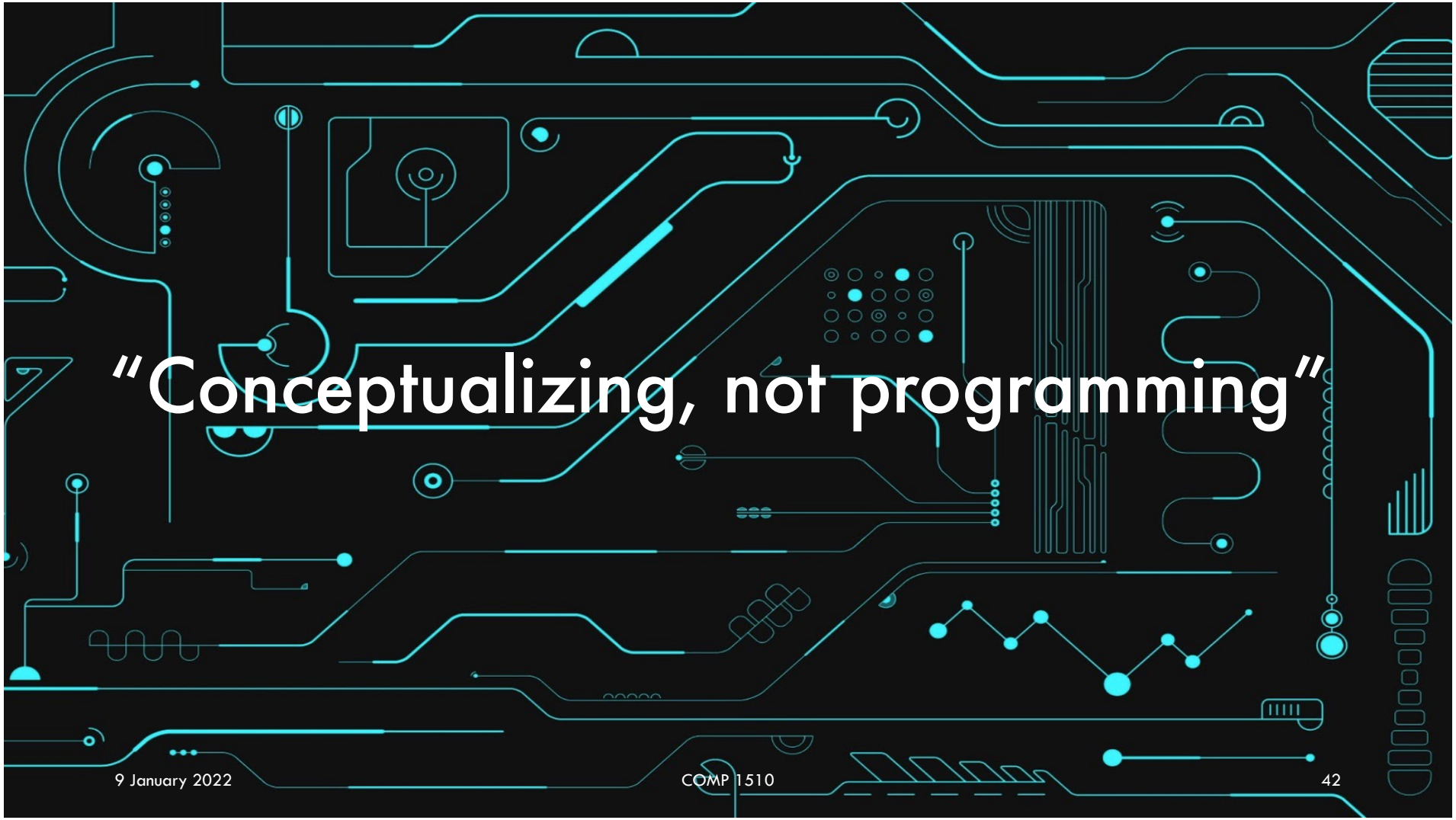
Viewpoint: “Computational Thinking”

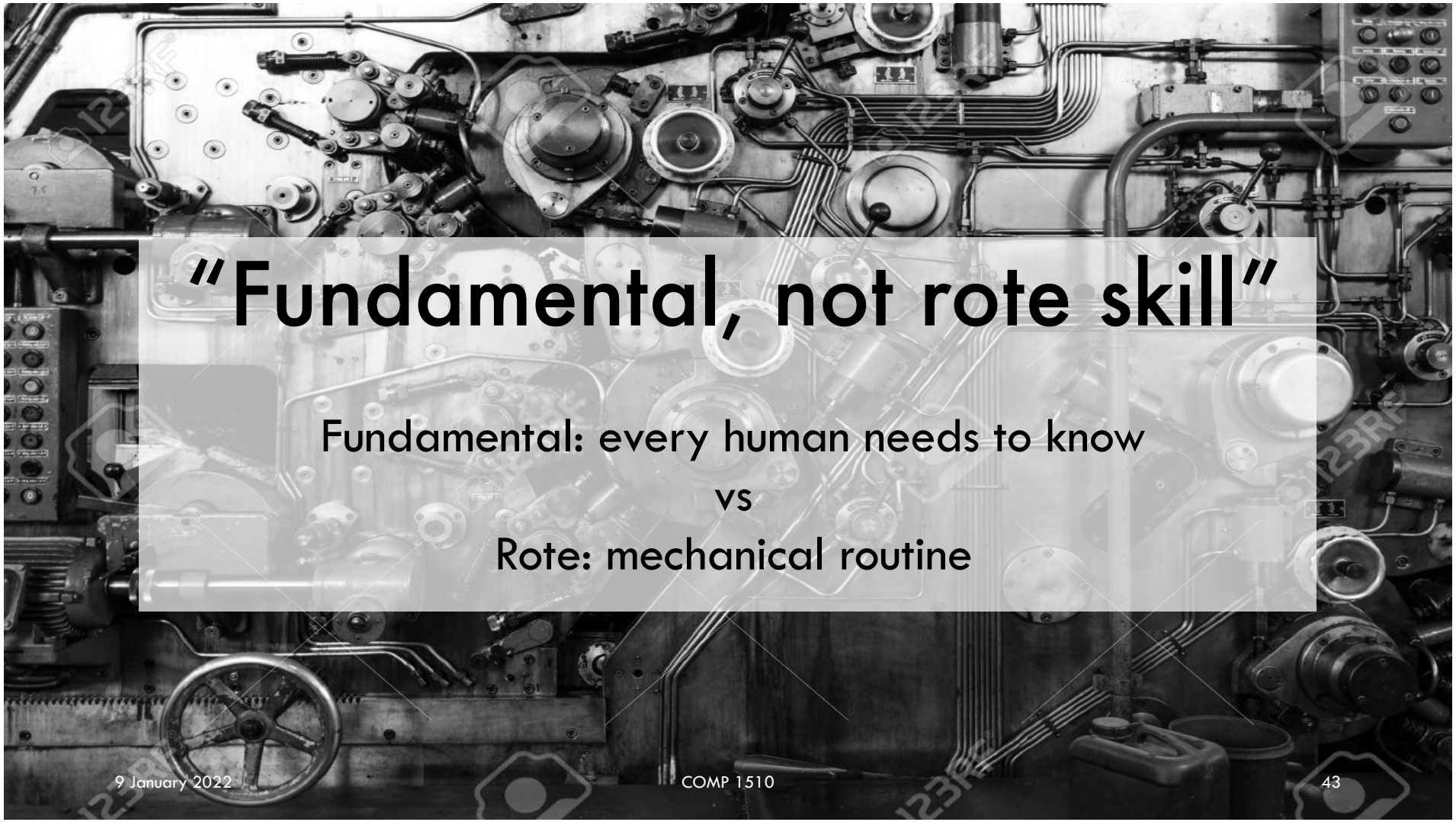
COMMUNICATIONS OF THE ACM
March 2006/Vol. 49, No. 3

Informally:

"[CT] describes the mental activity in **formulating a problem to admit a computational solution**. The solution can be carried out by a human or a machine or, more generally, by combinations of humans and machines."

[Wing 2010]


"Conceptualizing, not programming"

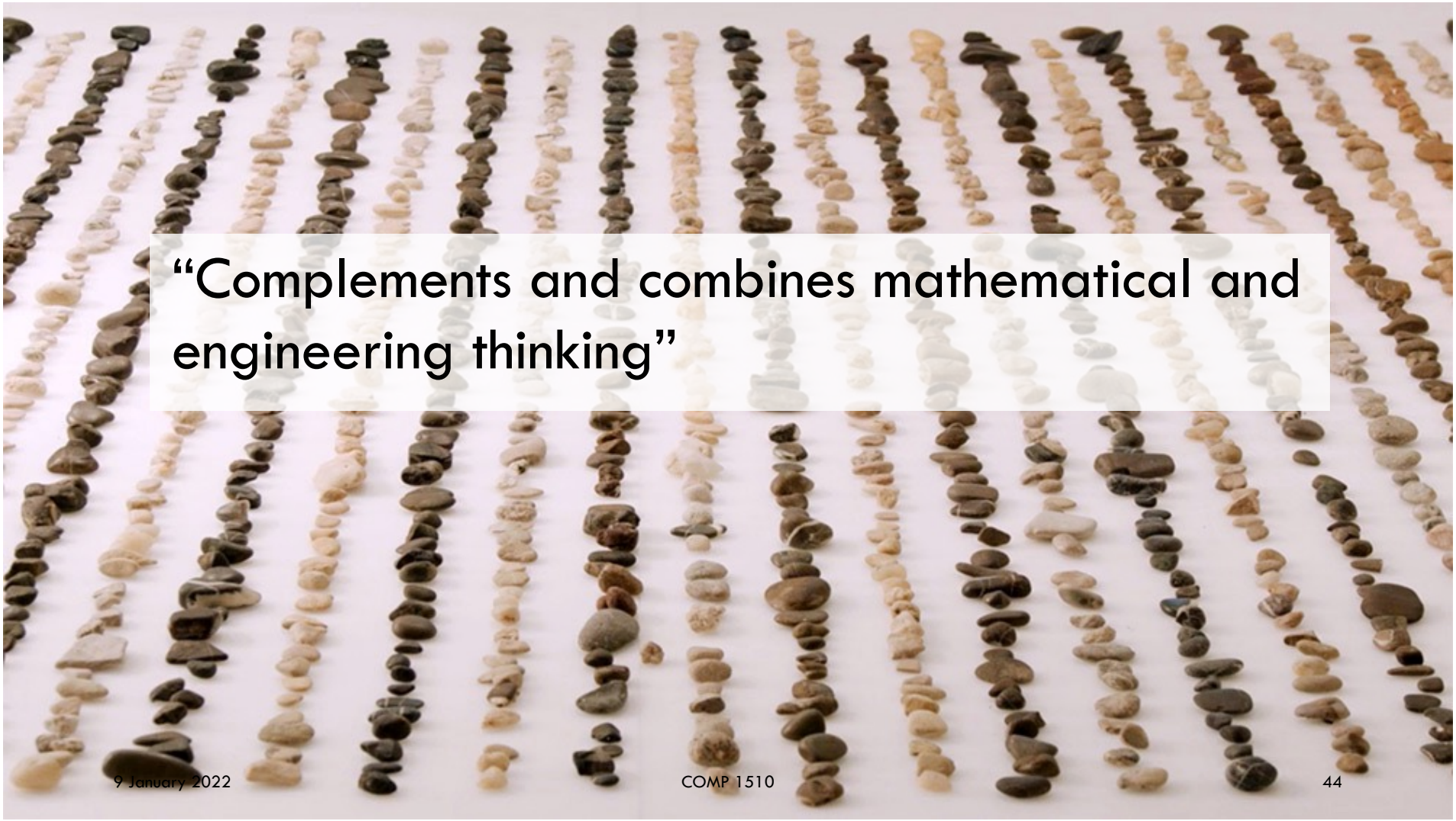


"Fundamental, not rote skill"

Fundamental: every human needs to know

vs

Rote: mechanical routine


"Complements and combines mathematical and engineering thinking"


"For everyone, everywhere"

Decomposition

Pattern Matching and Data Representation

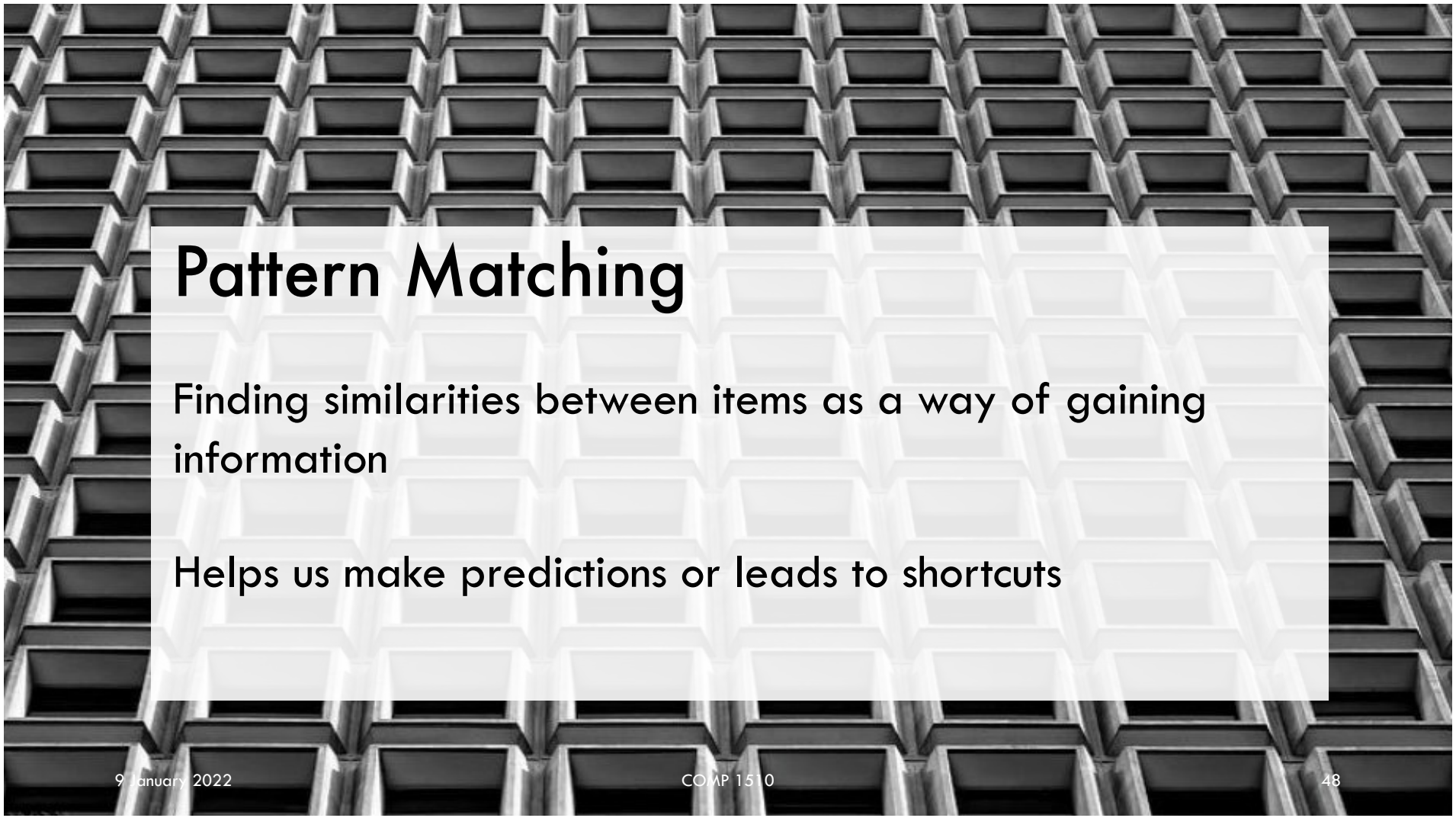
Abstraction and Generalization

Algorithms and Automation

Decomposition

Breaking a problem down into smaller, more manageable pieces, like:

- Remembering phone numbers in segments
- Reading words syllable by syllable
- Performing arithmetic operations



Pattern Matching

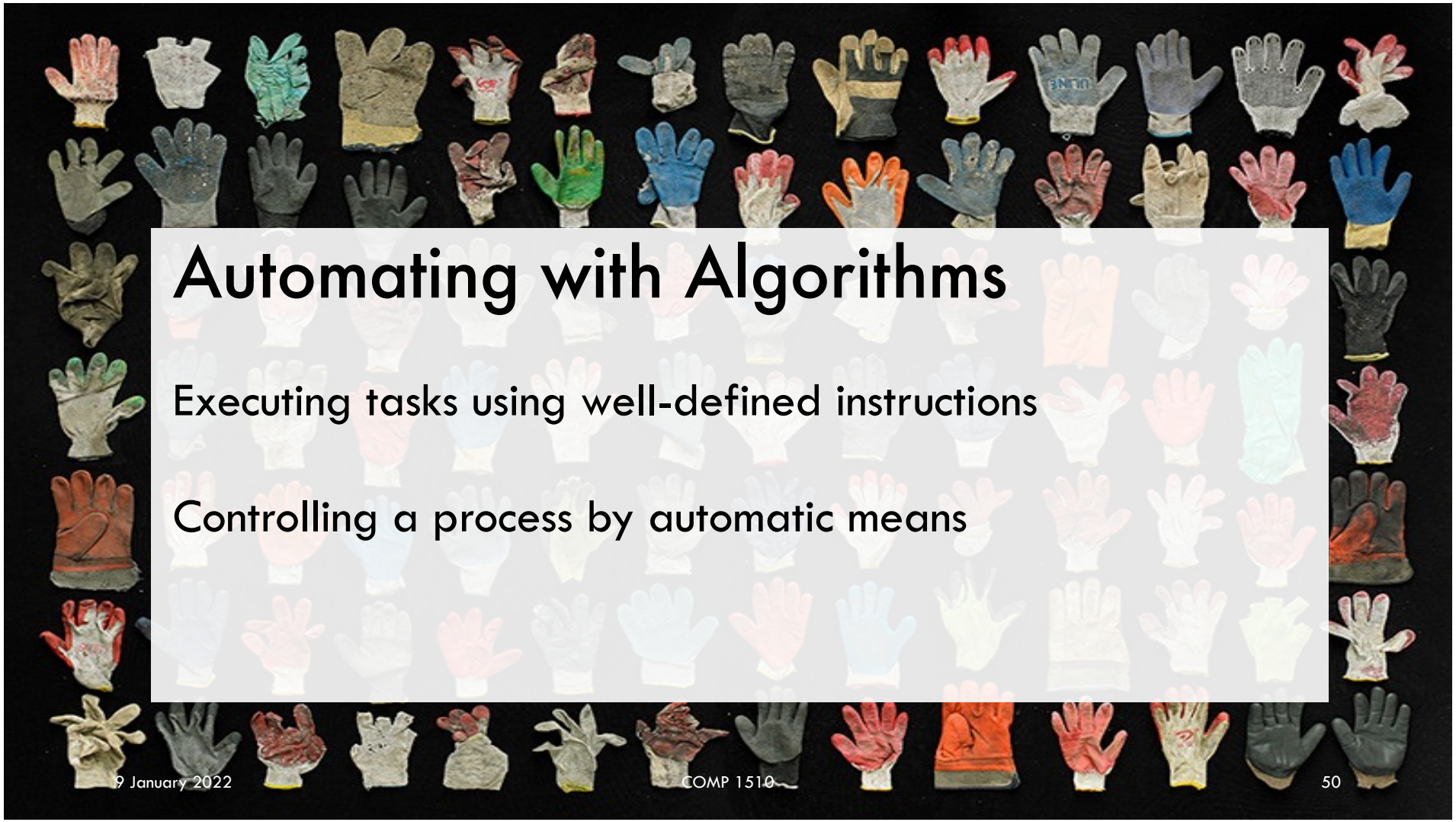
Finding similarities between items as a way of gaining information

Helps us make predictions or leads to shortcuts

Abstraction

Ignoring certain details in order to find a solution that works for a more general problem

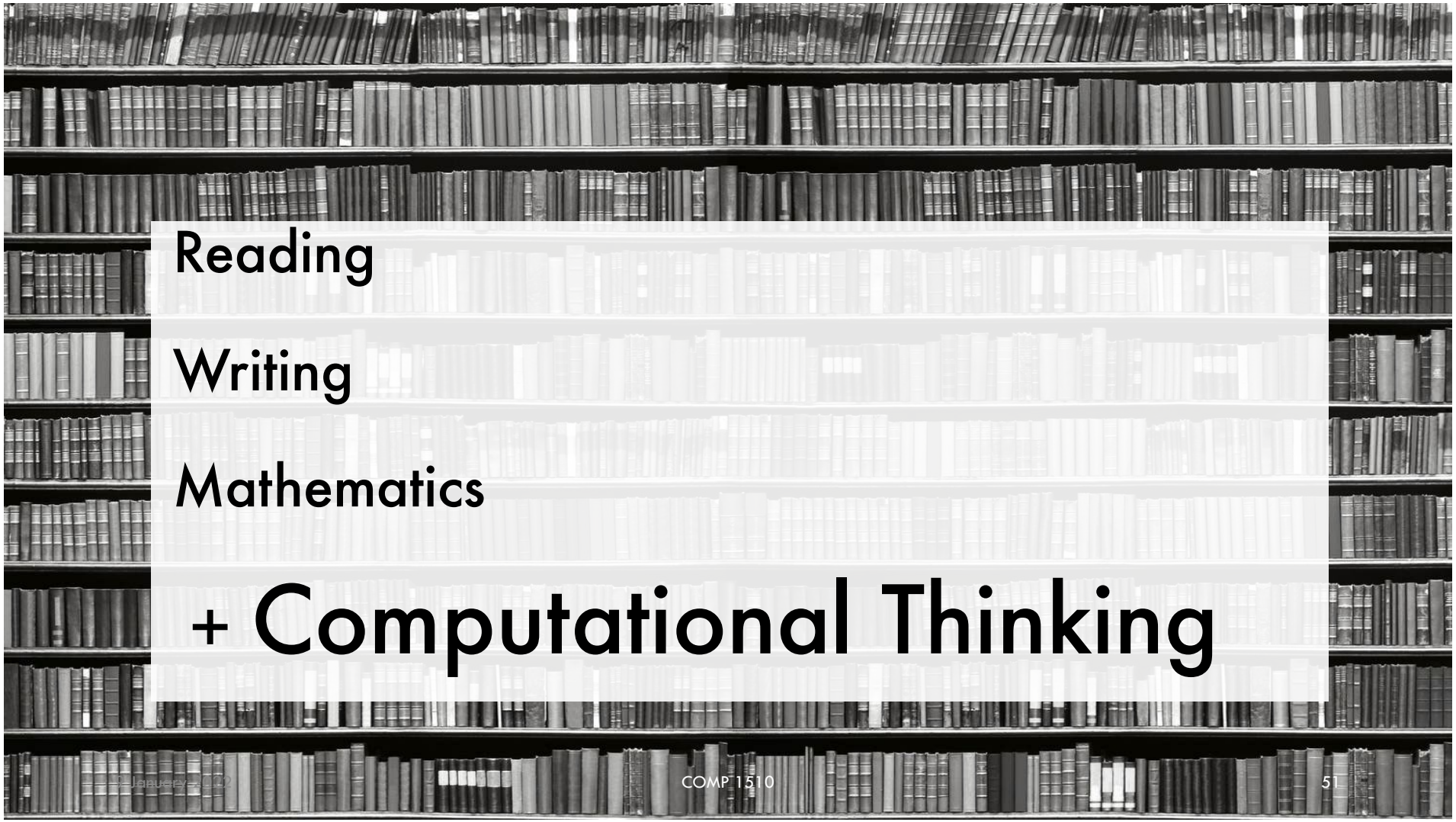
Ubiquitous (everywhere!) in CS (and in life!)

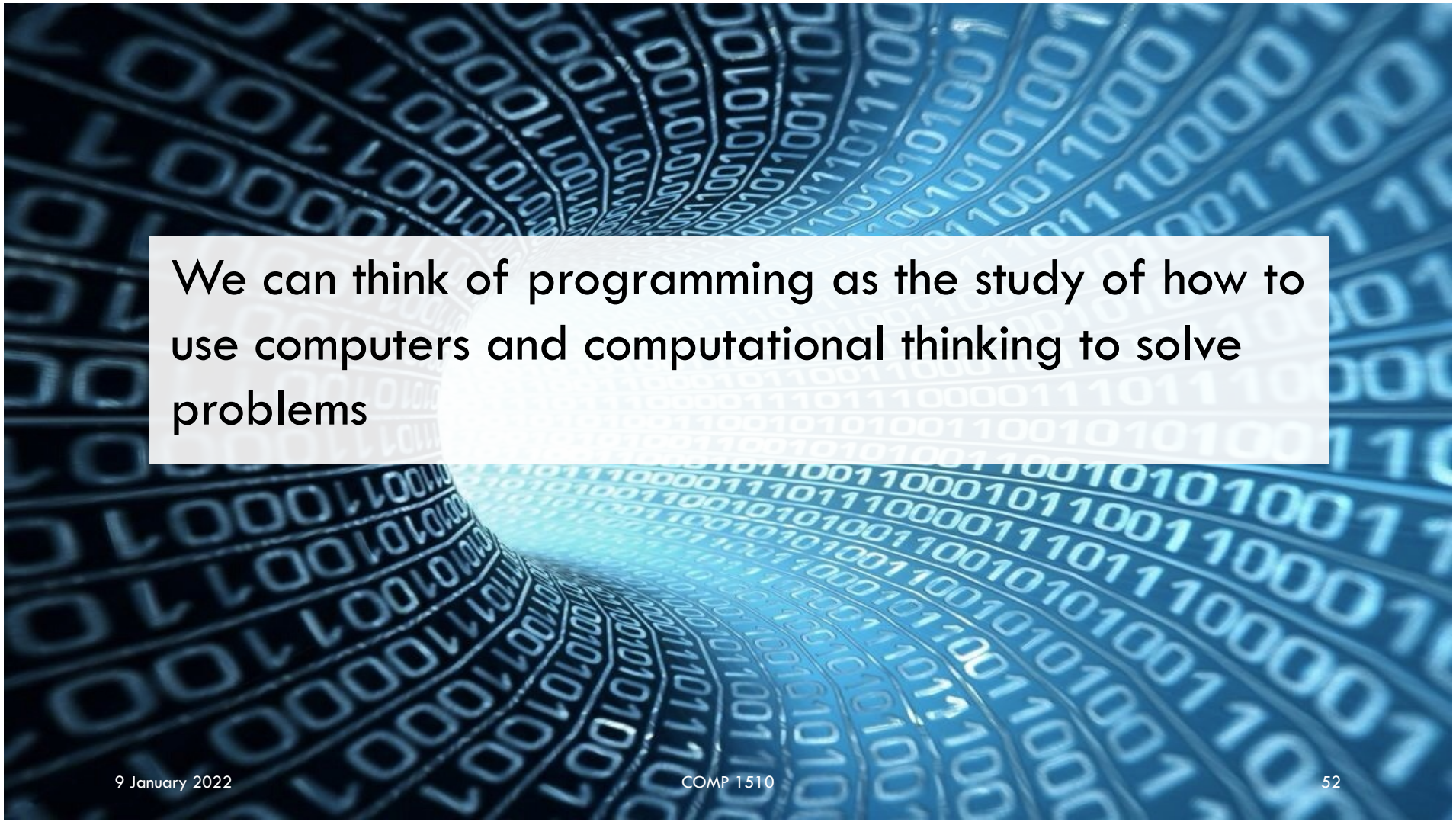


Automating with Algorithms

Executing tasks using well-defined instructions

Controlling a process by automatic means


Reading
Writing
Mathematics
+ Computational Thinking

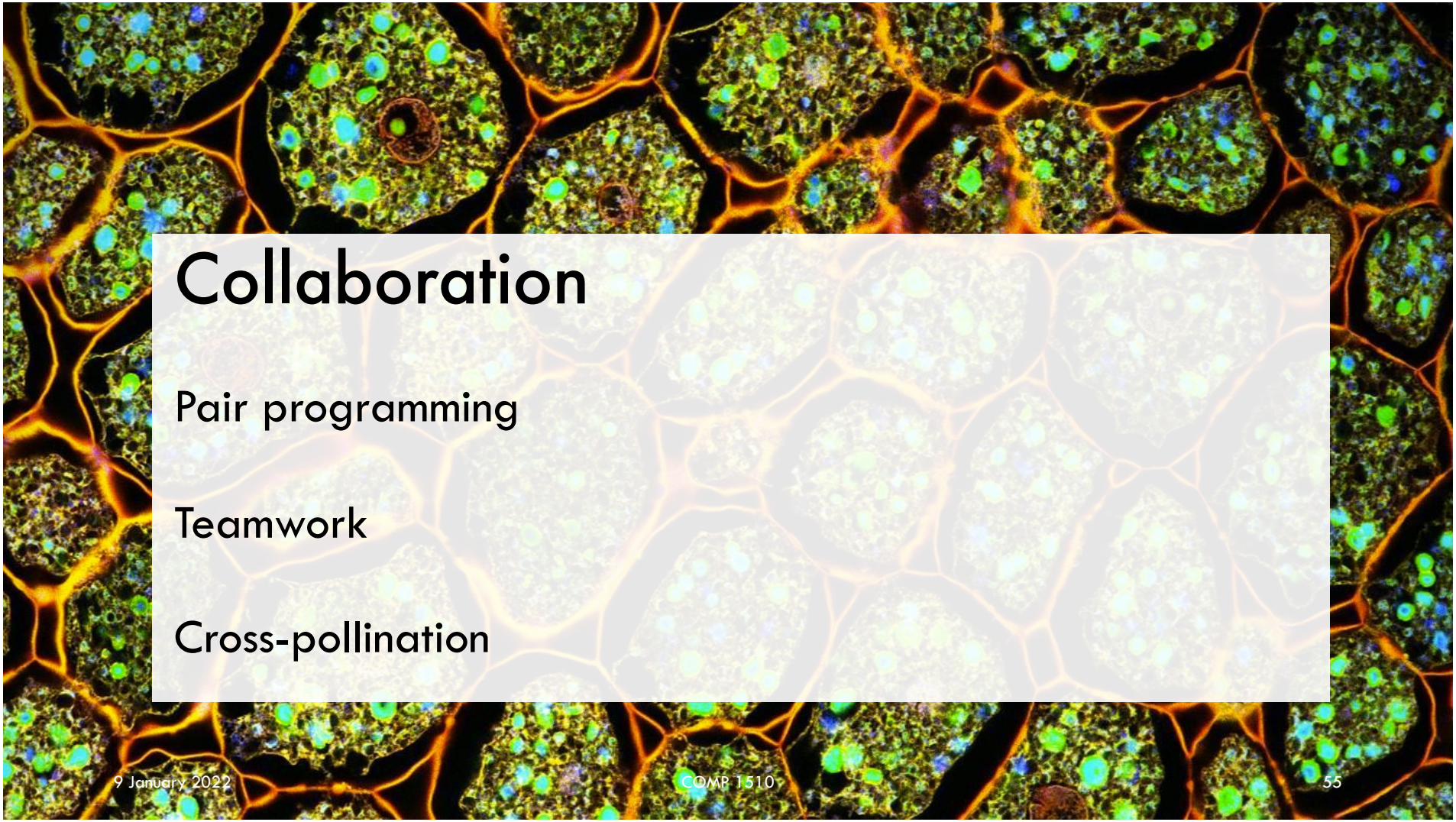

We can think of programming as the study of how to use computers and computational thinking to solve problems

Creativity	Collaboration	Communication
Persistence	Problem Posing	Problem Solving

Creativity

CT is an incubator for creativity

When we learn how to think creatively in a restricted environment, barriers become provocative challenges rather than boundaries.

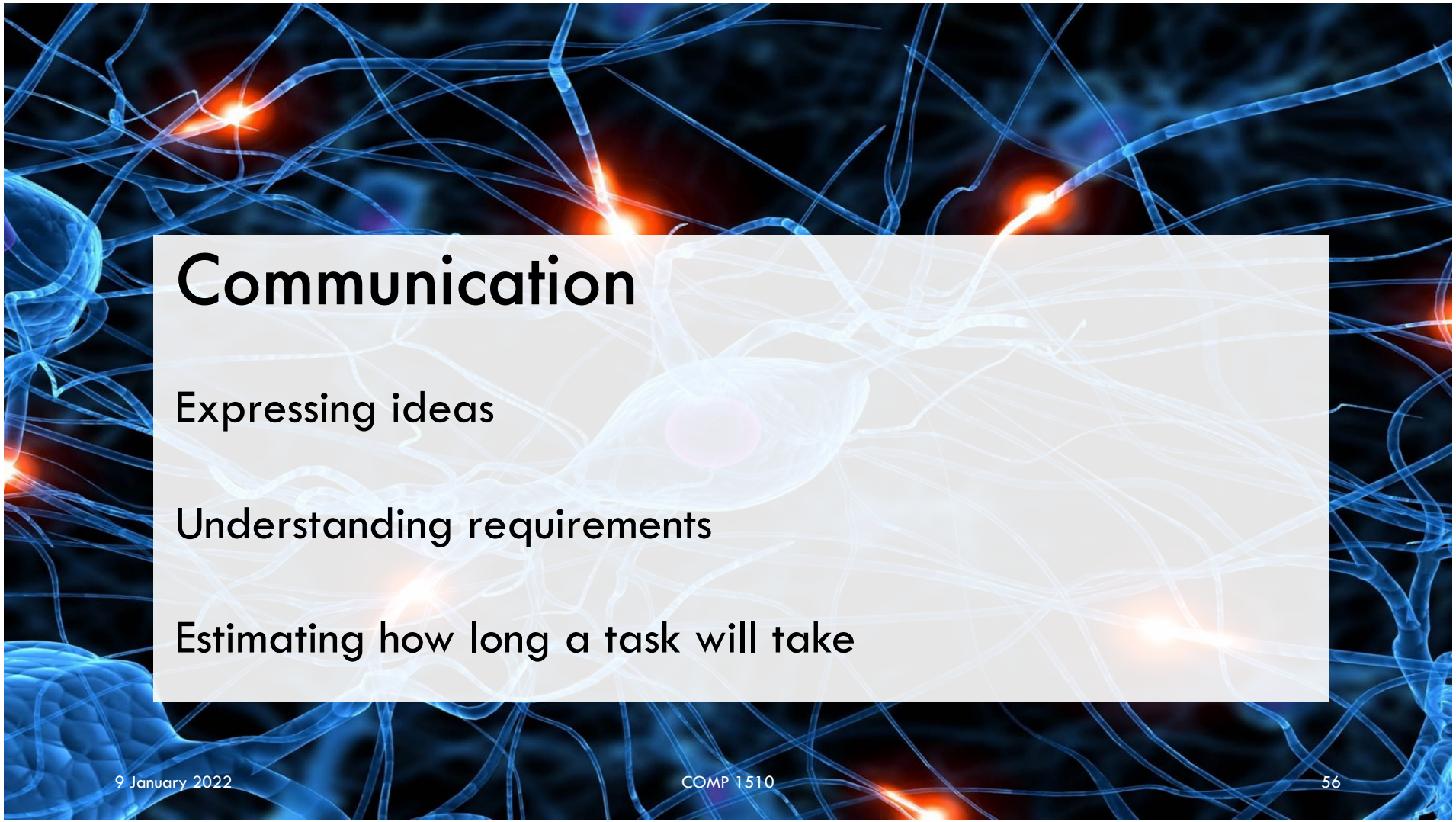


Collaboration

Pair programming

Teamwork

Cross-pollination

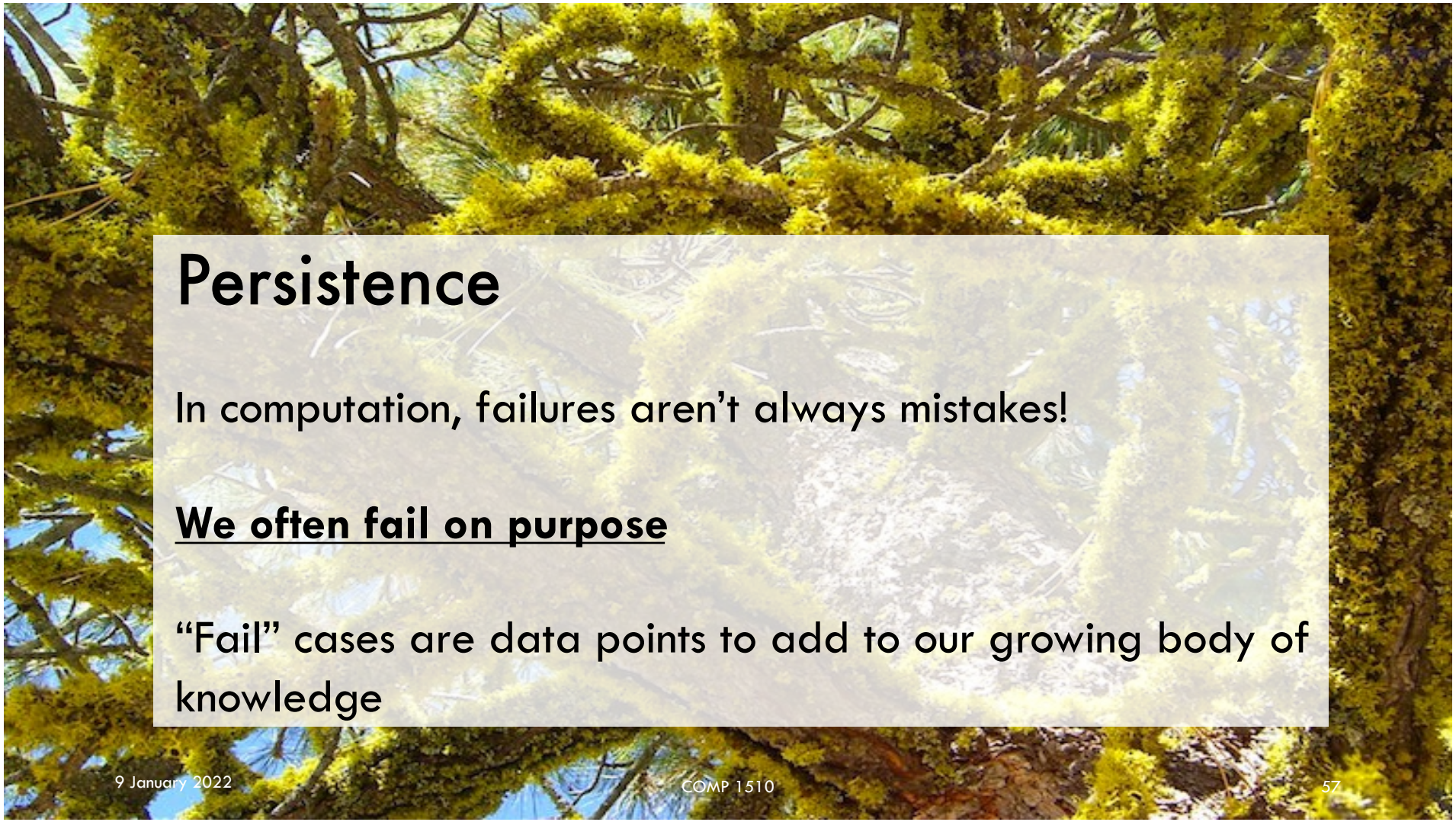


Communication

Expressing ideas

Understanding requirements

Estimating how long a task will take

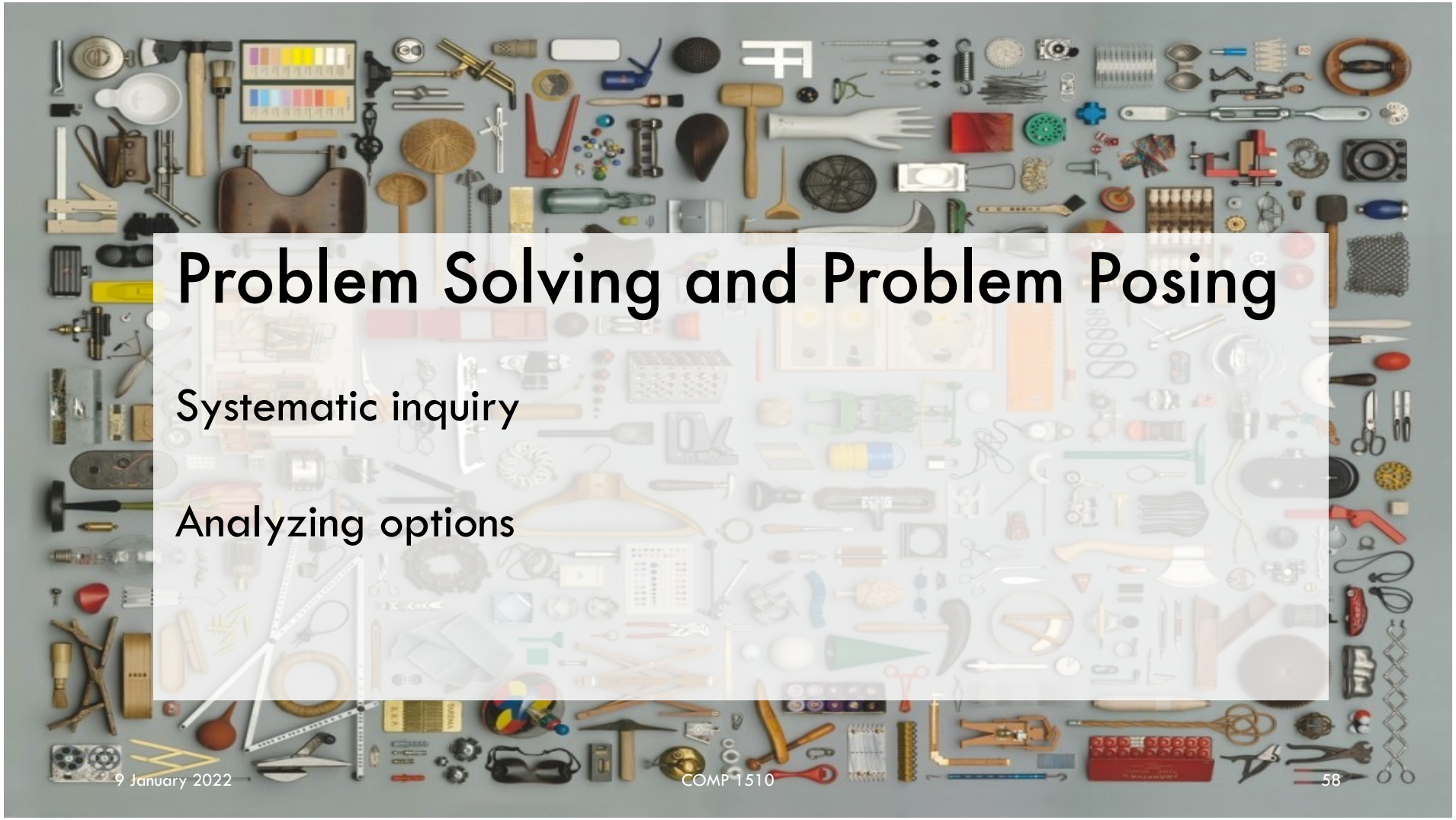


Persistence

In computation, failures aren't always mistakes!

We often fail on purpose

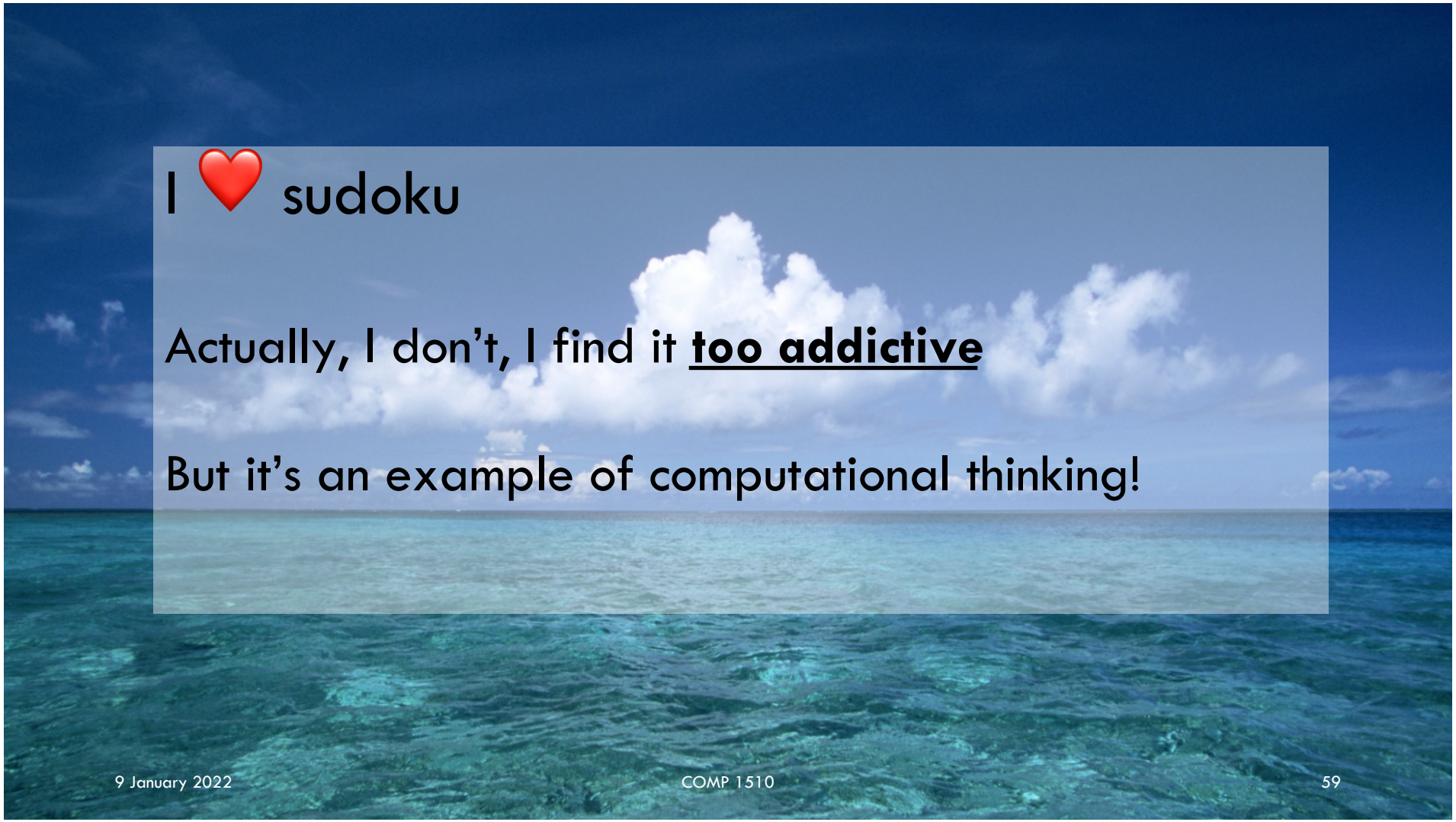
"Fail" cases are data points to add to our growing body of knowledge



Problem Solving and Problem Posing

Systematic inquiry

Analyzing options


I ❤️ sudoku
Actually, I don't, I find it **too addictive**
But it's an example of computational thinking!

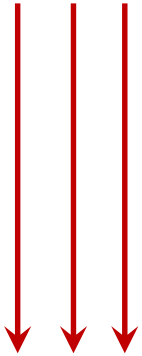
How to sudoku?

7	8	1	2	4	6	9		
	3	9			1	4	7	
4	5	2		9	8	6		
	9	5		2			1	8
2			7	1		5	3	9
1	7		5			2		6
	2	3	4	6	7	1		5
8	1				5		2	7
	6			8	2	3		

Pause: vocabulary alert!

- A sudoku is a matrix of values
- Matrices have **rows** and **columns**

COLUMNS ARE VERTICAL



7	8	1	2	4	6	9		
	3	9			1	4	7	
4	5	2		9	8	6		
	9	5		2			1	8
2			7	1		5	3	9
1	7		5			2		6
	2	3	4	6	7	1		5
8	1				5		2	7
	6			8	2	3		

ROWS ARE HORIZONTAL

But I don't know how to sudoku, Chris!

	3	4	
4			2
1			3
	2	1	

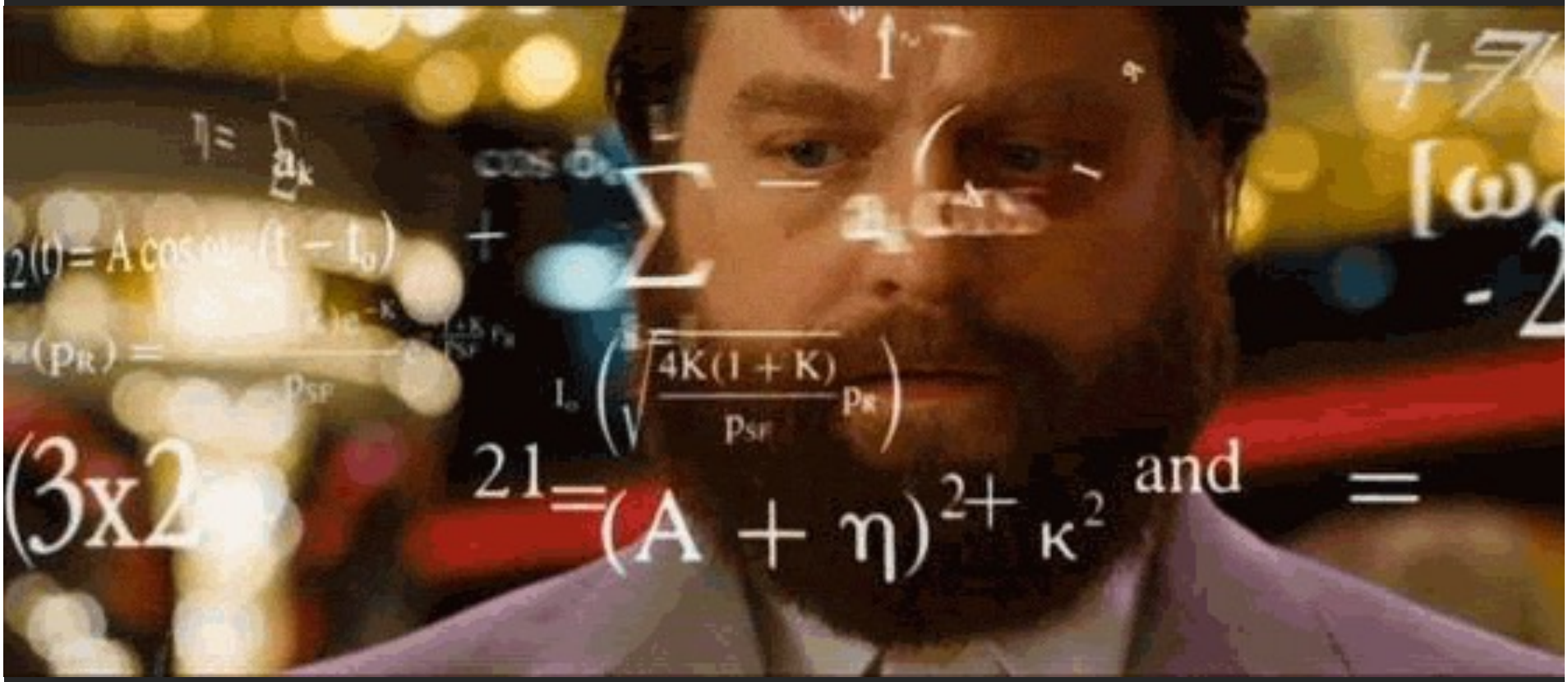
1. Look at the numbers missing in row 1 (1, 2)
2. Look at the numbers that are missing in column 1 (2, 3)
3. Look at the numbers missing in quadrant 1 (1, 2)
4. If there is only one number missing from all three sets, that is the number that goes in the upper left cell (2!)
5. If there is a second number missing from all three sets, continue to the next cell and return when we have more information.*

* This is an algorithm, btw: a recipe or sequence of how-to steps.

Where's the CT in this?

First let's see how we'd figure out what's missing from row 2 column 3:

1. Look at the numbers missing from row 2
2. Look at the numbers missing from column 3
3. Look at the numbers missing from quadrant 2
4. If there is only one number missing from all three sets, that's the number we want
5. If there is a second number missing from all three sets, continue to the next square and come back when you have more information.


$21 = (A + \eta)^2 + \kappa^2$ and
$\sqrt{\frac{4K(1+K)}{P_{SF}}} P_R$
P_{SF}

So where's the computational thinking?

1. **Decomposition**: we took a big problem and broke it down into small, manageable, solvable problems
2. **Pattern matching**: the instructions for each square are basically the same
3. **Abstraction**: we can remove details that are too specific so that one instruction works for multiple problems, i.e.,

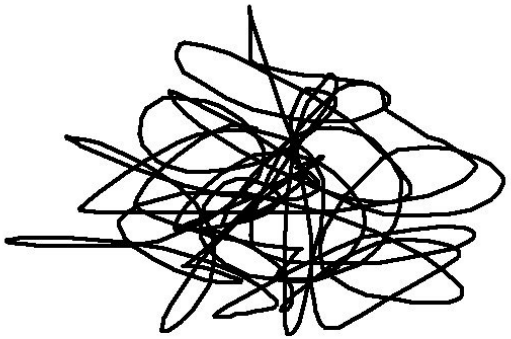
   Look at the numbers that are missing from row _____ .
4. **Algorithm**: we can generate a final set of instructions for solving a sudoku.

STRUCTURED CODE

Structured programming I

During the Wild, Wild West days of programming, we used to call our code Spaghetti Code

It was hard to understand!



It was hard to maintain!

It was hard to read!

(It was a hot mess)

Programming >>> Physics

- Edsger Wybe Dijkstra concluded (as a student in the 50s) that the intellectual challenge of programming was greater than the intellectual challenge of theoretical physics
- Programming was that hard!
- Programs of any complexity were too difficult for humans to manage
- Programmers needed to agree and adhere to a common structure or set of structures…

Structured programming II

- In 1968, Dr. Dijkstra proved mathematically that the unrestrained use of jumps (GOTO statements) is harmful to program structure
- Using GOTO statements prevents modules from being decomposed, and prevents divide-and-conquer algorithms
- We can replace those jumps with **sequence, selection, repetition, and indirection**
- Structured programming using these control structures imposed discipline on the flow of control.

Control structures

In 1966, Böhm and Jacopini proved that all programs can be constructed from just three structures:

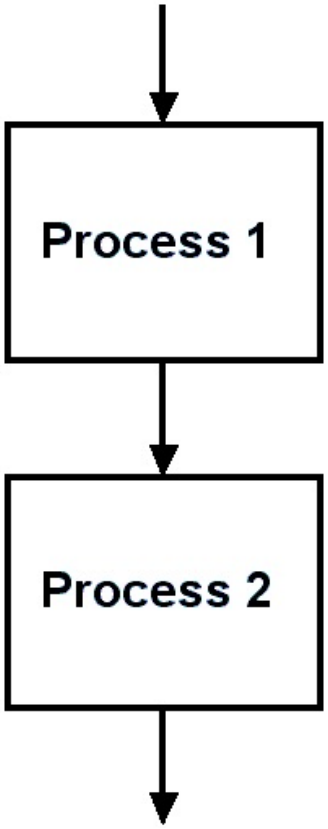
1. Sequence
2. Selection
3. Repetition

In the years since 1966, we have added indirection as a fourth control structure:

4. Indirection.

Sequence

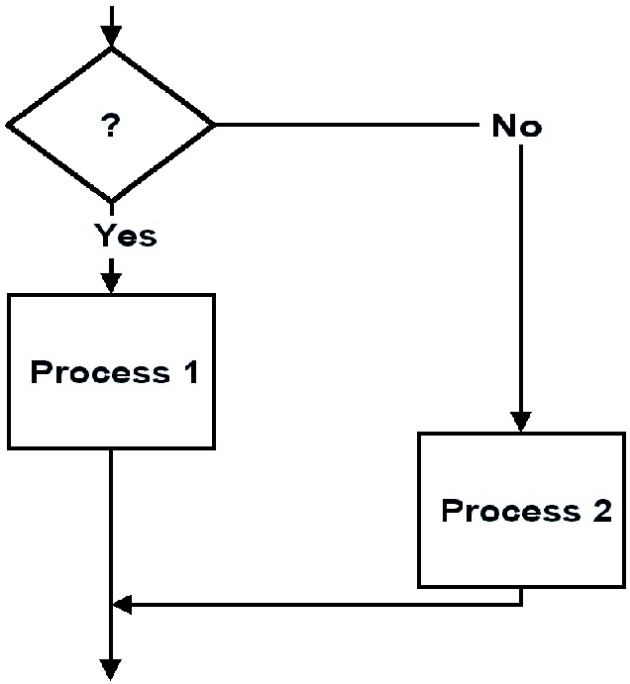
- Top to bottom
- Left to right
- **Ordered statements** executed in order
- Line after line after line after line…
- Each box can be replaced by another structure, which can be replaced by another structure, which can be replaced by another…*



* You will learn about this, recursion, later this year

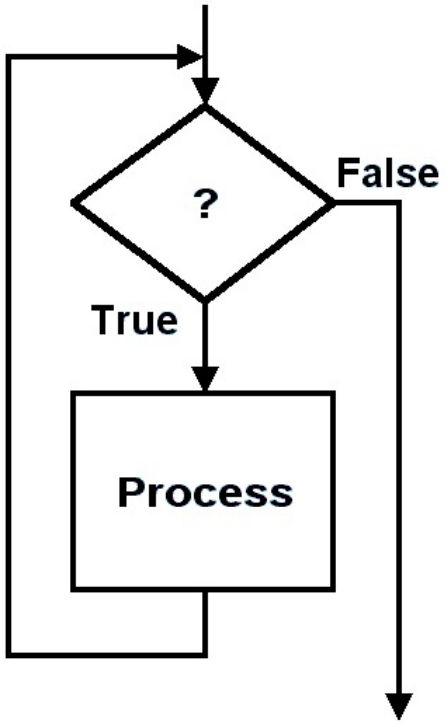
Selection

- Sometimes we need to **make a choice** that depends on the state of the program
- If X is true, do Y
- If X is true, do Y, else do Z
- If A is less than B, do C
- If T is equal to U, do V, else do W
- Then we continue with the sequence…



Repetition aka iteration

- Sometimes we want to repeat a command
- Counting
- **Looping**
- While X is true, do Y
- For each A in the group called B, do C
- Do S while T is less than W
- Then we continue with the sequence…



Indirection

- *It is tempting to write programs as one long, long, long sequence*
- This is challenging to:
  - Understand
  - Decode and debug
  - Maintain
- *Programmers divide code into modules*
  - Self-contained "chunks" of code that allows a sequence of statements to be referenced by a single statement
  - Subroutines, functions, procedures, structures, subprograms…
  - Easy to share, maintain, debug, grow, etc.

FLOWCHARTS

How do we design our programs?

- We use the four fundamental control structures
- Each of the control structures can be represented as an element of a flowchart, or a connected flowchart (indirection)
- A flowchart represents a workflow or process (a program!)
- Programmers like to use flowcharts to help design algorithms
- We can call this a **diagrammatic representation**
- You can use flowcharts this year to help us become great programmers
- Let's examine the fundamental parts of a flowchart

ANSI flowchart standards

- In the 1960s the **American National Standards Institute** set standards for flowcharts and the symbols we use in them
- In 1970 the ISO **International Standards Organization** adopted these symbols
- The current standard was revised in 1985
- *Everyone who's anyone makes flowcharts this way*
- Generally, flowcharts flow from:
  - Top to bottom, and
  - Left to right.

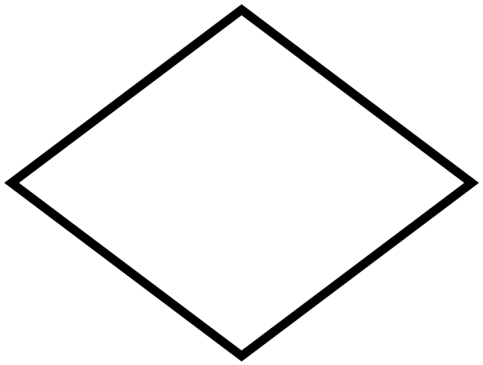
Symbols we will use

Terminal represents the START and the END of our process

Directed line represents the flow of control

Process represents some operation

Symbols we will use

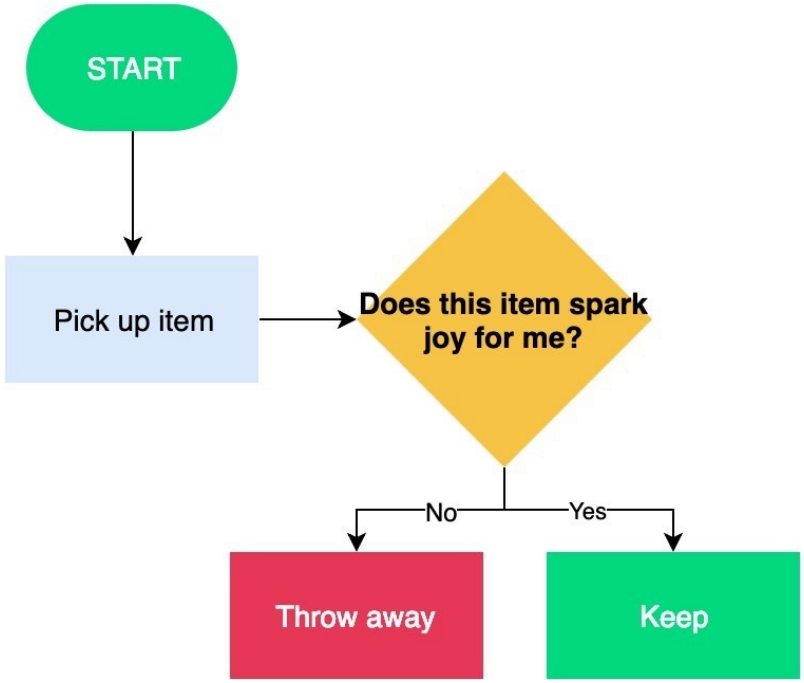


Decision shows a conditional operation that determines which path to take (usually a yes/no true/false question)

Input/Output represents entering data or displaying results

Let's try this out

- Can you create a flowchart that tells me how to:
  1. Ask someone for two numbers, add them, and display the result?
  2. Ask someone for their name and print it five times?



What's missing here?

Programming is like making flowcharts

- To write a program, we:
  1. Start by defining the goal we want to achieve
  2. Split the goal into a series of tasks that are involved in solving the puzzle
  3. We can (possibly mentally) represent this workflow as a flowchart with a start and and end
  4. Then we code each step. Every step for every task shown in the flowchart needs to be written in Python.
- This is a simplification, but it's a good starting model*.

* Flowcharts are your friend.

That's it for week 02!

Any questions?