Introduction to Computing

Understanding How Computers Work

Today We'll Explore

1. **Why computational skills matter** for your research career
2. **How computers actually work** - from binary to complex analysis
3. **Software building blocks** - the fundamental concepts
4. **Computational thinking** - systematic problem-solving approach
5. **AI tools and why understanding matters** - using AI safely and effectively

Section 1: Why This Matters for Your Career

Why Are We Here?

- You're studying drug sciences because you want to contribute to developing **better treatments**.
- Whether you end up working in computational modeling, in-vitro research, or any other area of pharmaceutical research, **you'll be using computer tools** every day. Programming isn't only for in-silico researchers.
- Even if you don't end up programming yourself, there's **real value** in understanding the **basics** of what's actually happening when you click those buttons in your research software or what the script that a colleague provided actually does.

Modern Drug Research

- **Scenario:** A researcher needs to analyze the effects of 1000 compounds on cell growth.
- **Current reality:** Upload data → automated analysis → Results in hours.
- **This is possible because:** Either someone in the group knows how to program, or specialized software was developed for this task.
- **Manual approach (without computational tools):** Measure each sample individually, calculate results by hand → Maybe several weeks of work.
- **The critical gap:** However, a lot of researchers know **how** to use the software, but not **what** it's actually doing.

The Hidden Problem

This gap in understanding creates issues:

- **Misinterpreted results** when you can't evaluate if the analysis fully makes sense (less likely but can happen)
- ⚠️ **Missed innovations** because you can't adapt tools to new problems
- ⚠️ **Complete dependence** on others when something goes wrong
- **Limited career growth** in an increasingly computational field

The opportunity: Understanding computation transforms you **from a tool user into a tool creator**, which is slowly but surely becoming a major career differentiator in research. So this isn't about becoming a programmer - it's about **understanding enough to be an effective scientist** in a computational world.

Common Misconceptions About Programming

- **Misconception 1:** "Programming is only for people doing computational research"
- **Misconception 2:** "It's all about memorizing syntax"
- **Misconception 3:** "I need to become an expert to benefit"
- **Reality:** Basic understanding **transforms how you approach any research problem**, and with AI, programming is becoming **more accessible than ever** but **still requires critical thinking and problem-solving skills** and **basic computational literacy**, will tell you later why.

Your Opportunity as Future Research Leaders

- **The innovation challenge:** The most groundbreaking research often requires tools that **don't exist yet**. When you're working on cutting-edge problems, there may be no software available to do what you need to do.
- **You would have an advantage if:** Learn to understand and create computational solutions for biological problems.
- **The reality:** Sometimes just a simple script can solve problems that no existing software can handle. Even if you don't write the code yourself and you have a computational colleague (who may not have your specific pharmaceutical expertise), being able to **think like a computer scientist and communicate your scientific needs clearly and simply** (so that they just have to implement what you explain) would save you hours and hours of back and forth.

Section 2: How Computers Actually Work

The Universal Pattern

Every single computation follows the same basic pattern:

INPUT → ALGORITHM (Process) → OUTPUT

This might seem obvious, but it's **powerful** because computers can **repeat this pattern perfectly**, millions of times, **without getting tired, distracted, or making calculation errors**.

The Fundamental Question

How does a machine that only understands 1s and 0s ends up analyzing complex molecular structures?

Let's build up from the basics...

Example: Cell Counting

Manual approach: Look at image → Identify what you think looks like cells → Mark them to avoid double-counting → Count marks

Computational approach:
- **INPUT:** Microscopy image (rows and columns of pixels with color values) + parameters defining cells
- **ALGORITHM:** Convert to numbers → Apply filters (matrix operations) → Identify regions → Count
- **OUTPUT:** Cell count + with computers: coordinates, different types of visualizations and plots

How Computers Work: From 1s and 0s to Complex Analysis

Everything is binary: 1s and 0s (electricity on/off)

Hardware: Processor (billions of transistors) + Memory (RAM + Storage)

The Process: Read binary → Apply patterns through transistor combinations → Store binary → **Repeat perfectly, millions of times**

Your microscopy image = read matrix of numbers → algorithm finds patterns → cell count stored and displayed

PB&J Exact Instructions Challenge

More Research Examples

DRUG SCREENING:

- **Input:** Plate reader data from compound library
- **Process:** Normalize, calculate inhibition for each compound
- **Output:** Ranking of active compounds with values of potency

PROTEIN POCKET ANALYSIS:

- **Input:** PDB file (list of elements/residues and coordinates)
- **Process:** Calculate distances, angles, predict binding pockets based on geometry and physicochemical properties
- **Output:** Prediction of binding pockets

Section 3: Building Blocks of Software

What Is Programming?

Programming is giving the computer a **series of precise instructions** to solve a problem as we said, think of it like **writing a very detailed recipe** that a computer can follow.

Programming Languages

Humans don't write in binary - we use programming languages that translate the instructions to binary

High-level languages: Python, R, MATLAB (believe it or not, closer to human language)
Low-level languages: C, Assembly (closer to machine language)

The Fundamental Building Blocks

Every computer program uses these basic concepts:

1. **Variables** - Storing and retrieving information
2. **Data Structures** - Organizing information
3. **Functions** - Reusable blocks of code
4. **Loops** - Repeating actions systematically
5. **Conditionals** - Making decisions based on data
6. **Files** - Storing the code & saving and sharing results

Variables: Storing Information

Containers that hold data:

- Store **experimental measurements** (e.g. cell count)
- Keep **track of sample** (e.g. drug) names
- Remember **calculation results**

Data Structures: Complex variables for organizing information

Ways to group related information:

- **Lists:** Ordered collections (e.g. list of IC50 values)
- **Dictionaries:** Key-value pairs (e.g. protein name, species, molecular weight)

Functions: Reusable Instructions

Reusable sets of instructions that perform specific tasks, for example with different inputs:

- Calculate molecular weight from formula
- Convert between concentration units
- Apply the same analysis to multiple datasets

Loops: Repeating Actions

Repeating the same identical process multiple times:

- "For each sample in the experiment, measure absorbance"
- "For each image, count the cells"
- "For each compound, calculate the IC50 value"

Conditionals: Making Decisions

Making decisions based on data:

- "If concentration is above threshold, flag as toxic"
- "If cell viability is below 80%, discard sample"
- "If results are inconsistent, repeat experiment"

Files: Permanent Storage

Saving and sharing results permanently:

- Raw data files from instruments
- Processed analysis results
- Reports and visualizations
- Sharing data with collaborators

Putting It All Together - Cell Counting

- **Variables:** Store image data, cell parameters, results
- **Data Structures:** List of cell coordinates (coordinates themselves being a list of x, y and z values), dictionary of cell properties
- **Functions:** `load_image`, `apply_filter`, `count_cells`. If you need to change how filtering works for example, you only update one function (programming also teaches you how to think in a modular and compartmentalized way, which is very useful for problem solving in general)
- **Loops:** "For each pixel" → "For each detected region"
- **Conditionals:** "If region matches cell parameters, then it's a cell"
- **Files:** Save original image, processed results, analysis report

Section 4: Computational Thinking

Computational Thinking

A systematic problem-solving approach that breaks complex challenges into manageable parts

The four key principles:

1. **Decomposition** - Break into smaller pieces
2. **Pattern Recognition** - Find similarities and trends
3. **Abstraction** - Focus on what matters most
4. **Algorithm Design** - Create repeatable procedures

Decomposition: Break It Down

The Problem: "Why isn't my cell culture growing?"

Decomposed into testable pieces:

- Media composition and pH
- Incubator temperature and CO_2 levels
- Contamination check
- Cell line authenticity
- Passage number and age

Pattern Recognition: Find the Trends

The Situation: Your drug screening results seem inconsistent

Patterns to look for:

- Do compounds with similar structures show similar effects?
- Are certain experimental days always different?
- Do specific plate positions consistently behave oddly?

Abstraction: Focus on What Matters

- **The Challenge:** Modeling how a drug reaches its target in the body
- **Essential features:** Drug concentration, binding affinity, target location
- **Ignore for now:** Individual blood cell interactions, minor metabolites

Algorithm Design: Create Repeatable Procedures

The Goal: Optimize cell transfection efficiency

Your Algorithm:

1. Test DNA concentrations: 0.5, 1.0, 2.0 µg
2. For each concentration, test reagent ratios: 2:1, 3:1, 4:1
3. Measure efficiency 24h and 48h post-transfection
4. Select best combination and validate with biological replicates

Computational Thinking in Action: Software Example

Research Challenge: "Predict which small molecules will bind to my protein target"

Decomposition: Structure preparation → Ligand library → Docking parameters → Docking & Scoring → Analysis
Pattern Recognition: Do high-scoring compounds share structural features? Binding modes?
Abstraction: Focus on binding affinity predictions, ignore solvent dynamics details
Algorithm Design: Systematic pipeline from structure to ranked compound list

Computational Thinking in Action: Lab Example

Real Research Scenario: "My Western blot results are inconsistent"

- **Decomposition:** Protein extraction → SDS-PAGE → Transfer → Antibodies → Detection
- **Pattern Recognition:** Are certain samples always problematic? Specific days? Antibody batches?
- **Abstraction:** Focus on the steps most likely to cause variability (usually antibodies or transfer)
- **Algorithm Design:** Create systematic troubleshooting protocol testing one variable at a time

Section 5: AI Tools and Why Understanding Matters

AI Tools and Why Understanding Matters

The big question: If AI can write code, why learn programming fundamentals?

The answer: Understanding lets you use AI safely and effectively, rather than blindly trusting it

The Hidden Dangers of AI Code

- **Plausible but Wrong:** Code runs smoothly but produces incorrect results (for example calculations seem correct but are actually wrong because the wrong formula was used, you may end up publishing incorrect conclusions)
- **Missing Biology:** Ignores experimental constraints and statistical requirements
- **Debugging Nightmare:** You're stuck when (not if) things break
- **Security Risks:** Vulnerabilities and potential data corruption

Smart AI Usage: Be the Director, Not the Audience

Effective AI prompts:
- "Help me read this `.czi` microscopy file format"
- "Optimize this cell counting loop I designed"
- "What edge cases should I test in my IC50 calculation?"

Dangerous AI prompts:
- "Write a complete proteomics analysis pipeline"
- "Create statistical analysis for my drug screen data"

→ Will give you way more **details about how to use AI properly** since I know you will use it anyway.

What Comes Next

Next lesson: Computer-Based Problem Solving
- Break down real research problems into computational steps
- Practice thinking systematically about data analysis
- Discuss different approaches to solving the same problem
- **No programming syntax yet** - just logical thinking

Then: Python Programming Basics
- Learn the actual programming language
- Implement the solutions you designed in lesson 2
- Need a Student GitHub account starting here

Finally: Solve Real Problems in Code
- Apply everything to solve the problems from lesson 2
- Create working programs for research tasks

Assignment

Think of something you'd love to automate - from your studies, work, or personal life / hobbies.

Create a PDF explaining:
- **What** would you want to automate?
- **INPUT:** What information/materials do you start with?
- **ALGORITHM:** What are the step-by-step processes? (Think PB&J video - be specific!)
- **OUTPUT:** What would you want as the final result?

Examples: Lab protocol, data analysis, meal planning, workout routine, literature review process, whatever!

Deadline: Next Wednesday, 24th of September, 11:59pm

Goal: Practice thinking systematically about breaking down problems into computational steps.

Thank You for your attention!
Questions?