Python for Synthetic Biology

Executable workflows for biological engineering

Gonzalo Vidal

Contents

1 Python for Synthetic Biology . . . 11
1.1 Why this book exists . . . 11
1.2 How to read this book . . . 11
1.3 A working philosophy . . . 11
1.4 First tiny example . . . 12

2 Preface . . . 13
2.1 What makes this book different . . . 13
2.2 What this book can grow into . . . 13

3 Roadmap . . . 15
3.1 Part I - Foundations . . . 15
3.2 Part II - Biological Data in Python . . . 15
3.3 Part III - Models, Standards, and Design Automation . . . 15
3.4 Part IV - Case Studies from Real Lab Software . . . 15
3.5 Part V - The Future Lab . . . 15
3.6 Companion tools to develop alongside the book . . . 15

I Part I - Foundations

4 Why Python for Synthetic Biology . . . 19
4.1 Synthetic biology is already computational . . . 19
4.2 From one-off script to research software . . . 19
4.3 Why Python works so well here . . . 20
4.3.1 It reads almost like pseudocode . . . 20
4.3.2 It scales from tiny tasks to large systems . . . 20
4.3.3 It has excellent scientific tooling . . . 20
4.3.4 It plays well with notebooks . . . 21
4.4 A first example: turning measurements into decisions . . . 21
4.5 Code lets you preserve reasoning . . . 22
4.6 Python supports the whole design-build-test-learn cycle . . . 22
4.6.1 Design . . . 22
4.6.2 Build . . . 23
4.6.3 Test . . . 23
4.6.4 Learn . . . 23
4.7 Reproducibility is a practical skill, not a slogan . . . 23
4.8 Open source matters in synthetic biology . . . 23
4.9 What this book will ask of you . . . 24
4.10 A miniature design task . . . 24

4.11 How this book is organized . . . 25
4.12 Exercises . . . 25
4.13 Key ideas from this chapter . . . 25

5 Python Basics for Biologists . . . 27
5.1 Running code is part of reading code . . . 27
5.2 Variables: naming biological quantities . . . 27
5.3 Numbers and arithmetic . . . 28
5.4 Strings: sequences and labels . . . 28
5.4.1 Useful string methods . . . 29
5.5 Writing your first function . . . 29
5.5.1 Make functions safer . . . 29
5.6 Conditionals: making decisions in code . . . 30
5.7 Lists: working with collections . . . 30
5.7.1 Looping through a list . . . 31
5.8 Building lists from loops . . . 31
5.9 Dictionaries: attaching names to values . . . 31
5.10 A list of dictionaries: a tiny experimental dataset . . . 32
5.11 Filtering data with conditionals . . . 33
5.12 Counting things with dictionaries . . . 34
5.13 Reading simple tabular data . . . 35
5.14 Strings plus dictionaries: reverse complements . . . 36
5.15 A slightly richer example: screening constructs . . . 36
5.16 Common beginner mistakes . . . 37
5.16.1 Forgetting that Python starts counting at zero . . . 38
5.16.2 Mixing strings and numbers . . . 38
5.16.3 Using = when you mean comparison . . . 38
5.16.4 Forgetting to return a value from a function . . . 38
5.17 Style matters because science is collaborative . . . 38
5.18 Practice: modify the code, do not only read it . . . 39
5.19 Exercises . . . 39
5.20 Key ideas from this chapter . . . 39

6 Jupyter, Files, and Reproducibility . . . 41
6.1 Code in science has more than one form . . . 41
6.1.1 Scripts . . . 41
6.1.2 Notebooks . . . 41
6.1.3 Rendered documents . . . 41
6.2 Why notebooks became so influential . . . 42
6.3 Your computational environment matters . . . 42
6.4 Virtual environments: keeping projects separate . . . 43
6.5 A reproducible project has a shape . . . 43
6.6 Paths are part of scientific thinking . . . 43
6.7 Writing a small dataset to disk . . . 44
6.8 Reading data back in . . . 44

6.9 Notebook state is helpful and dangerous . . . 46
6.10 Small functions make notebooks stronger . . . 46
6.11 Saving processed data . . . 47
6.12 Saving metadata alongside results . . . 48
6.13 Summaries should be reproducible too . . . 48
6.14 Randomness should be controlled when it matters . . . 49
6.15 Reproducibility is social as well as technical . . . 50
6.16 Notebooks versus scripts: when to use each . . . 50
6.17 How Quarto fits into this workflow . . . 51
6.18 A full miniature workflow in one view . . . 51
6.19 Common beginner mistakes in reproducible work . . . 51
6.19.1 Hard-coding machine-specific paths . . . 51
6.19.2 Editing raw data in place . . . 51
6.19.3 Forgetting which cell created a result . . . 51
6.19.4 Installing packages globally without tracking them . . . 52
6.19.5 Mixing exploration with final reporting carelessly . . . 52
6.20 Practice ideas . . . 52
6.21 Exercises . . . 52
6.22 Key ideas from this chapter . . . 52

II Part II - Biological Data in Python

7 Sequences as Data . . . 55
7.1 Sequences look like text, but they are structured text . . . 55
7.2 Indexing and slicing sequences . . . 55
7.3 Length is a biological property . . . 56
7.4 Counting bases . . . 56
7.5 Cleaning and validating sequence input . . . 57
7.6 Complement and reverse complement . . . 58
7.7 Searching for motifs . . . 58
7.8 DNA to RNA . . . 59
7.9 Translation: codons to amino acids . . . 60
7.10 Open reading frame checks . . . 61
7.11 Sliding windows and local GC content . . . 61
7.12 FASTA: the first real file format many people meet . . . 62
7.13 Summarizing a small sequence library . . . 63
7.14 Screening designs against simple rules . . . 64
7.15 Ranking candidate sequences . . . 64
7.16 K-mers: treating sequences as overlapping words . . . 65
7.17 Sequences as records, not just strings . . . 65
7.18 A mini workflow: from FASTA to a design report . . . 66
7.19 When to stop writing your own sequence utilities . . . 67
7.20 Biopython: from strings to standardized sequence records . . . 67

7.21 `Seq`: a biological sequence object 68
7.22 `SeqRecord`: a sequence plus metadata 68
7.23 Reading FASTA with `SeqIO` 69
7.24 GenBank: richer records with annotations and features 69
7.25 Standardizing your own records 71
7.26 BLAST with Biopython 71
7.27 Why this matters in synthetic biology 73
7.28 Exercises 74
7.29 Recap 74

8 Tabular Experimental Data 75
8.1 Why tidy data matters 75
8.2 A wide table is often the first thing you get 75
8.3 Reshaping wide data into tidy data 76
8.4 Building a tidy experiment table 77
8.5 Inspecting a DataFrame 78
8.6 Selecting columns 79
8.7 Filtering rows 80
8.8 Adding derived columns 81
8.9 Grouping and summarizing replicates 81
8.10 Quantifying induction 82
8.11 Missing values happen 83
8.12 Reading tidy data from a CSV file 84
8.13 Merging measurement tables with metadata 85
8.14 Selecting one table shape for downstream work 86
8.15 Saving processed data 86
8.16 A small end-to-end example 87
8.17 Recap 87
8.18 Exercises 87

9 Networks, Circuits, and Graphs 89
9.1 Graphs as tidy data 89
9.2 Installing NetworkX 90
9.3 A first regulatory network 90
9.4 Building a directed graph from a tidy edge table 91
9.5 Nodes, edges, predecessors, and successors 91
9.6 Using a node table for metadata 92
9.7 Drawing a small circuit 93
9.8 Keeping edge metadata tidy 94
9.9 Finding paths through a circuit 94
9.10 Feed-forward logic 95
9.11 Detecting feedback loops 95
9.12 When an adjacency matrix is useful 96

9.13 Converting a graph back into a tidy edge table . . . 96
9.14 Assembly dependencies as a directed acyclic graph . . . 97
9.15 Detecting impossible workflows . . . 98
9.16 Merging tidy metadata with graph results . . . 98
9.17 Choosing between a table and a graph . . . 99
9.18 Exercises . . . 99
9.19 Recap . . . 100

III Part III - Models, Standards, and Design Automation

10 Modeling Gene Expression . . . 103
10.1 Why model gene expression? . . . 103
10.2 Variables, parameters, and rates . . . 103
10.2.1 Variables . . . 104
10.2.2 Parameters . . . 104
10.2.3 Rules of change . . . 104
10.3 The simplest expression model: production and loss . . . 104
10.4 Computing the analytical solution . . . 104
10.5 Visualizing the approach to steady state . . . 105
10.6 Simulating the same model with Euler's method . . . 106
10.7 Time scales matter . . . 108
10.8 Building a tidy summary table . . . 109
10.9 A two-stage model: mRNA and protein . . . 110
10.10 Simulating the two-stage model . . . 110
10.11 Comparing promoter strengths . . . 112
10.12 Induction and Hill functions . . . 113
10.12.1 Activation . . . 113
10.12.2 Repression . . . 113
10.13 Coding Hill functions . . . 114
10.14 Linking induction to protein production . . . 115
10.15 Parameter sweeps should also be tidy . . . 116
10.16 Simulating an induction time course . . . 118
10.17 A note on deterministic vs stochastic models . . . 119
10.18 Choosing the right level of model complexity . . . 119
10.19 Exercises . . . 120
10.20 Recap . . . 120

11 SBOL and Tooling . . . 121
11.1 Why not stop at FASTA or GenBank? . . . 121
11.2 SBOL as a design language . . . 122
11.3 The tooling layers we will use . . . 122
11.3.1 `pySBOL3` . . . 122
11.3.2 `SBOL-utilities` . . . 122
11.3.3 VisBOL . . . 122
11.3.4 DNAplotlib . . . 123

11.4 Installing the packages . . . **123**
11.5 A first SBOL document with `pySBOL3` . . . **123**
11.6 What just happened? . . . **125**
11.6.1 `Document` . . . 125
11.6.2 `Component` . . . 125
11.6.3 `Sequence` . . . 126
11.6.4 `SubComponent` . . . 126
11.6.5 `Constraint` . . . 126
11.7 Representing function, not only sequence . . . **126**
11.8 Writing the design to disk . . . **127**
11.9 Visualizing the design with VisBOL . . . **127**
11.10 Programmable visualization with DNAplotlib . . . **128**
11.11 Using `SBOL-utilities` to reduce boilerplate . . . **129**
11.12 Converting older sequence formats into SBOL . . . **130**
11.12.1 FASTA to SBOL . . . 130
11.12.2 GenBank to SBOL . . . 130
11.13 A practical mindset for using SBOL . . . **132**
11.14 Recommended workflow . . . **132**
11.15 Exercises . . . **132**
11.16 Recap . . . **132**

12 Design-Build-Test-Learn . . . **135**
12.1 Why DBTL matters computationally . . . **135**
12.2 One DBTL cycle as a data transformation pipeline . . . **135**
12.3 A thesis-derived software stack for closing the loop . . . **136**
12.3.1 LOICA: design and model-driven iteration . . . 136
12.3.2 PUDU: build planning and liquid-handling automation . . . 136
12.3.3 Flapjack: test and learn . . . 136
12.4 Three levels of workflow closure . . . **137**
12.5 The software architecture of a closed loop . . . **137**
12.6 Using LOICA at the design stage . . . **138**
12.6.1 Adding simulation context . . . 139
12.6.2 Uploading and characterizing the result . . . 139
12.7 LOICA as a Python lesson . . . **140**
12.8 Using PUDU at the build stage . . . **140**
12.8.1 A minimal Loop assembly example . . . 140
12.8.2 Simulating a protocol before running it . . . 140
12.8.3 PUDU and human-readable output . . . 141
12.8.4 A plate-setup style example . . . 141
12.9 PUDU as a Python lesson . . . **141**
12.10 Using Flapjack for test and learn . . . **141**
12.10.1 Connecting with `pyFlapjack` . . . 142
12.10.2 Plotting measurements and analysis results . . . 142
12.10.3 Getting tabular outputs for further learning . . . 143
12.10.4 Flapjack as the bridge between test and design . . . 143
12.11 A worked conceptual loop . . . **143**
12.12 Why standards matter here . . . **144**

12.13 A practical teaching strategy . . . **144**
12.13.1 1. Start with simulated DBTL . . . 144
12.13.2 2. Add real experimental data . . . 144
12.13.3 3. Add automated build concepts . . . 144
12.14 What students should learn from this chapter . . . **145**
12.14.1 1. A workflow is a chain of data structures . . . 145
12.14.2 2. Abstractions matter . . . 145
12.14.3 3. Metadata is part of the science . . . 145
12.14.4 4. Simulation and experiment should talk to each other . . . 145
12.14.5 5. Standards make software ecosystems possible . . . 145
12.15 Minimal installation notes . . . **145**
12.16 Exercises . . . **145**
12.17 Closing thought . . . **146**

IV Part IV - Case Studies from Real Lab Software

13 Advanced Computational Biology . . . **149**
13.1 What makes this chapter different from earlier modeling chapters? . . . **149**
13.2 The biological story . . . **149**
13.3 Growth profiles are spatial data . . . **150**
13.4 Building the growth profile in Python . . . **150**
13.5 Visualizing the growth profile . . . **151**
13.6 From growth profile to spatial phase differences . . . **151**
13.7 A toy spatial oscillator . . . **152**
13.8 Converting a kymograph into tidy data . . . **153**
13.9 Plotting a kymograph . . . **154**
13.10 Static rings versus traveling waves . . . **155**
13.11 Extracting simple summaries from the toy model . . . **156**
13.12 How maximal expression changes wavelength . . . **157**
13.13 How degradation changes temporal progression . . . **158**
13.14 Why kymographs are such a powerful representation . . . **159**
13.15 Thinking in layers: full model, reduced model, analysis pipeline . . . **159**
13.15.1 Layer 1: a biological mechanism . . . 159
13.15.2 Layer 2: a physical profile . . . 159
13.15.3 Layer 3: a reduced mathematical model . . . 160
13.15.4 Layer 4: a richer computational model . . . 160
13.15.5 Layer 5: analysis outputs . . . 160
13.16 What the repository teaches about computational workflow . . . **160**
13.17 A computational biologist's checklist for spatial circuit models . . . **160**
13.17.1 What is the spatial coordinate? . . . 160
13.17.2 What couples space to circuit dynamics? . . . 160
13.17.3 What output representation makes the phenomenon interpretable? . . . 161
13.17.4 Can you build a reduced model before the full model? . . . 161
13.18 A small extension: exporting tidy simulation summaries . . . **161**
13.19 What this chapter does *not* capture . . . **161**

13.20 Exercises . . . 162
13.21 Recap . . . 162

14 Myers Lab Case Studies . . . 163
14.1 Draft focus . . . 163
14.2 Candidate tools . . . 163
14.3 Draft note . . . 163

15 Personal Projects . . . 165
15.1 Draft focus . . . 165
15.2 Candidate threads . . . 165
15.3 Draft note . . . 165

V Part V - The Future Lab

16 The DRAGGON Lab Vision . . . 169
16.1 Draft focus . . . 169
16.2 Themes . . . 169
16.3 Draft note . . . 169

17 Capstone Projects . . . 171
17.1 Draft focus . . . 171
17.2 Candidate capstones . . . 171
17.3 Draft note . . . 171

18 References . . . 173

Bibliography . . . 175

1. Python for Synthetic Biology

Executable workflows for biological engineering

```python
message = "Welcome to Python for Synthetic Biology"
message
```

```
'Welcome to Python for Synthetic Biology'
```

This book teaches Python through the workflows, abstractions, and software patterns that matter in synthetic biology.

Instead of learning programming in the abstract and only later trying to apply it to biology, we build intuition directly from synthetic biology problems:

- representing DNA sequences and annotations
- working with experimental measurements in tidy format
- modeling gene expression and regulatory circuits
- structuring design-build-test-learn workflows
- learning from real open-source tools developed in research labs

1.1 Why this book exists

Synthetic biology increasingly depends on software. We design parts, assemble circuits, manage metadata, analyze experiments, run simulations, exchange models, and automate lab workflows through code. Yet many trainees still learn these pieces in fragments.

This book is an attempt to provide a coherent path from first Python scripts to research-grade computational thinking.

It is designed for readers who want to:

- learn Python with examples that feel biologically meaningful
- understand how biological systems become computational objects
- move from scripts to software thinking
- connect teaching examples with real tools and research directions

1.2 How to read this book

You can read the book in order, but it is also designed to support selective reading.

- **Part I** builds the programming foundation.
- **Part II** introduces biological data as Python objects.
- **Part III** moves into models, standards, and design automation.
- **Part IV** uses open-source lab tools as case studies.
- **Part V** looks forward to the design of future synthetic biology labs.

1.3 A working philosophy

Every chapter should answer three questions:

1. What is the biological problem?
2. How should we think about it computationally?
3. What does the Python actually look like?

1.4 First tiny example

```
sequence = "ATGCGTACGTTAG"
length = len(sequence)
gc_fraction = (sequence.count("G") + sequence.count("C")) / length
{"sequence": sequence, "length": length, "gc_fraction": round(gc_fraction, 3)}
```

```
{'sequence': 'ATGCGTACGTTAG', 'length': 13, 'gc_fraction': 0.462}
```

This is intentionally simple. But already we are treating biology as structured, computable information. That shift in perspective will appear again and again throughout the book.

2. Preface

This project sits at the intersection of synthetic biology, computational biology, software engineering, and scientific education.

The book takes inspiration from technical books that treat code as a first-class teaching medium while keeping the prose compact and purposeful. It also takes inspiration from open educational resources that pair conceptual models with executable examples.

2.1 What makes this book different

This is not just a Python book with a few biology examples added on top.

It is a synbio-native programming book.

The core premise is that synthetic biology already has its own computational objects, recurring workflows, and software abstractions:

- sequences
- features and annotations
- constructs and designs
- measurements and metadata
- regulatory networks
- simulation models
- standard representations
- automated DBTL pipelines

Python becomes useful here not only because it is popular, but because it is expressive enough to make these abstractions visible.

2.2 What this book can grow into

Over time, the book can evolve into a broader educational and software platform:

- a technical book
- a workshop resource
- a companion set of teaching libraries
- a bridge between introductory training and research software

That is one reason the repository is structured so that code can mature alongside the text.

3. Roadmap

This page gives a draft map of the book.

3.1 Part I - Foundations

- Why Python for synthetic biology?
- Python basics for biologists
- Jupyter, reproducibility, and computational practice

3.2 Part II - Biological Data in Python

- Sequences as strings, records, and objects
- Experimental data as tidy tables
- Circuits and interactions as graphs

3.3 Part III - Models, Standards, and Design Automation

- Modeling gene expression and regulation
- Design-build-test-learn workflows
- SBOL and software interoperability

3.4 Part IV - Case Studies from Real Lab Software

- RudgeLab tools and architecture lessons
- MyersResearchGroup tools and standards-driven workflows
- Personal projects as bridges between research and product thinking

3.5 Part V - The Future Lab

- The DRAGGON Lab vision
- Capstones and tool ideas that can grow with the book

3.6 Companion tools to develop alongside the book

Possible repositories or modules to create later:

- `synbio_sequences`: teaching utilities for sequence manipulation
- `synbio_dbtl`: minimal DBTL workflow objects
- `synbio_models`: simple dynamical models and plotting helpers
- `synbio_io`: lightweight import and export helpers for educational examples

I

Part I - Foundations

4.2	From one-off script to research software	19
4.3	Why Python works so well here	20
4.4	A first example: turning measurements into decisions	21
4.5	Code lets you preserve reasoning	22
4.6	Python supports the whole design-build-test-learn cycle	22
4.7	Reproducibility is a practical skill, not a slogan	23
4.8	Open source matters in synthetic biology	23
4.9	What this book will ask of you	24
4.10	A miniature design task	24
4.11	How this book is organized	25
4.12	Exercises	25
4.13	Key ideas from this chapter	25
5	**Python Basics for Biologists**	**27**
5.1	Running code is part of reading code	27
5.2	Variables: naming biological quantities	27
5.3	Numbers and arithmetic	28
5.4	Strings: sequences and labels	28
5.5	Writing your first function	29
5.6	Conditionals: making decisions in code	30
5.7	Lists: working with collections	30
5.8	Building lists from loops	31
5.9	Dictionaries: attaching names to values	31
5.10	A list of dictionaries: a tiny experimental dataset	32
5.11	Filtering data with conditionals	33
5.12	Counting things with dictionaries	34
5.13	Reading simple tabular data	35
5.14	Strings plus dictionaries: reverse complements	36
5.15	A slightly richer example: screening constructs	36
5.16	Common beginner mistakes	37
5.17	Style matters because science is collaborative	38
5.18	Practice: modify the code, do not only read it	39
5.19	Exercises	39
5.20	Key ideas from this chapter	39
6	**Jupyter, Files, and Reproducibility**	**41**
6.1	Code in science has more than one form	41
6.2	Why notebooks became so influential	42
6.3	Your computational environment matters	42
6.4	Virtual environments: keeping projects separate	43
6.5	A reproducible project has a shape	43
6.6	Paths are part of scientific thinking	43
6.7	Writing a small dataset to disk	44
6.8	Reading data back in	44
6.9	Notebook state is helpful and dangerous	46
6.10	Small functions make notebooks stronger	46
6.11	Saving processed data	47
6.12	Saving metadata alongside results	48
6.13	Summaries should be reproducible too	48
6.14	Randomness should be controlled when it matters	49
6.15	Reproducibility is social as well as technical	50
6.16	Notebooks versus scripts: when to use each	50
6.17	How Quarto fits into this workflow	51
6.18	A full miniature workflow in one view	51
6.19	Common beginner mistakes in reproducible work	51
6.20	Practice ideas	52
6.21	Exercises	52
6.22	Key ideas from this chapter	52

4. Why Python for Synthetic Biology

Synthetic biology is often described as an engineering discipline for biology. We design DNA, build strains, measure performance, and then learn from the results to make the next design better.

That sounds experimental, and it is. But it is also deeply computational.

A modern synthetic biology project usually includes at least some of the following tasks:

- organizing sequence designs
- planning assemblies and experiments
- controlling instruments or robots
- cleaning plate-reader or microscopy data
- fitting models to growth and expression curves
- comparing circuit variants across conditions
- sharing reusable analyses with collaborators

In other words, even when the center of gravity is the bench, the work surrounding the bench is increasingly software-mediated.

Python has become one of the most useful languages for this kind of work because it sits in the middle of several worlds at once:

- it is readable enough for new programmers
- it is powerful enough for serious data analysis and scientific computing
- it connects well to notebooks, web services, robots, databases, and machine learning tools
- it has a huge open-source ecosystem

This book is about learning Python in that specific context. We are not learning Python as an abstract computer science exercise. We are learning it as a practical language for **designing, measuring, modeling, and automating biological systems**.

4.1 Synthetic biology is already computational

A useful mental shift is this: computational work in synthetic biology is not a side task that starts after the "real" experiment. It is part of the experiment.

Consider a simple characterization workflow for a small promoter library:

1. design a panel of constructs
2. transform or assemble them
3. grow cultures under multiple conditions
4. measure optical density and fluorescence over time
5. normalize the measurements
6. compare variants
7. choose the next set of designs

Only steps 2 and 3 look purely wet-lab. The rest involve representations, decisions, and transformations that are easier, faster, and safer when expressed in code.

A spreadsheet can carry some of this load, but spreadsheets become fragile as soon as the work becomes iterative. File names drift. Columns move. A formula is copied into the wrong cells. The same analysis gets repeated in slightly different ways by different people.

A small Python script is often the first step away from that fragility.

4.2 From one-off script to research software

Most scientists begin with small scripts. They may start by asking for something modest:

- "Can I calculate GC content for all of these sequences?"
- "Can I rename these files automatically?"
- "Can I merge these plate-reader exports?"

- "Can I make this plot every week without clicking through the same steps?"

These are good beginner projects because they solve real problems. They also teach a bigger lesson: many recurring lab tasks are just data transformations with domain-specific meaning.

Once you recognize that, a path opens up:

- a quick script becomes a reusable notebook
- a notebook becomes a small package
- a package becomes a shared lab tool
- a shared tool becomes part of a broader design-build-test-learn workflow

That progression is visible in open-source synthetic biology software. Public lab repositories often start from concrete needs such as design automation, protocol generation, simulation, or data handling, and gradually evolve into broader platforms. The Rudge Lab organization, for example, highlights LOICA for genetic design automation, PUDU for liquid-handling workflows, Flapjack for data management and analysis, and CellModeller for multicellular modeling. The Myers Research Group organization highlights tools and projects such as SynBioSuite, SeqImprove, BuildCompiler, PUDU, and iBioSim. These are useful examples because they show that code in synthetic biology is not limited to data analysis; it spans the entire research cycle. Rudge Lab, Genetic Logic Lab

The goal of this book is not just to help you *use* that software. It is to help you understand how to think in the same way that such tools are built.

4.3 Why Python works so well here

Python is not the only useful language in biology. R is powerful for statistics, Java is still important for parts of computational biology infrastructure, JavaScript matters for interfaces and dashboards, and many domain-specific tools expose their own scripting layers.

But Python is unusually effective as a **bridge language**.

4.3.1 It reads almost like pseudocode

For newcomers, Python is often less intimidating than languages with heavier syntax. Consider this simple example.

```
from collections import Counter

sequence = "ATGATCGGCTTACGAT"
base_counts = Counter(sequence)
base_counts
```

```
Counter({'T': 5, 'A': 4, 'G': 4, 'C': 3})
```

Even if you have never programmed before, the intent is fairly visible:

- store a DNA sequence in a variable
- count each character
- inspect the result

That readability matters in collaborative science. Code is not only executed by computers. It is also read by students, labmates, reviewers, and your future self.

4.3.2 It scales from tiny tasks to large systems

The same language can support very different levels of ambition.

A beginner may write a 10-line function to compute GC content. A more advanced user may build a data-processing pipeline, a simulator, or an API for a lab platform. That continuity means you do not have to switch languages every time your project grows.

4.3.3 It has excellent scientific tooling

Python's ecosystem includes libraries for:

- arrays and numerical computing

- tabular data analysis
- plotting and visualization
- statistical modeling
- optimization and machine learning
- web servers and APIs
- automation and workflow orchestration

Synthetic biology sits at the intersection of all of these.

4.3.4 It plays well with notebooks

Jupyter notebooks are not perfect, but they are extremely useful for education, exploration, and reproducible demonstrations. They combine prose, code, output, and figures in one place.

That is one reason Python feels natural for a book like this. We can explain an idea, run the code directly underneath it, and then inspect the result immediately.

4.4 A first example: turning measurements into decisions

To see why code matters, let us walk through a tiny synthetic biology style task.

Imagine that you tested three genetic constructs and collected endpoint optical density (OD) and fluorescence values. A common first-pass analysis is to compute fluorescence normalized by culture density.

```
measurements = [
    {"construct": "pTac-GFP", "od600": 0.82, "fluorescence": 15420},
    {"construct": "pTet-GFP", "od600": 0.79, "fluorescence": 11210},
    {"construct": "pBAD-GFP", "od600": 0.76, "fluorescence": 8450},
]

for row in measurements:
    row["expression_per_od"] = row["fluorescence"] / row["od600"]

measurements
```

```
[{'construct': 'pTac-GFP',
  'od600': 0.82,
  'fluorescence': 15420,
  'expression_per_od': 18804.87804878049},
 {'construct': 'pTet-GFP',
  'od600': 0.79,
  'fluorescence': 11210,
  'expression_per_od': 14189.87341772152},
 {'construct': 'pBAD-GFP',
  'od600': 0.76,
  'fluorescence': 8450,
  'expression_per_od': 11118.421052631578}]
```

Now we can rank the constructs.

```
sorted_measurements = sorted(
    measurements,
    key=lambda row: row["expression_per_od"],
    reverse=True,
)

for row in sorted_measurements:
    print(f"{row['construct']}: {row['expression_per_od']:.1f}")
```

```
pTac-GFP: 18804.9
pTet-GFP: 14189.9
pBAD-GFP: 11118.4
```

This is simple, but notice what we gained:

- the calculation is explicit
- the analysis is repeatable
- the transformation is inspectable
- the ranking can be reused downstream

That is the real advantage of programming in research. It is not only speed. It is **clarity and repeatability**.

4.5 Code lets you preserve reasoning

A spreadsheet typically preserves a result. A script can preserve a result **and the reasoning that produced it**.

That distinction becomes important when experiments are revised.

Suppose you later discover that one of the cultures had an OD value below your quality threshold. In code, the decision can be encoded and documented.

```
quality_threshold = 0.78

filtered = [
    row for row in measurements
    if row["od600"] >= quality_threshold
]

filtered
```

```
[{'construct': 'pTac-GFP',
  'od600': 0.82,
  'fluorescence': 15420,
  'expression_per_od': 18804.87804878049},
 {'construct': 'pTet-GFP',
  'od600': 0.79,
  'fluorescence': 11210,
  'expression_per_od': 14189.87341772152}]
```

The code is now a record of scientific judgment:

- which measurements were included
- what threshold was used
- when the rule changed
- how the analysis outcome depended on that rule

This is one reason scripting is so useful in synthetic biology. Many tasks are not merely computations. They are chains of domain decisions.

4.6 Python supports the whole design-build-test-learn cycle

Synthetic biology is often organized as a design-build-test-learn loop.

Python can contribute at every stage.

4.6.1 Design

Python can help represent parts, sequences, metadata, and circuit logic. It is useful for tasks such as:

- manipulating DNA strings
- generating combinatorial construct sets
- validating naming conventions
- preparing standardized data structures

4.6.2 Build

Python can describe and automate laboratory procedures, generate instruction files, and glue software systems together. In some labs, protocol tooling and build planning are already first-class software problems.

4.6.3 Test

This is where many beginners first meet Python. The language is excellent for:

- reading plate-reader exports
- cleaning microscopy or flow cytometry data
- normalizing replicates
- plotting time series
- summarizing experimental batches

4.6.4 Learn

Learning from experiments often means modeling, optimization, and decision support. Python supports:

- curve fitting
- probabilistic reasoning
- mechanistic simulation
- machine learning
- active-learning style workflows

The most exciting thing about this list is that it does not force you into a narrow identity. You do not need to choose between being "the wet-lab person" and "the computational person." Python makes it easier to move between those roles.

4.7 Reproducibility is a practical skill, not a slogan

Scientists often hear the word reproducibility in a moral or methodological sense, as though it were only about good intentions. But reproducibility is also a technical property.

A result is easier to reproduce when:

- the raw inputs are preserved
- the transformation steps are recorded
- the software environment is documented
- the outputs can be regenerated automatically

Python helps because scripts and notebooks can become executable records.

For example, a short script can define the sequence of operations from raw measurements to a final plot. That script can be rerun when:

- new data arrive
- a collaborator asks for clarification
- a reviewer questions a filtering choice
- you revisit the work six months later

In this book, we will treat reproducibility as part of normal working style rather than as an extra task you remember at the end.

4.8 Open source matters in synthetic biology

Synthetic biology depends heavily on shared methods, standards, and tools. Open-source software fits naturally into that culture.

Public repositories let you:

- inspect how a method is implemented
- reuse ideas rather than reinventing them
- contribute improvements back to the community
- teach from real scientific tools instead of toy examples

That is part of the spirit of this book. We will build foundational skills with small examples, but we will also connect those skills to real software ecosystems from lab and community projects.

The public organizations you shared form a useful landscape:

- **RudgeLab** offers examples around design automation, protocol automation, analysis, and simulation.
- **MyersResearchGroup** offers examples around standards-aware workflows, curation, CAD tooling, and integrations.
- **Gonza10V** provides a personal bridge between those ecosystems, showing how individual projects, experiments, and software contributions can coexist in one research trajectory.
- **DRAGGON-Lab** gives a forward-looking direction for a future lab where research outputs and software co-evolve.

Later chapters will return to those examples in more concrete detail.

4.9 What this book will ask of you

This book assumes curiosity, not prior programming expertise.

You do not need to arrive already comfortable with:

- command-line tools
- package management
- version control
- object-oriented design
- scientific computing jargon

We will build those ideas gradually.

What you *do* need is a willingness to work actively. Programming is learned by reading and running code, then changing it and seeing what breaks.

So as you work through the chapters:

1. run every code block
2. modify inputs and observe the output
3. make small mistakes on purpose
4. rewrite examples in your own words
5. connect each idea back to a real biological task

That final habit matters most. The point is not to memorize syntax. The point is to build a computational way of thinking that helps you do better biology.

4.10 A miniature design task

Here is a slightly richer example that blends sequence handling with design reasoning. Suppose we have a small set of coding sequences and want a quick first-pass screen for GC content.

```python
sequences = {
    "variant_A": "ATGAAACGTTTACGCGCTAA",
    "variant_B": "ATGCGCGCGCGTTATATATAA",
    "variant_C": "ATGAATTTCGATCGATTTAA",
}


def gc_content(seq: str) -> float:
    seq = seq.upper()
    gc = seq.count("G") + seq.count("C")
    return gc / len(seq)


for name, seq in sequences.items():
    print(f"{name}: GC={gc_content(seq):.2%}")
```

```
variant_A: GC=40.00%
variant_B: GC=42.86%
variant_C: GC=25.00%
```

This is not a full design pipeline. It is just one small diagnostic. But it illustrates an important principle:

> Biology becomes programmable when you can represent biological objects as data and biological questions as transformations on that data.

Sequences can be strings. Samples can be rows. Part libraries can be tables. Experimental rules can be functions. Model parameters can be dictionaries or data frames.

Once you see that mapping clearly, Python starts to feel less like "coding" and more like a language for expressing research logic.

4.11 How this book is organized

The first part of the book builds the foundations:

- how to read and write Python
- how to work in notebooks
- how to keep analyses reproducible

Then we move toward biological data:

- sequences
- tabular experiment outputs
- networks and graphs

Then we connect those skills to synthetic biology workflows:

- gene-expression models
- design-build-test-learn pipelines
- standards and tooling

Finally, we move into real software case studies and future-facing lab design.

The long-term goal is not only that you can write code snippets. It is that you can imagine and build tools that sit naturally inside a synthetic biology research program.

4.12 Exercises

1. In the normalized fluorescence example, add a field called `condition` for each construct and group your interpretation by condition.
2. Change the quality threshold from `0.78` to `0.80`. Which constructs remain in the filtered set?
3. Add a fourth sequence to the GC example and rank all variants from highest GC content to lowest.
4. Write a short paragraph in your own words answering this question: *Which parts of your current research already involve hidden computation, even if you do not yet use Python for them?*

4.13 Key ideas from this chapter

- Synthetic biology is already computational, even when the work feels mostly experimental.
- Python is valuable because it is readable, flexible, and connected to a large scientific ecosystem.
- Code is not only about automation. It is also about preserving reasoning and improving reproducibility.
- Biological work becomes programmable when we represent biological objects and decisions as data structures and transformations.
- Learning Python is not separate from learning better research workflows.

5. Python Basics for Biologists

In this chapter we begin writing Python directly.

Our goal is not to cover every part of the language. Our goal is to learn enough of the core ideas that you can read simple programs, write your own small analyses, and modify examples with confidence.

We will focus on a compact set of tools that reappear constantly in scientific work:

- variables
- numbers and strings
- lists and dictionaries
- loops and conditionals
- functions
- simple record-oriented data processing

Every concept will be tied to a biological example.

5.1 Running code is part of reading code

When you study programming from a book, it is tempting to read passively and tell yourself that the example "makes sense." Resist that temptation.

You should run the code, inspect the output, and then change something. Change a sequence. Change a threshold. Introduce a typo. Replace a number. Most understanding comes from that short loop between action and feedback.

5.2 Variables: naming biological quantities

A variable is a name attached to a value.

In biology, values might represent:

- a DNA sequence
- an inducer concentration
- a fluorescence measurement
- a strain name
- the number of replicates in an experiment

```
strain = "E_coli_MG1655"
plasmid = "pTac-GFP"
inducer_mM = 0.5
replicates = 3
```

These names now point to values. We can inspect them.

```
print(strain)
print(plasmid)
print(inducer_mM)
print(replicates)
```

```
E_coli_MG1655
pTac-GFP
0.5
3
```

Good variable names make scientific code easier to understand. Compare these two names:

- `x`
- `fluorescence_au`

The second name carries experimental meaning. In research code, good names are often more valuable than short names.

5.3 Numbers and arithmetic

Python works naturally with integers and decimal values.

```
colonies = 148
volume_uL = 12.5
fluorescence = 15230
od600 = 0.81

expression_per_od = fluorescence / od600
expression_per_od
```

```
18802.46913580247
```

You can combine calculations just as you would in a lab notebook.

```
dilution_factor = 100
final_concentration = inducer_mM / dilution_factor
final_concentration
```

```
0.005
```

Parentheses help make the logic explicit.

```
replicate_values = [15100, 15230, 14980]
average_expression = sum(replicate_values) / len(replicate_values)
average_expression
```

```
15103.333333333334
```

5.4 Strings: sequences and labels

Strings are pieces of text. In synthetic biology they often represent names, identifiers, or biological sequences.

```
sequence = "ATGCGTACCTGA"
promoter = "pTet"
```

You can ask for the length of a sequence.

```
len(sequence)
```

```
12
```

You can access individual characters by position. Python counts from zero.

```
sequence[0], sequence[1], sequence[2]
```

```
('A', 'T', 'G')
```

You can also take slices.

```
sequence[0:6]
```

```
'ATGCGT'
```

That slice means "start at position 0 and stop before position 6."

5.4.1 Useful string methods

A method is an operation attached to a value. Strings come with many helpful methods.

```
raw_sequence = " atgcgtacctga\n"
clean_sequence = raw_sequence.strip().upper()
clean_sequence
```

```
'ATGCGTACCTGA'
```

Two methods that appear constantly in sequence work are `.count()` and `.replace()`.

```
sequence = "ATGCGTACCTGA"
print(sequence.count("G"))
print(sequence.replace("T", "U"))
```

```
3
AUGCGUACCUGA
```

The first counts how many times `"G"` appears. The second makes a new string where every `T` is replaced with `U`, which is a crude way to represent the corresponding RNA sequence.

5.5 Writing your first function

A function packages a piece of logic so you can reuse it.

Let us write a function for GC content.

```
def gc_content(seq: str) -> float:
    seq = seq.upper()
    gc = seq.count("G") + seq.count("C")
    return gc / len(seq)
```

Now we can apply it to any sequence.

```
for seq in ["ATGC", "ATATGGCC", "GCGCGC"]:
    print(seq, round(gc_content(seq), 3))
```

```
ATGC 0.5
ATATGGCC 0.5
GCGCGC 1.0
```

Functions are one of the most important ideas in programming because they let you give a name to a scientific operation.

In the same way that a protocol gives a name to a recurring experimental procedure, a function gives a name to a recurring computational procedure.

5.5.1 Make functions safer

Real data are messy. What happens if we pass an empty string?

```
def safe_gc_content(seq: str) -> float:
    seq = seq.strip().upper()
    if len(seq) == 0:
        return 0.0
    gc = seq.count("G") + seq.count("C")
    return gc / len(seq)


print(safe_gc_content(""))
print(safe_gc_content(" aaGGcc "))
```

```
0.0
0.6666666666666666
```

This introduces our first conditional.

5.6 Conditionals: making decisions in code

Conditionals let a program do different things depending on the situation.

```python
def classify_gc(seq: str) -> str:
    gc = safe_gc_content(seq)
    if gc >= 0.60:
        return "high_gc"
    elif gc >= 0.40:
        return "medium_gc"
    else:
        return "low_gc"


for seq in ["ATATAT", "ATGCGT", "GCGCGC"]:
    print(seq, classify_gc(seq))
```

```
ATATAT low_gc
ATGCGT medium_gc
GCGCGC high_gc
```

This is a simple version of rule-based reasoning. In biology we often encode decisions in exactly this way:

- if growth is below threshold, flag the culture
- if a sequence contains a forbidden site, reject the design
- if fluorescence exceeds a target, keep the construct

The syntax may be new, but the logic should feel familiar.

5.7 Lists: working with collections

A list stores multiple values in order.

```python
constructs = ["pTac-GFP", "pTet-GFP", "pBAD-GFP"]
constructs
```

```
['pTac-GFP', 'pTet-GFP', 'pBAD-GFP']
```

You can access elements by index.

```python
constructs[0], constructs[2]
```

```
('pTac-GFP', 'pBAD-GFP')
```

You can add a new element.

```python
constructs.append("pLacI-mCherry")
constructs
```

```
['pTac-GFP', 'pTet-GFP', 'pBAD-GFP', 'pLacI-mCherry']
```

Lists are useful whenever you have a collection of related objects:

- a set of sequences
- a list of strain IDs
- replicate measurements
- candidate designs

5.7.1 Looping through a list

A `for` loop repeats an action for each item in a collection.

```
sequences = [
    "ATGAAACGTTTACGCGCTAA",
    "ATGCGCGCGCGTTATATATAA",
    "ATGAATTTCGATCGATTTAA",
]

for seq in sequences:
    print(seq, f"GC={gc_content(seq):.2%}")
```

```
ATGAAACGTTTACGCGCTAA GC=40.00%
ATGCGCGCGCGTTATATATAA GC=42.86%
ATGAATTTCGATCGATTTAA GC=25.00%
```

This pattern appears everywhere in scientific programming: “for each sample, do the same analysis.”

5.8 Building lists from loops

Sometimes we do not only want to print results. We want to store them.

```
gc_values = []

for seq in sequences:
    gc_values.append(gc_content(seq))

gc_values
```

```
[0.4, 0.42857142857142855, 0.25]
```

Now we can summarize the results.

```
min(gc_values), max(gc_values), sum(gc_values) / len(gc_values)
```

```
(0.25, 0.42857142857142855, 0.3595238095238095)
```

Python also offers a compact syntax called a list comprehension.

```
gc_values_compact = [gc_content(seq) for seq in sequences]
gc_values_compact
```

```
[0.4, 0.42857142857142855, 0.25]
```

For beginners, comprehensions may look dense at first. You do not need to force yourself to use them immediately. A regular `for` loop is often easier to read while you are learning.

5.9 Dictionaries: attaching names to values

A dictionary stores key-value pairs. This is extremely useful for biological data because experiments often combine measurements with metadata.

```
sample = {
    "construct": "pTac-GFP",
    "strain": "E_coli_MG1655",
    "od600": 0.82,
    "fluorescence": 15420,
}

sample
```

```
{'construct': 'pTac-GFP',
 'strain': 'E_coli_MG1655',
 'od600': 0.82,
 'fluorescence': 15420}
```

You can access values by key.

```
sample["construct"], sample["fluorescence"]
```

```
('pTac-GFP', 15420)
```

You can add new values.

```
sample["expression_per_od"] = sample["fluorescence"] / sample["od600"]
sample
```

```
{'construct': 'pTac-GFP',
 'strain': 'E_coli_MG1655',
 'od600': 0.82,
 'fluorescence': 15420,
 'expression_per_od': 18804.87804878049}
```

This is a very common style for small data tasks: represent one biological observation as a dictionary, then store many observations in a list.

5.10 A list of dictionaries: a tiny experimental dataset

Let us represent a miniature experiment.

```
experiment = [
    {"construct": "pTac-GFP", "condition": "glucose", "od600": 0.82, "fluorescence":
15420},
    {"construct": "pTac-GFP", "condition": "glycerol", "od600": 0.77, "fluorescence":
14110},
    {"construct": "pTet-GFP", "condition": "glucose", "od600": 0.79, "fluorescence":
11210},
    {"construct": "pTet-GFP", "condition": "glycerol", "od600": 0.74, "fluorescence":
12600},
]

experiment
```

```
[{'construct': 'pTac-GFP',
  'condition': 'glucose',
  'od600': 0.82,
  'fluorescence': 15420},
 {'construct': 'pTac-GFP',
  'condition': 'glycerol',
  'od600': 0.77,
  'fluorescence': 14110},
 {'construct': 'pTet-GFP',
  'condition': 'glucose',
  'od600': 0.79,
  'fluorescence': 11210},
 {'construct': 'pTet-GFP',
  'condition': 'glycerol',
  'od600': 0.74,
  'fluorescence': 12600}]
```

Now let us compute normalized expression for every row.

```
for row in experiment:
    row["expression_per_od"] = row["fluorescence"] / row["od600"]

experiment
```

```
[{'construct': 'pTac-GFP',
  'condition': 'glucose',
  'od600': 0.82,
  'fluorescence': 15420,
  'expression_per_od': 18804.87804878049},
 {'construct': 'pTac-GFP',
  'condition': 'glycerol',
  'od600': 0.77,
  'fluorescence': 14110,
  'expression_per_od': 18324.675324675325},
 {'construct': 'pTet-GFP',
  'condition': 'glucose',
  'od600': 0.79,
  'fluorescence': 11210,
  'expression_per_od': 14189.87341772152},
 {'construct': 'pTet-GFP',
  'condition': 'glycerol',
  'od600': 0.74,
  'fluorescence': 12600,
  'expression_per_od': 17027.027027027027}]
```

And let us print a readable summary.

```
for row in experiment:
    print(
        f"{row['construct']} in {row['condition']}: "
        f"{row['expression_per_od']:.1f} AU/OD"
    )
```

```
pTac-GFP in glucose: 18804.9 AU/OD
pTac-GFP in glycerol: 18324.7 AU/OD
pTet-GFP in glucose: 14189.9 AU/OD
pTet-GFP in glycerol: 17027.0 AU/OD
```

This is already a recognizable analysis pattern.

5.11 Filtering data with conditionals

You will often want to keep only rows that satisfy a rule.

```
qc_pass = []

for row in experiment:
    if row["od600"] >= 0.76:
        qc_pass.append(row)

qc_pass
```

```
[{'construct': 'pTac-GFP',
  'condition': 'glucose',
  'od600': 0.82,
  'fluorescence': 15420,
  'expression_per_od': 18804.87804878049},
 {'construct': 'pTac-GFP',
  'condition': 'glycerol',
```

```
  'od600': 0.77,
  'fluorescence': 14110,
  'expression_per_od': 18324.675324675325},
 {'construct': 'pTet-GFP',
  'condition': 'glucose',
  'od600': 0.79,
  'fluorescence': 11210,
  'expression_per_od': 14189.87341772152}]
```

The same logic can be written as a list comprehension.

```
qc_pass_compact = [row for row in experiment if row["od600"] >= 0.76]
qc_pass_compact
```

```
[{'construct': 'pTac-GFP',
  'condition': 'glucose',
  'od600': 0.82,
  'fluorescence': 15420,
  'expression_per_od': 18804.87804878049},
 {'construct': 'pTac-GFP',
  'condition': 'glycerol',
  'od600': 0.77,
  'fluorescence': 14110,
  'expression_per_od': 18324.675324675325},
 {'construct': 'pTet-GFP',
  'condition': 'glucose',
  'od600': 0.79,
  'fluorescence': 11210,
  'expression_per_od': 14189.87341772152}]
```

Filtering is one of the most common data-cleaning tasks in biology.

5.12 Counting things with dictionaries

Dictionaries are also useful for summarizing categories.

Suppose we have a list of annotations for a set of constructs.

```
annotations = [
    "promoter",
    "promoter",
    "cds",
    "terminator",
    "cds",
    "rbs",
    "promoter",
]

feature_counts = {}

for feature in annotations:
    if feature not in feature_counts:
        feature_counts[feature] = 0
    feature_counts[feature] += 1

feature_counts
```

```
{'promoter': 3, 'cds': 2, 'terminator': 1, 'rbs': 1}
```

Python can also do this more directly with `collections.Counter`, which we saw in the previous chapter.

```
from collections import Counter

Counter(annotations)
```

```
Counter({'promoter': 3, 'cds': 2, 'terminator': 1, 'rbs': 1})
```

The longer version is still worth studying because it teaches you how counting works step by step.

5.13 Reading simple tabular data

Eventually you will use tools like pandas for larger data tables. But it is helpful to first understand the basic structure of tabular data.

Here we will use the built-in `csv` module together with an in-memory text buffer.

```
import csv
from io import StringIO

csv_text = """sample,od600,fluorescence
A1,0.81,15230
A2,0.77,14120
B1,0.79,11340
"""

reader = csv.DictReader(StringIO(csv_text))
rows = list(reader)
rows
```

```
[{'sample': 'A1', 'od600': '0.81', 'fluorescence': '15230'},
 {'sample': 'A2', 'od600': '0.77', 'fluorescence': '14120'},
 {'sample': 'B1', 'od600': '0.79', 'fluorescence': '11340'}]
```

Notice that CSV data are read as strings by default.

```
rows[0]
```

```
{'sample': 'A1', 'od600': '0.81', 'fluorescence': '15230'}
```

So we often need to convert numeric fields.

```
for row in rows:
    row["od600"] = float(row["od600"])
    row["fluorescence"] = int(row["fluorescence"])
    row["expression_per_od"] = row["fluorescence"] / row["od600"]

rows
```

```
[{'sample': 'A1',
  'od600': 0.81,
  'fluorescence': 15230,
  'expression_per_od': 18802.46913580247},
 {'sample': 'A2',
  'od600': 0.77,
  'fluorescence': 14120,
  'expression_per_od': 18337.662337662336},
 {'sample': 'B1',
  'od600': 0.79,
  'fluorescence': 11340,
  'expression_per_od': 14354.430379746835}]
```

This example is small, but conceptually it matches what happens when you import real instrument output.

5.14 Strings plus dictionaries: reverse complements

Let us build a slightly more biological utility. A reverse complement is a classic beginner exercise because it combines strings, dictionaries, loops, and functions.

```
def reverse_complement(seq: str) -> str:
    complements = {
        "A": "T",
        "T": "A",
        "G": "C",
        "C": "G",
    }
    seq = seq.upper()
    reversed_bases = reversed(seq)
    return "".join(complements[base] for base in reversed_bases)


reverse_complement("ATGCCGTA")
```

```
'TACGGCAT'
```

Let us test it on a few sequences.

```
for seq in ["ATGC", "GGGAAA", "TTAACCGG"]:
    print(seq, "->", reverse_complement(seq))
```

```
ATGC -> GCAT
GGGAAA -> TTTCCC
TTAACCGG -> CCGGTTAA
```

This function is useful not only because reverse complements matter biologically, but because it demonstrates how to decompose a task:

1. define a mapping
2. standardize the input
3. reverse the sequence
4. translate each character
5. join the result back into a string

That is computational thinking in a very practical form.

5.15 A slightly richer example: screening constructs

Now let us combine multiple ideas into one small workflow.

We will represent several constructs, compute a few properties, and decide which ones pass a simple screen.

```
construct_library = [
    {"name": "variant_A", "sequence": "ATGAAACGTTTACGCGCTAA", "promoter": "pTac"},
    {"name": "variant_B", "sequence": "ATGCGCGCGCGTTATATATAA", "promoter": "pTet"},
    {"name": "variant_C", "sequence": "ATGAATTTCGATCGATTTAA", "promoter": "pBAD"},
]

for construct in construct_library:
    seq = construct["sequence"]
    construct["length_bp"] = len(seq)
    construct["gc_fraction"] = gc_content(seq)
    construct["gc_class"] = classify_gc(seq)

construct_library
```

```
[{'name': 'variant_A',
  'sequence': 'ATGAAACGTTTACGCGCTAA',
  'promoter': 'pTac',
  'length_bp': 20,
  'gc_fraction': 0.4,
  'gc_class': 'medium_gc'},
 {'name': 'variant_B',
  'sequence': 'ATGCGCGCGCGTTATATATAA',
  'promoter': 'pTet',
  'length_bp': 21,
  'gc_fraction': 0.42857142857142855,
  'gc_class': 'medium_gc'},
 {'name': 'variant_C',
  'sequence': 'ATGAATTTCGATCGATTTAA',
  'promoter': 'pBAD',
  'length_bp': 20,
  'gc_fraction': 0.25,
  'gc_class': 'low_gc'}]
```

Now let us define a simple selection rule.

```
selected = []

for construct in construct_library:
    if construct["length_bp"] >= 20 and construct["gc_fraction"] >= 0.40:
        selected.append(construct)

selected
```

```
[{'name': 'variant_A',
  'sequence': 'ATGAAACGTTTACGCGCTAA',
  'promoter': 'pTac',
  'length_bp': 20,
  'gc_fraction': 0.4,
  'gc_class': 'medium_gc'},
 {'name': 'variant_B',
  'sequence': 'ATGCGCGCGCGTTATATATAA',
  'promoter': 'pTet',
  'length_bp': 21,
  'gc_fraction': 0.42857142857142855,
  'gc_class': 'medium_gc'}]
```

And let us print a short report.

```
for construct in selected:
    print(
        f"{construct['name']} ({construct['promoter']}): "
        f"length={construct['length_bp']} bp, "
        f"GC={construct['gc_fraction']:.2%}, "
        f"class={construct['gc_class']}"
    )
```

```
variant_A (pTac): length=20 bp, GC=40.00%, class=medium_gc
variant_B (pTet): length=21 bp, GC=42.86%, class=medium_gc
```

This is still a toy example, but it already looks like something that could grow into a real design-screening utility.

5.16 Common beginner mistakes

You are going to make mistakes. That is normal. Here are a few that appear often.

5.16.1 Forgetting that Python starts counting at zero

```
sequence = "ATGCGT"
print(sequence[0])
print(sequence[5])
```

```
A
T
```

There is no element at position 6 because indexing starts at 0.

5.16.2 Mixing strings and numbers

```
value_text = "10"
value_number = 10

print(value_text)
print(value_number)
print(int(value_text) + value_number)
```

```
10
10
20
```

Data from files often arrive as text and must be converted before calculation.

5.16.3 Using = when you mean comparison

- `=` assigns a value
- `==` checks whether two values are equal

```
condition = "induced"
print(condition == "induced")
```

```
True
```

5.16.4 Forgetting to return a value from a function

```
def promoter_label(name: str) -> str:
    if name.startswith("p"):
        return "looks_like_promoter_name"
    return "other"


promoter_label("pTac")
```

```
'looks_like_promoter_name'
```

When a function should produce a result, `return` is what sends that result back.

5.17 Style matters because science is collaborative

As your scripts become more useful, other people will read them. Clear style makes that easier.

A few practical habits:

- use descriptive variable names
- keep functions short when possible
- write code in small testable steps

- avoid copying and pasting the same logic many times
- add a short comment when the scientific reasoning is not obvious from the code alone

Here is an example of a helpful scientific comment.

```
# Normalize fluorescence by OD600 so cultures with different densities
# are easier to compare on a per-biomass basis.
normalized_expression = fluorescence / od600
normalized_expression
```

```
18802.46913580247
```

A comment should explain *why* when the why is not already obvious.

5.18 Practice: modify the code, do not only read it

Before moving on, try changing the examples in concrete ways.

- Add another construct to `construct_library`.
- Change the GC classification thresholds.
- Add a new field called `replicates` to each experimental row.
- Rewrite `reverse_complement()` so it handles lowercase input without calling `.upper()`.
- Change the selection rule so that constructs with `promoter == "pBAD"` always pass.

These small changes are where fluency begins.

5.19 Exercises

1. Write a function called `at_content()` that returns the fraction of `A` and `T` bases in a sequence.
2. Given a list of fluorescence values, compute the mean using `sum()` and `len()`.
3. Create a list of dictionaries representing three strains with fields `name`, `growth_rate`, and `passed_qc`. Print only the names of strains that passed QC.
4. Modify `reverse_complement()` so that it raises an informative error if the sequence contains a character other than `A`, `T`, `G`, or `C`.
5. Represent a tiny promoter library as a dictionary mapping promoter names to strengths, and print the strongest promoter.

5.20 Key ideas from this chapter

- Variables give names to biological quantities and measurements.
- Strings are useful for labels and sequences.
- Lists hold collections, and dictionaries attach names to values.
- Loops let you apply the same logic across many biological objects.
- Conditionals let you encode scientific decision rules.
- Functions package reusable analysis steps.
- Even simple Python structures are enough to represent meaningful biological workflows.

6. Jupyter, Files, and Reproducibility

By the end of the previous chapter, you can already write small Python programs.

That is an important milestone, but it is not enough for real scientific work.

In research, code lives inside a larger workflow. You write in a notebook, save files, clean data, rerun analyses, revise plots, and send results to collaborators. A script that works once on your laptop is useful. A workflow that another person can rerun next month is much more useful.

This chapter is about that second level of practice.

We will introduce four ideas that belong together:

- **Jupyter notebooks** for exploration and explanation
- **files and folders** as the raw material of analysis
- **environments** for keeping software dependencies under control
- **reproducibility** as a habit of working, not only a technical trick

These topics may look less glamorous than modeling circuits or designing constructs, but they make the difference between fragile code and trustworthy research.

6.1 Code in science has more than one form

A beginner often imagines that programming means writing one kind of thing: a program.

In practice, scientific Python work usually appears in three complementary forms.

6.1.1 Scripts

A script is a file that runs a sequence of steps from top to bottom.

Scripts are good when you want to:

- clean raw data the same way every time
- rename or reorganize files
- process a batch of sequences
- turn a manual analysis into a repeatable workflow

Scripts are especially useful once a task stops being exploratory and starts becoming routine.

6.1.2 Notebooks

A notebook mixes code, output, equations, and prose. It is ideal for:

- trying ideas quickly
- inspecting intermediate results
- teaching or documenting a workflow
- producing a shareable computational narrative

That is why notebooks became so popular in biology, data science, and quantitative research. They let you think with code in public.

6.1.3 Rendered documents

A rendered document, such as a Quarto chapter or report, takes reproducibility one step further. Instead of a notebook that only runs interactively, you can create a document that executes code and produces a polished output in HTML or PDF.

That is one reason Quarto is a good fit for this book. It encourages a workflow in which explanation and computation live together.

A healthy research project often uses all three:

- notebooks for exploration

- scripts for reusable tasks
- rendered reports or book chapters for communication

6.2 Why notebooks became so influential

Jupyter notebooks are not just popular because they are convenient. They match the real rhythm of experimental reasoning.

When you are characterizing a promoter library, you rarely know the entire analysis in advance. You want to:

1. load the data
2. inspect a few rows
3. notice a suspicious value
4. clean the data
5. recompute a summary
6. try a different normalization
7. make a quick plot
8. write down what you learned

That loop of *inspect, modify, rerun, interpret* is exactly what notebooks are good at.

They are especially valuable for beginners because they reduce the distance between writing code and seeing what it does.

6.3 Your computational environment matters

When scientists say, “the code works on my machine,” they often mean something narrower than they realize. The code works with:

- a particular Python version
- a particular set of installed packages
- a particular folder layout
- a particular set of input files
- a particular order of execution

Reproducibility means making those assumptions visible and manageable.

Let us start by asking Python about itself.

```
import platform
import sys
from pathlib import Path
import tempfile

chapter3_demo_dir = Path(tempfile.mkdtemp(prefix="synbio_ch3_"))

{
    "python_executable": sys.executable,
    "python_version": sys.version.split()[0],
    "platform": platform.platform(),
    "demo_directory": str(chapter3_demo_dir),
}
```

```
{'python_executable': '/opt/hostedtoolcache/Python/3.11.15/x64/bin/python3',
 'python_version': '3.11.15',
 'platform': 'Linux-6.17.0-1010-azure-x86_64-with-glibc2.39',
 'demo_directory': '/tmp/synbio_ch3_k7q3bntn'}
```

This kind of information is boring until something breaks. Then it becomes extremely valuable.

If a collaborator cannot run your notebook, one of the first questions is: *Are we even using the same Python environment?*

6.4 Virtual environments: keeping projects separate

A virtual environment is an isolated Python installation for a specific project.

Why bother with that?

Because scientific projects accumulate dependencies. One analysis may need a recent version of `pandas`. Another may depend on an older version of a modeling package. If everything is installed globally, projects start interfering with each other.

A virtual environment gives each project its own small software world.

A typical setup looks like this:

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install jupyter
```

On Windows PowerShell, the activation command is different:

```
py -m venv .venv
.venv\Scripts\Activate.ps1
py -m pip install jupyter
```

The point is not the exact command syntax. The point is the habit:

- create an environment per project
- activate it before you work
- install dependencies into that environment
- record those dependencies when the project matures

That habit prevents a surprising amount of pain.

6.5 A reproducible project has a shape

Reproducibility is easier when your work lives in a predictable folder structure.

A small research project might look like this:

```
project/
├── data/
│   ├── raw/
│   └── processed/
├── notebooks/
├── scripts/
├── results/
├── figures/
└── README.md
```

You do not need to be rigid about this exact layout. What matters is the principle:

- **raw data** should stay separate from modified data
- **scripts** should be kept under version control
- **results** should be regenerable from code
- **README files** should explain what the project is and how to run it

A folder structure is not just organizational. It is a model of how work flows through the project.

6.6 Paths are part of scientific thinking

Beginners often treat file paths as annoying details. In reality, paths are how your code locates the world.

Python's `pathlib` module makes path handling much clearer than manual string concatenation.

```
raw_dir = chapter3_demo_dir / "data" / "raw"
processed_dir = chapter3_demo_dir / "data" / "processed"
results_dir = chapter3_demo_dir / "results"

for directory in [raw_dir, processed_dir, results_dir]:
```

```
    directory.mkdir(parents=True, exist_ok=True)

sorted(str(path.relative_to(chapter3_demo_dir))          for          path          in
chapter3_demo_dir.iterdir())
```

```
['data', 'results']
```

The / operator in `pathlib` joins path components in a readable way. This is much safer than manually building strings like `"data/raw/file.csv"`, especially if you want your code to work across operating systems.

Let us create a file path for a plate-reader export.

```
plate_reader_file = raw_dir / "plate_reader_day1.csv"
plate_reader_file
```

```
PosixPath('/tmp/synbio_ch3_k7q3bntn/data/raw/plate_reader_day1.csv')
```

That object is not just text. It is a `Path`, which means Python can use it directly for reading, writing, testing existence, and more.

6.7 Writing a small dataset to disk

To understand reproducibility, it helps to work with real files rather than only in-memory objects.

Here we will create a tiny synthetic dataset that resembles growth and fluorescence measurements from a characterization experiment.

```
import csv

rows_to_write = [
    {"sample": "A1", "construct": "pTac-GFP", "condition": "glucose",  "od600": 0.81,
"fluorescence": 15230},
    {"sample": "A2", "construct": "pTac-GFP", "condition": "glycerol", "od600": 0.77,
"fluorescence": 14120},
    {"sample": "B1", "construct": "pTet-GFP", "condition": "glucose",  "od600": 0.79,
"fluorescence": 11340},
    {"sample": "B2", "construct": "pTet-GFP", "condition": "glycerol", "od600": 0.74,
"fluorescence": 12600},
]

with plate_reader_file.open("w", newline="") as handle:
    writer = csv.DictWriter(
        handle,
        fieldnames=["sample", "construct", "condition", "od600", "fluorescence"],
    )
    writer.writeheader()
    writer.writerows(rows_to_write)

plate_reader_file.exists(), plate_reader_file.stat().st_size
```

```
(True, 177)
```

Now the dataset lives in a file, not only in the notebook state.

That distinction matters. A notebook variable disappears when the kernel restarts. A file can be rerun, versioned, shared, and inspected independently.

6.8 Reading data back in

A reproducible workflow should be able to reconstruct its analysis from saved inputs.

```
with plate_reader_file.open() as handle:
    reader = csv.DictReader(handle)
    measurements = list(reader)

measurements[:2]
```

```
[{'sample': 'A1',
  'construct': 'pTac-GFP',
  'condition': 'glucose',
  'od600': '0.81',
  'fluorescence': '15230'},
 {'sample': 'A2',
  'construct': 'pTac-GFP',
  'condition': 'glycerol',
  'od600': '0.77',
  'fluorescence': '14120'}]
```

Notice the same issue we saw in the previous chapter: CSV values come in as strings.
That means data cleaning begins immediately.

```
for row in measurements:
    row["od600"] = float(row["od600"])
    row["fluorescence"] = int(row["fluorescence"])
    row["expression_per_od"] = row["fluorescence"] / row["od600"]

measurements
```

```
[{'sample': 'A1',
  'construct': 'pTac-GFP',
  'condition': 'glucose',
  'od600': 0.81,
  'fluorescence': 15230,
  'expression_per_od': 18802.46913580247},
 {'sample': 'A2',
  'construct': 'pTac-GFP',
  'condition': 'glycerol',
  'od600': 0.77,
  'fluorescence': 14120,
  'expression_per_od': 18337.662337662336},
 {'sample': 'B1',
  'construct': 'pTet-GFP',
  'condition': 'glucose',
  'od600': 0.79,
  'fluorescence': 11340,
  'expression_per_od': 14354.430379746835},
 {'sample': 'B2',
  'construct': 'pTet-GFP',
  'condition': 'glycerol',
  'od600': 0.74,
  'fluorescence': 12600,
  'expression_per_od': 17027.027027027027}]
```

Already we can see a core pattern of computational biology:

1. load raw data
2. standardize types
3. derive new quantities
4. save or report the result

6.9 Notebook state is helpful and dangerous

A notebook remembers what you have already run.

That is incredibly helpful during exploration, but it also creates one of the most common sources of confusion for beginners: **hidden state**.

Suppose you define a threshold.

```
qc_threshold = 0.76
[row["sample"] for row in measurements if row["od600"] >= qc_threshold]
```

```
['A1', 'A2', 'B1']
```

Now imagine that, twenty minutes later, you redefine that threshold in another cell.

```
qc_threshold = 0.80
[row["sample"] for row in measurements if row["od600"] >= qc_threshold]
```

```
['A1']
```

Nothing about the data changed. Only the notebook state changed.

This is one reason people get different answers from the "same notebook." They are not always running the same sequence of cells.

Two habits reduce this problem dramatically:

- restart the kernel and run all cells from top to bottom
- keep important parameters near the top of the notebook or report

A notebook becomes much more trustworthy when it can be executed cleanly from a fresh start.

6.10 Small functions make notebooks stronger

A notebook should not become a wall of ad hoc code. Even in exploratory work, small functions help isolate logic and reduce mistakes.

Let us define a reusable normalization function.

```
def normalize_expression(row: dict) -> float:
    if row["od600"] <= 0:
        raise ValueError("OD600 must be positive for normalization")
    return row["fluorescence"] / row["od600"]


[round(normalize_expression(row), 1) for row in measurements]
```

```
[18802.5, 18337.7, 14354.4, 17027.0]
```

And let us use it to build a cleaner processed dataset.

```
processed_measurements = []

for row in measurements:
    processed_measurements.append(
        {
            "sample": row["sample"],
            "construct": row["construct"],
            "condition": row["condition"],
            "od600": row["od600"],
            "fluorescence": row["fluorescence"],
            "expression_per_od": round(normalize_expression(row), 2),
            "passed_qc": row["od600"] >= 0.76,
        }
    )
```

```
processed_measurements
```

```
[{'sample': 'A1',
  'construct': 'pTac-GFP',
  'condition': 'glucose',
  'od600': 0.81,
  'fluorescence': 15230,
  'expression_per_od': 18802.47,
  'passed_qc': True},
 {'sample': 'A2',
  'construct': 'pTac-GFP',
  'condition': 'glycerol',
  'od600': 0.77,
  'fluorescence': 14120,
  'expression_per_od': 18337.66,
  'passed_qc': True},
 {'sample': 'B1',
  'construct': 'pTet-GFP',
  'condition': 'glucose',
  'od600': 0.79,
  'fluorescence': 11340,
  'expression_per_od': 14354.43,
  'passed_qc': True},
 {'sample': 'B2',
  'construct': 'pTet-GFP',
  'condition': 'glycerol',
  'od600': 0.74,
  'fluorescence': 12600,
  'expression_per_od': 17027.03,
  'passed_qc': False}]
```

This is more reproducible than manually editing columns in a spreadsheet, because the transformation is explicit and rerunnable.

6.11 Saving processed data

A project is easier to debug when you save important intermediate results.

Let us write the processed dataset to a new file.

```
processed_file = processed_dir / "plate_reader_day1_processed.csv"

with processed_file.open("w", newline="") as handle:
    writer = csv.DictWriter(
        handle,
        fieldnames=[
            "sample",
            "construct",
            "condition",
            "od600",
            "fluorescence",
            "expression_per_od",
            "passed_qc",
        ],
    )
    writer.writeheader()
    writer.writerows(processed_measurements)

processed_file.exists(), processed_file.name
```

```
(True, 'plate_reader_day1_processed.csv')
```

And we can confirm its contents.

```
with processed_file.open() as handle:
    print(handle.read())
```

```
sample,construct,condition,od600,fluorescence,expression_per_od,passed_qc
A1,pTac-GFP,glucose,0.81,15230,18802.47,True
A2,pTac-GFP,glycerol,0.77,14120,18337.66,True
B1,pTet-GFP,glucose,0.79,11340,14354.43,True
B2,pTet-GFP,glycerol,0.74,12600,17027.03,False
```

This is a good habit for larger workflows too. Raw data should remain intact, while processed data should be clearly labeled as derived.

6.12 Saving metadata alongside results

Reproducibility is not only about data values. It is also about context.
For example, if you save processed results, you may also want to save:

- when the analysis was run
- which Python version was used
- which QC threshold was applied
- which input file was analyzed

JSON is a convenient format for lightweight metadata.

```
import json
from datetime import datetime, timezone

metadata = {
    "created_at_utc": datetime.now(timezone.utc).isoformat(),
    "python_version": sys.version.split()[0],
    "input_file": str(plate_reader_file),
    "output_file": str(processed_file),
    "qc_threshold": 0.76,
    "n_rows": len(processed_measurements),
}

metadata_file = results_dir / "analysis_metadata.json"
metadata_file.write_text(json.dumps(metadata, indent=2))

print(metadata_file.read_text())
```

```
{
  "created_at_utc": "2026-04-17T17:35:04.249898+00:00",
  "python_version": "3.11.15",
  "input_file": "/tmp/synbio_ch3_k7q3bntn/data/raw/plate_reader_day1.csv",
                  "output_file":          "/tmp/synbio_ch3_k7q3bntn/data/processed/
plate_reader_day1_processed.csv",
  "qc_threshold": 0.76,
  "n_rows": 4
}
```

A month later, that metadata can answer questions you will no longer remember reliably.

6.13 Summaries should be reproducible too

A common mistake is to make processed data reproducible but leave final summaries manual.
Instead, summaries should also come from code.

```python
def mean(values):
    return sum(values) / len(values)

summary_by_construct = {}

for row in processed_measurements:
    construct = row["construct"]
    summary_by_construct.setdefault(construct, []).append(row["expression_per_od"])

summary_table = []
for construct, values in summary_by_construct.items():
    summary_table.append(
        {
            "construct": construct,
            "mean_expression_per_od": round(mean(values), 2),
            "n_measurements": len(values),
        }
    )

summary_table
```

```
[{'construct': 'pTac-GFP',
  'mean_expression_per_od': 18570.07,
  'n_measurements': 2},
 {'construct': 'pTet-GFP',
  'mean_expression_per_od': 15690.73,
  'n_measurements': 2}]
```

Now let us save that summary as well.

```python
summary_file = results_dir / "summary_by_construct.csv"

with summary_file.open("w", newline="") as handle:
    writer = csv.DictWriter(
        handle,
        fieldnames=["construct", "mean_expression_per_od", "n_measurements"],
    )
    writer.writeheader()
    writer.writerows(summary_table)

print(summary_file.read_text())
```

```
construct,mean_expression_per_od,n_measurements
pTac-GFP,18570.07,2
pTet-GFP,15690.73,2
```

At this point we have a tiny but genuine workflow:

- raw data file
- processing step
- QC rule
- processed output
- summary output
- metadata file

That is the beginning of a reproducible analysis pipeline.

6.14 Randomness should be controlled when it matters

Biology contains real randomness, and computational biology often uses simulated randomness too.

If you use randomness in code, reproducibility may require fixing a seed.

```
import random

random.seed(7)
[random.randint(80, 120) for _ in range(5)]
```

```
[100, 89, 105, 83, 84]
```

If we reset the seed and run the same code again, we get the same result.

```
random.seed(7)
[random.randint(80, 120) for _ in range(5)]
```

```
[100, 89, 105, 83, 84]
```

This does not remove randomness from the concept. It simply makes the computational sequence reproducible.

That can be important for:

- sampling procedures
- simulation studies
- train/test splits in machine learning
- randomized search or optimization

6.15 Reproducibility is social as well as technical

It is tempting to frame reproducibility as a purely software issue: use environments, save files, set seeds, and the problem is solved.

But reproducibility is also social.

A reproducible project is easier to hand off to:

- a new student joining the lab
- a collaborator at another institution
- your future self after six months
- a reviewer asking how a result was generated

This means good research code usually includes not only code, but also explanation.

A small `README.md` that answers the following questions is often worth far more than another clever function:

- What does this project do?
- Where are the inputs?
- How do I run the analysis?
- What files are generated?
- Which environment should I use?

In other words, reproducibility is a communication practice.

6.16 Notebooks versus scripts: when to use each

This is not a battle where one tool must win.

A practical rule is:

- use a **notebook** when you are exploring or teaching
- use a **script** when a process should run the same way every time
- use a **rendered document** when you want reproducible explanation plus output

Often the best workflow is sequential:

1. explore in a notebook
2. notice which steps have stabilized
3. move stable logic into functions or scripts
4. call those functions from a notebook or Quarto report

That pattern keeps notebooks flexible without letting them become chaotic.

6.17 How Quarto fits into this workflow

This book is written in Quarto rather than in a plain notebook interface.

That choice reflects an important idea: scientific code should often be both executable and readable.

Quarto lets you:

- write prose around the code
- execute code when rendering
- include outputs directly in the final document
- treat the chapter itself as part of the reproducible workflow

In that sense, a Quarto chapter is not only a teaching document. It is also a compact example of literate scientific programming.

6.18 A full miniature workflow in one view

Let us collect the main generated files so we can see what our demo project produced.

```
sorted(str(path.relative_to(chapter3_demo_dir)) for path in
chapter3_demo_dir.rglob("*") if path.is_file())
```

```
['data/processed/plate_reader_day1_processed.csv',
 'data/raw/plate_reader_day1.csv',
 'results/analysis_metadata.json',
 'results/summary_by_construct.csv']
```

That file list is small, but it encodes a real computational story.

- We created a raw measurement file.
- We read and cleaned it.
- We derived normalized quantities.
- We saved processed data.
- We saved summary results.
- We saved metadata about the analysis itself.

That is already much closer to real scientific computing than a few isolated code fragments.

6.19 Common beginner mistakes in reproducible work

6.19.1 Hard-coding machine-specific paths

This is fragile:

```
file_path = "/Users/your_name/Desktop/final_real_data_NEW.csv"
```

Machine-specific paths break immediately for collaborators and often break for you later.

Prefer project-relative paths built with `pathlib`.

6.19.2 Editing raw data in place

If the only copy of a raw file has been manually modified, you have already lost part of the provenance of the analysis.

Keep raw data separate from processed outputs.

6.19.3 Forgetting which cell created a result

Notebook state can hide missing steps. Restarting and running all cells is one of the simplest and most effective integrity checks.

6.19.4 Installing packages globally without tracking them

That makes it hard to recreate the analysis environment later. Use project-specific environments whenever possible.

6.19.5 Mixing exploration with final reporting carelessly

Exploration is messy, and that is normal. But final results should come from a workflow that can be rerun deliberately.

6.20 Practice ideas

Try modifying the workflow in concrete ways.

- Add a third construct and rerun the summary.
- Change the QC threshold and regenerate the processed file.
- Add a `replicate` column to the raw dataset.
- Save a second metadata field recording the research question.
- Write a function that filters out rows failing QC before computing the summary.

Each of these is small, but each pushes you toward more deliberate analysis.

6.21 Exercises

1. Create a new temporary project directory with subfolders `data/raw`, `data/processed`, and `results` using `pathlib`.
2. Write a CSV file containing three sequence records with columns `name` and `sequence`, then read it back into Python.
3. Add a new column called `gc_fraction` to the sequence records and save the processed table.
4. Save a JSON metadata file containing the input file path, output file path, and Python version.
5. Simulate ten random fluorescence values with a fixed seed and compute their mean.
6. Explain in your own words why restarting and running all cells is an important notebook habit.

6.22 Key ideas from this chapter

- Scientific Python work usually combines notebooks, scripts, and rendered reports.
- Reproducibility depends on environments, folder structure, files, and execution order.
- `pathlib` makes file handling clearer and safer.
- Raw data, processed data, summaries, and metadata should be separated intentionally.
- Notebook state is powerful, but hidden state can produce misleading results.
- Small functions and saved outputs make analysis workflows easier to trust and share.
- Reproducibility is not only technical; it is also a way of communicating research clearly.

II

Part II - Biological Data in Python

7.14 Screening designs against simple rules 64
7.15 Ranking candidate sequences 64
7.16 K-mers: treating sequences as overlapping words 65
7.17 Sequences as records, not just strings 65
7.18 A mini workflow: from FASTA to a design report 66
7.19 When to stop writing your own sequence utilities 67
7.20 Biopython: from strings to standardized sequence records 67
7.21 `Seq`: a biological sequence object 68
7.22 `SeqRecord`: a sequence plus metadata 68
7.23 Reading FASTA with `SeqIO` 69
7.24 GenBank: richer records with annotations and features 69
7.25 Standardizing your own records 71
7.26 BLAST with Biopython 71
7.27 Why this matters in synthetic biology 73
7.28 Exercises 74
7.29 Recap 74

8 Tabular Experimental Data 75
8.1 Why tidy data matters 75
8.2 A wide table is often the first thing you get . . . 75
8.3 Reshaping wide data into tidy data 76
8.4 Building a tidy experiment table 77
8.5 Inspecting a DataFrame 78
8.6 Selecting columns 79
8.7 Filtering rows 80
8.8 Adding derived columns 81
8.9 Grouping and summarizing replicates 81
8.10 Quantifying induction 82
8.11 Missing values happen 83
8.12 Reading tidy data from a CSV file 84
8.13 Merging measurement tables with metadata . . . 85
8.14 Selecting one table shape for downstream work . 86
8.15 Saving processed data 86
8.16 A small end-to-end example 87
8.17 Recap 87
8.18 Exercises 87

9 Networks, Circuits, and Graphs 89
9.1 Graphs as tidy data 89
9.2 Installing NetworkX 90
9.3 A first regulatory network 90
9.4 Building a directed graph from a tidy edge table . 91
9.5 Nodes, edges, predecessors, and successors 91
9.6 Using a node table for metadata 92
9.7 Drawing a small circuit 93
9.8 Keeping edge metadata tidy 94
9.9 Finding paths through a circuit 94
9.10 Feed-forward logic 95
9.11 Detecting feedback loops 95
9.12 When an adjacency matrix is useful 96
9.13 Converting a graph back into a tidy edge table . 96
9.14 Assembly dependencies as a directed acyclic graph 97
9.15 Detecting impossible workflows 98
9.16 Merging tidy metadata with graph results 98
9.17 Choosing between a table and a graph 99
9.18 Exercises 99
9.19 Recap 100

7. Sequences as Data

Synthetic biology begins with physical molecules, but computational work usually begins with **representations**.

A DNA construct in the freezer is a physical object. A GenBank file, a FASTA entry, or a short Python string is a representation of that object. The representation is not the biology itself, but it is what our software can inspect, transform, compare, and store.

That is why sequence handling is one of the first real computational skills in synthetic biology. If you can represent sequences cleanly in Python, you can begin to automate many common tasks:

- checking whether a design contains a forbidden restriction site
- computing GC content
- finding open reading frames
- converting DNA to RNA
- translating coding sequences into protein sequences
- scanning a small library of candidate parts
- reading sequence files and summarizing them automatically

This chapter is about learning to think of sequences as **data structures** rather than only as strings typed into a document.

We will stay deliberately concrete. By the end of the chapter, you will be able to write small, readable Python programs that manipulate DNA sequences in the way a working synthetic biology project often requires.

7.1 Sequences look like text, but they are structured text

At first glance, a DNA sequence looks like ordinary text.

```
sequence = "ATGCGTACCGTTAG"
sequence
```

```
'ATGCGTACCGTTAG'
```

Python stores that value as a string. That is convenient, because strings already support many operations that are useful for biology.

But a biological sequence is not arbitrary text. Each character has meaning, and the order matters. In DNA,

- `A`, `T`, `G`, and `C` represent nucleotides
- triplets can encode codons
- orientation matters
- motifs can signal biological function
- subsequences can be treated as parts, features, or domains

So even though Python begins with a generic string, our job is to build **biological meaning** on top of it.

7.2 Indexing and slicing sequences

Because sequences are ordered, indexing and slicing are immediately useful.

```
sequence = "ATGCGTACCGTTAG"

first_base = sequence[0]
start_codon = sequence[0:3]
middle_region = sequence[3:9]
last_three = sequence[-3:]
```

```
first_base, start_codon, middle_region, last_three
```

```
('A', 'ATG', 'CGTACC', 'TAG')
```

This matters because many biological questions are positional.
You may want to ask:

- does the sequence begin with `ATG`?
- what are bases 10 through 20?
- what is the last codon?
- what is the promoter region upstream of a coding sequence?

A slice gives you a direct answer.

```
sequence.startswith("ATG"), sequence.endswith("TAG")
```

```
(True, True)
```

The two methods above are simple, but they already capture a common form of sequence quality control.

7.3 Length is a biological property

The length of a sequence is often the first thing you inspect.

```
len(sequence)
```

```
14
```

Length affects many later decisions:

- Is a coding sequence divisible by 3?
- Is a promoter variant the expected size?
- Did an assembly likely produce an insertion or deletion?
- Are multiple parts aligned to the same design template?

A short function makes that check reusable.

```
def is_coding_sequence_length(seq: str) -> bool:
    return len(seq) % 3 == 0


is_coding_sequence_length("ATGAAATGA"), is_coding_sequence_length("ATGAAATG")
```

```
(True, False)
```

That function says nothing about whether the sequence is biologically valid as a protein-coding region. It only checks whether the length is compatible with codons. That distinction matters. Good scientific code often separates a large biological question into smaller computational checks.

7.4 Counting bases

A sequence is also a composition. We often want to know how many times each base appears.

```
from collections import Counter

sequence = "ATGCGTACCGTTAG"
base_counts = Counter(sequence)
base_counts
```

```
Counter({'T': 4, 'G': 4, 'A': 3, 'C': 3})
```

From there, we can compute GC content.

```
def gc_content(seq: str) -> float:
    seq = seq.upper()
    if len(seq) == 0:
        return 0.0
    gc = seq.count("G") + seq.count("C")
    return gc / len(seq)


gc_content(sequence)
```

```
0.5
```

To make the result easier to read, we often show it as a percentage.

```
print(f"GC content: {gc_content(sequence):.1%}")
```

```
GC content: 50.0%
```

GC content is one of those simple measurements that appears everywhere in synthetic biology and bioinformatics:

- checking whether a construct is unusually GC-rich or AT-rich
- comparing designs from different organisms
- screening codon-optimized variants
- looking for problematic local sequence composition

7.5 Cleaning and validating sequence input

Real input is messy. A sequence copied from a spreadsheet or a file may contain lowercase letters, spaces, line breaks, or unexpected symbols.

```
raw_sequence = " atgcgtaccgttag\n"
clean_sequence = raw_sequence.strip().upper()
clean_sequence
```

```
'ATGCGTACCGTTAG'
```

That is a start, but in biology we also care whether every character belongs to the expected alphabet.

```
def clean_dna(seq: str) -> str:
    return "".join(seq.split()).upper()


def is_valid_dna(seq: str) -> bool:
    allowed = set("ATGC")
    cleaned = clean_dna(seq)
    return all(base in allowed for base in cleaned)


is_valid_dna("ATG CCG\nTTA"), is_valid_dna("ATGBCC")
```

```
(True, False)
```

A more practical function should both clean and validate, and raise an error when something is wrong.

```
def require_valid_dna(seq: str) -> str:
    cleaned = clean_dna(seq)
    invalid_bases = sorted(set(cleaned) - set("ATGC"))
    if invalid_bases:
```

```
        raise ValueError(f"Invalid DNA symbols: {invalid_bases}")
    return cleaned


require_valid_dna("atg ccg tta")
```

```
'ATGCCGTTA'
```

Now the function is doing real work for us:

- removing irrelevant formatting
- converting case consistently
- refusing to continue when the input does not match our assumptions

This is an important habit. Scientific code should fail clearly when its assumptions are violated.

7.6 Complement and reverse complement

Many beginner sequence tasks become possible as soon as you can compute a reverse complement.

In double-stranded DNA,

- `A` pairs with `T`
- `G` pairs with `C`

Python's string translation tools make this neat and readable.

```
complement_map = str.maketrans("ATGC", "TACG")


def complement(seq: str) -> str:
    seq = require_valid_dna(seq)
    return seq.translate(complement_map)


def reverse_complement(seq: str) -> str:
    seq = require_valid_dna(seq)
    return seq.translate(complement_map)[::-1]


sequence = "ATGCCGTA"
complement(sequence), reverse_complement(sequence)
```

```
('TACGGCAT', 'TACGGCAT')
```

The expression `[::-1]` reverses a string. Combined with complementing the bases, it gives the reverse complement.

That operation matters constantly in cloning and design work. For example, primers are often reasoned about in terms of reverse complements, and many motif searches need to consider both strands.

7.7 Searching for motifs

A sequence is rarely only a sequence. It is also a container for motifs.

A motif could be:

- a restriction site
- a ribosome binding site
- a start codon
- a terminator-like pattern
- a barcode
- a homopolymer stretch

Python strings already support motif search.

```
sequence = "ATGGAATTCGCCGTTAA"
sequence.find("GAATTC")
```

```
3
```

The return value is the starting index of the first match. If the motif is absent, Python returns `-1`.

```
sequence.find("GGATCC")
```

```
-1
```

For many lab tasks, a simple yes-or-no answer is enough.

```
def contains_site(seq: str, site: str) -> bool:
    seq = require_valid_dna(seq)
    site = require_valid_dna(site)
    return site in seq


contains_site(sequence, "GAATTC"), contains_site(sequence, "GGATCC")
```

```
(True, False)
```

Here is a small scanner for multiple common restriction sites.

```
restriction_sites = {
    "EcoRI": "GAATTC",
    "BamHI": "GGATCC",
    "BsaI": "GGTCTC",
    "NotI": "GCGGCCGC",
}


def find_present_sites(seq: str, site_map: dict[str, str]) -> list[str]:
    seq = require_valid_dna(seq)
    present = []
    for enzyme, site in site_map.items():
        if site in seq:
            present.append(enzyme)
    return present


find_present_sites("ATGGAATTCGGTCTCTAA", restriction_sites)
```

```
['EcoRI', 'BsaI']
```

This kind of function is small, but it already resembles real screening logic used in design workflows.

7.8 DNA to RNA

Once a sequence is validated, transcription is conceptually simple: replace `T` with `U`.

```
def transcribe_dna(seq: str) -> str:
    seq = require_valid_dna(seq)
    return seq.replace("T", "U")


coding_dna = "ATGGCCATTGTAATGGGCCGCTGA"
transcribe_dna(coding_dna)
```

```
'AUGGCCAUUGUAAUGGGCCGCUGA'
```

This function is not modeling transcriptional regulation or strand choice. It is only converting a coding-style DNA sequence into an RNA-style sequence representation.

That may sound modest, but that is often enough for downstream analysis, teaching, and simple tooling.

7.9 Translation: codons to amino acids

Translation is one of the first places where a sequence starts to feel like more than text.

Instead of interpreting one character at a time, we interpret the sequence in triplets.

Below is a standard codon table expressed as a Python dictionary.

```python
CODON_TABLE = {
    "TTT": "F", "TTC": "F", "TTA": "L", "TTG": "L",
    "TCT": "S", "TCC": "S", "TCA": "S", "TCG": "S",
    "TAT": "Y", "TAC": "Y", "TAA": "*", "TAG": "*",
    "TGT": "C", "TGC": "C", "TGA": "*", "TGG": "W",
    "CTT": "L", "CTC": "L", "CTA": "L", "CTG": "L",
    "CCT": "P", "CCC": "P", "CCA": "P", "CCG": "P",
    "CAT": "H", "CAC": "H", "CAA": "Q", "CAG": "Q",
    "CGT": "R", "CGC": "R", "CGA": "R", "CGG": "R",
    "ATT": "I", "ATC": "I", "ATA": "I", "ATG": "M",
    "ACT": "T", "ACC": "T", "ACA": "T", "ACG": "T",
    "AAT": "N", "AAC": "N", "AAA": "K", "AAG": "K",
    "AGT": "S", "AGC": "S", "AGA": "R", "AGG": "R",
    "GTT": "V", "GTC": "V", "GTA": "V", "GTG": "V",
    "GCT": "A", "GCC": "A", "GCA": "A", "GCG": "A",
    "GAT": "D", "GAC": "D", "GAA": "E", "GAG": "E",
    "GGT": "G", "GGC": "G", "GGA": "G", "GGG": "G",
}


def translate_dna(seq: str, stop_at_stop: bool = False) -> str:
    seq = require_valid_dna(seq)
    if len(seq) % 3 != 0:
        raise ValueError("Sequence length must be divisible by 3 for translation.")

    protein = []
    for i in range(0, len(seq), 3):
        codon = seq[i:i + 3]
        amino_acid = CODON_TABLE[codon]
        if stop_at_stop and amino_acid == "*":
            break
        protein.append(amino_acid)
    return "".join(protein)


translate_dna(coding_dna)
```

```
'MAIVMGR*'
```

The stop codon is represented here by *, which is common in simple protein sequence displays.

If we want translation to stop at the first stop codon, we can ask for that explicitly.

```python
translate_dna(coding_dna, stop_at_stop=True)
```

```
'MAIVMGR'
```

This is a good example of how a biological rule becomes a computational rule.

- move through the sequence three bases at a time

- map each codon to an amino acid
- optionally stop when a termination codon appears

Once you can write that logic, you begin to see how many "bioinformatics" tasks are just careful transformations of structured text.

7.10 Open reading frame checks

A biologically useful translation function is often paired with quick screening rules.

```
def looks_like_simple_orf(seq: str) -> bool:
    seq = require_valid_dna(seq)
    if len(seq) < 6:
        return False
    if len(seq) % 3 != 0:
        return False
    if not seq.startswith("ATG"):
        return False
    if seq[-3:] not in {"TAA", "TAG", "TGA"}:
        return False
    internal_codons = [seq[i:i+3] for i in range(3, len(seq) - 3, 3)]
    return all(codon not in {"TAA", "TAG", "TGA"} for codon in internal_codons)


looks_like_simple_orf("ATGGCCATTGTAATGGGCCGCTGA"),
looks_like_simple_orf("ATGTAGGCCTAA")
```

```
(True, False)
```

This function is not a full ORF finder. It is a compact quality-control check that asks whether a sequence matches a simple protein-coding pattern:

- starts with `ATG`
- has a length divisible by 3
- ends with a stop codon
- contains no internal stop codons

Those kinds of screening functions are excellent beginner projects because they are both conceptually clear and genuinely useful.

7.11 Sliding windows and local GC content

Global GC content is useful, but local composition can matter too. A sequence may have an acceptable overall GC content and still contain a problematic local region.

A sliding window lets us inspect local sequence properties.

```
def gc_content_windows(seq: str, window_size: int) -> list[tuple[int, str, float]]:
    seq = require_valid_dna(seq)
    windows = []
    for start in range(0, len(seq) - window_size + 1):
        window = seq[start:start + window_size]
        windows.append((start, window, gc_content(window)))
    return windows


window_results = gc_content_windows("ATGCGCGCTTAA", window_size=4)
window_results[:5]
```

```
[(0, 'ATGC', 0.5),
 (1, 'TGCG', 0.75),
 (2, 'GCGC', 1.0),
```

```
 (3, 'CGCG', 1.0),
 (4, 'GCGC', 1.0)]
```

We can summarize the most GC-rich local window like this.

```
max(window_results, key=lambda row: row[2])
```

```
(2, 'GCGC', 1.0)
```

This pattern generalizes well. Once you know how to slide a window along a sequence, you can search for many local properties:

- GC-rich regions
- homopolymers
- repeated motifs
- candidate primer regions
- subsequences matching a design rule

7.12 FASTA: the first real file format many people meet

In practice, sequences rarely arrive one at a time. They come in files.

FASTA is one of the simplest and most common formats for sequence data. A minimal FASTA file looks like this:

```
>seq1
ATGCGTAC
>seq2
ATGGGCTA
```

Each record begins with a header line starting with >, followed by one or more sequence lines.

Let us create a tiny FASTA file inside a temporary project area.

```
from pathlib import Path
import tempfile

chapter4_demo_dir = Path(tempfile.mkdtemp(prefix="synbio_ch4_"))
fasta_path = chapter4_demo_dir / "candidate_parts.fasta"

fasta_text = ">promoter_variant_1\nATGCGTACCGTTAG\n>promoter_variant_2\nATGGAATTCGGTCTCTAA\n>coding_varia

fasta_path.write_text(fasta_text)
str(fasta_path)
```

```
'/tmp/synbio_ch4_hjyjansg/candidate_parts.fasta'
```

Now we can write a simple FASTA parser.

```
def read_fasta(path: Path) -> list[dict[str, str]]:
    records = []
    header = None
    sequence_lines = []

    with path.open() as handle:
        for line in handle:
            line = line.strip()
            if not line:
                continue
            if line.startswith(">"):
                if header is not None:
                    records.append({
                        "id": header,
```

```
                    "sequence": require_valid_dna("".join(sequence_lines)),
                })
                header = line[1:]
                sequence_lines = []
            else:
                sequence_lines.append(line)

        if header is not None:
            records.append({
                "id": header,
                "sequence": require_valid_dna("".join(sequence_lines)),
            })

    return records


records = read_fasta(fasta_path)
records
```

```
[{'id': 'promoter_variant_1', 'sequence': 'ATGCGTACCGTTAG'},
 {'id': 'promoter_variant_2', 'sequence': 'ATGGAATTCGGTCTCTAA'},
 {'id': 'coding_variant_1', 'sequence': 'ATGGCCATTGTAATGGGCCGCTGA'}]
```

This parser is intentionally small and readable. A production parser would need more features and more testing, and later in the book we will discuss library-based approaches. But it is valuable to build one yourself at least once. It helps you understand what the file format actually contains.

7.13 Summarizing a small sequence library

Once the records are in Python, we can compute a compact summary for each one.

```
def summarize_sequence_record(record: dict[str, str]) -> dict[str, object]:
    seq = record["sequence"]
    return {
        "id": record["id"],
        "length": len(seq),
        "gc_percent": round(gc_content(seq) * 100, 1),
        "starts_with_atg": seq.startswith("ATG"),
        "contains_ecori": "GAATTC" in seq,
        "contains_bsai": "GGTCTC" in seq,
        "simple_orf": looks_like_simple_orf(seq),
    }


summaries = [summarize_sequence_record(record) for record in records]
summaries
```

```
[{'id': 'promoter_variant_1',
  'length': 14,
  'gc_percent': 50.0,
  'starts_with_atg': True,
  'contains_ecori': False,
  'contains_bsai': False,
  'simple_orf': False},
 {'id': 'promoter_variant_2',
  'length': 18,
  'gc_percent': 38.9,
  'starts_with_atg': True,
  'contains_ecori': True,
  'contains_bsai': True,
```

```
  'simple_orf': True},
 {'id': 'coding_variant_1',
  'length': 24,
  'gc_percent': 54.2,
  'starts_with_atg': True,
  'contains_ecori': False,
  'contains_bsai': False,
  'simple_orf': True}]
```

This is the heart of many sequence analysis workflows:

1. read a file
2. transform each record into a standard internal representation
3. compute derived properties
4. store those properties in a summary structure
5. sort, filter, or export the results

That pattern will appear again and again throughout the book.

7.14 Screening designs against simple rules

Suppose you are evaluating a few candidate coding sequences for inclusion in an assembly workflow. You might want sequences that satisfy rules such as:

- valid DNA only
- begins with `ATG`
- looks like a simple ORF
- does not contain EcoRI or BsaI sites
- GC content between 35% and 65%

That is easy to express in code.

```
def passes_simple_design_screen(seq: str) -> bool:
    seq = require_valid_dna(seq)
    gc = gc_content(seq)
    return (
        looks_like_simple_orf(seq)
        and "GAATTC" not in seq
        and "GGTCTC" not in seq
        and 0.35 <= gc <= 0.65
    )


for record in records:
    seq = record["sequence"]
    print(record["id"], passes_simple_design_screen(seq))
```

```
promoter_variant_1 False
promoter_variant_2 False
coding_variant_1 True
```

Here, the code is acting like an explicit protocol for computational triage. Instead of vaguely saying “screen the candidates,” we state the rules exactly.

That clarity has three advantages:

- you can rerun the same screen later
- collaborators can inspect the logic
- you can change the rules intentionally rather than accidentally

7.15 Ranking candidate sequences

Screening is often followed by ranking.

Imagine that we want to prefer sequences that pass the screen and are closest to 50% GC.

```
def design_score(seq: str) -> tuple[bool, float]:
    seq = require_valid_dna(seq)
    passes = passes_simple_design_screen(seq)
    gc_distance_from_target = abs(gc_content(seq) - 0.50)
    return (passes, -gc_distance_from_target)


ranked_records = sorted(
    records,
    key=lambda record: design_score(record["sequence"]),
    reverse=True,
)

[(record["id"], design_score(record["sequence"])) for record in ranked_records]
```

```
[('coding_variant_1', (True, -0.041666666666666663)),
 ('promoter_variant_1', (False, -0.0)),
 ('promoter_variant_2', (False, -0.1111111111111111))]
```

This example is intentionally simple, but it captures a real idea in computational design: once a design objective is formalized, Python can help you compare candidates consistently.

7.16 K-mers: treating sequences as overlapping words

Another useful way to think about sequences is as overlapping "words" of fixed length.

For DNA, a 3-mer is any sequence of 3 consecutive bases. A 4-mer is any sequence of 4 consecutive bases, and so on.

```
def kmers(seq: str, k: int) -> list[str]:
    seq = require_valid_dna(seq)
    return [seq[i:i+k] for i in range(len(seq) - k + 1)]


kmers("ATGCGTA", 3)
```

```
['ATG', 'TGC', 'GCG', 'CGT', 'GTA']
```

We can count them too.

```
three_mer_counts = Counter(kmers("ATGCGTATGC", 3))
three_mer_counts
```

```
Counter({'ATG': 2, 'TGC': 2, 'GCG': 1, 'CGT': 1, 'GTA': 1, 'TAT': 1})
```

K-mers appear in many kinds of biological computation:

- sequence comparison
- motif discovery
- composition analysis
- feature engineering for machine learning
- quick heuristics for clustering or classification

Even if you never use the term "k-mer" in daily lab conversation, the computational idea is worth learning.

7.17 Sequences as records, not just strings

By now you may notice a pattern: a sequence alone is rarely enough.

In real projects, a useful internal representation often looks more like this:

```
sequence_record = {
    "id": "coding_variant_1",
    "sequence": "ATGGCCATTGTAATGGGCCGCTGA",
    "description": "candidate reporter CDS",
    "source": "design_round_1",
}

sequence_record
```

```
{'id': 'coding_variant_1',
 'sequence': 'ATGGCCATTGTAATGGGCCGCTGA',
 'description': 'candidate reporter CDS',
 'source': 'design_round_1'}
```

That is, the sequence lives alongside metadata.

This is an important software design lesson for synthetic biology. Biological objects usually need:

- an identifier
- the raw sequence
- annotations or provenance
- derived properties computed later

As projects grow, you may use richer structures such as data classes, Biopython objects, SBOL representations, or database tables. But the core idea is already visible here.

7.18 A mini workflow: from FASTA to a design report

Let us finish the chapter with a compact workflow that ties together several ideas.

We will read the FASTA file, summarize each sequence, keep the best candidates, and write a CSV report.

```
import csv

report_rows = []

for record in records:
    seq = record["sequence"]
    report_rows.append({
        "id": record["id"],
        "length": len(seq),
        "gc_percent": round(gc_content(seq) * 100, 1),
        "passes_screen": passes_simple_design_screen(seq),
       "protein": translate_dna(seq, stop_at_stop=True) if looks_like_simple_orf(seq)
else "",
    })

report_rows
```

```
[{'id': 'promoter_variant_1',
  'length': 14,
  'gc_percent': 50.0,
  'passes_screen': False,
  'protein': ''},
 {'id': 'promoter_variant_2',
  'length': 18,
  'gc_percent': 38.9,
  'passes_screen': False,
  'protein': 'MEFGL'},
 {'id': 'coding_variant_1',
  'length': 24,
```

```
  'gc_percent': 54.2,
  'passes_screen': True,
  'protein': 'MAIVMGR'}]
```

Now we write the report to disk.

```
report_path = chapter4_demo_dir / "sequence_report.csv"

with report_path.open("w", newline="") as handle:
    writer = csv.DictWriter(
        handle,
        fieldnames=["id", "length", "gc_percent", "passes_screen", "protein"],
    )
    writer.writeheader()
    writer.writerows(report_rows)

report_path.exists(), report_path.name
```

```
(True, 'sequence_report.csv')
```

And we can inspect the contents.

```
report_path.read_text()
```

```
'id,length,gc_percent,passes_screen,protein\npromoter_variant_1,14,50.0,False,
\npromoter_variant_2,18,38.9,False,MEFGL\ncoding_variant_1,24,54.2,True,MAIVMGR\n'
```

This is not yet an industrial pipeline, but it is real computational biology:

- input file
- validation
- transformation
- screening
- output file

That is already enough to save time in a lab and reduce avoidable mistakes.

7.19 When to stop writing your own sequence utilities

It is good to build core tools like `reverse_complement` and a simple FASTA parser yourself once. These exercises teach the logic of sequence manipulation.

But in larger projects, it is usually better to rely on well-tested libraries rather than continually rebuilding everything from scratch.

Later in this book we will meet more specialized ecosystems and data models. For now, the important point is conceptual:

- write small utilities yourself to understand the problem
- switch to library abstractions when scale, complexity, or file-format breadth grows

That balance matters in research software. Reinventing every wheel is inefficient, but using a tool you do not understand can also make debugging difficult.

7.20 Biopython: from strings to standardized sequence records

Up to this point, we have deliberately written many sequence operations ourselves.

That was worth doing. It taught us what a reverse complement actually is, what FASTA records really contain, and how translation logic works.

But this is also the point where a working synthetic biology project usually reaches for a library.

Biopython is the most widely used general-purpose Python toolkit for molecular biology. Its core abstractions are especially important for sequence work:

- `Seq`, a sequence object that behaves much like a string but also knows about biological operations

- `SeqRecord`, a richer container that stores a sequence together with identifiers, descriptions, annotations, and features
- `SeqIO`, a uniform interface for reading and writing sequence file formats such as FASTA and GenBank
- `SearchIO`, a standardized interface for parsing sequence-search outputs such as BLAST

In other words, Biopython helps you move from *sequence-like strings* to *standardized sequence records*.

If you want to run the examples below locally, install the package in the same environment you use for Quarto:

```
pip install biopython
```

7.21 `Seq`: a biological sequence object

The Biopython `Seq` object looks familiar because it behaves in many ways like a string, but it also provides biological methods directly.

```
from Bio.Seq import Seq

bio_seq = Seq("ATGGCCATTGTAATGGGCCGCTGA")

len(bio_seq), bio_seq[:6], bio_seq.reverse_complement()
```

```
(24, Seq('ATGGCC'), Seq('TCAGCGGCCCATTACAATGGCCAT'))
```

Now operations that we previously implemented by hand come built in.

```
bio_seq.transcribe(), bio_seq.translate(), bio_seq.translate(to_stop=True)
```

```
(Seq('AUGGCCAUUGUAAUGGGCCGCUGA'), Seq('MAIVMGR*'), Seq('MAIVMGR'))
```

This is a good example of why library abstractions matter. The biological intent is visible directly in the code.

7.22 `SeqRecord`: a sequence plus metadata

A bare sequence string is useful, but most real work needs more context.

- What is the identifier?
- What construct or part does this correspond to?
- What description should appear in a file?
- What annotations or features belong to this sequence?

That is what `SeqRecord` is for.

```
from Bio.SeqRecord import SeqRecord

record = SeqRecord(
    bio_seq,
    id="demo_cds_1",
    name="demo_cds_1",
    description="Synthetic demonstration coding sequence",
)
record.annotations["source"] = "chapter4_demo"

record.id, record.description, str(record.seq)
```

```
('demo_cds_1',
 'Synthetic demonstration coding sequence',
 'ATGGCCATTGTAATGGGCCGCTGA')
```

The key shift is conceptual.

A SeqRecord is not just a string that happens to contain nucleotides. It is a **standardized record object** with agreed-upon fields and behaviors. Once your code works with records rather than naked strings, it becomes easier to read, easier to extend, and easier to exchange with other tools.

7.23 Reading FASTA with SeqIO

Earlier in the chapter, we wrote our own FASTA parser. That was useful as a learning exercise. In most real projects, however, it is better to use Biopython's parser.

```
from Bio import SeqIO

bio_records = list(SeqIO.parse(fasta_path, "fasta"))
[(record.id, len(record), str(record.seq[:6])) for record in bio_records]
```

```
[('promoter_variant_1', 14, 'ATGCGT'),
 ('promoter_variant_2', 18, 'ATGGAA'),
 ('coding_variant_1', 24, 'ATGGCC')]
```

Notice what changed.

We did not get back dictionaries or raw strings. We got a list of SeqRecord objects. Each one has a standard interface:

```
first_bio_record = bio_records[0]
first_bio_record.id,
first_bio_record.description, type(first_bio_record.seq).__name__
```

```
('promoter_variant_1', 'promoter_variant_1', 'Seq')
```

That standardization makes common operations concise.

```
record_dict = SeqIO.to_dict(bio_records)
record_dict["coding_variant_1"].seq.translate(to_stop=True)
```

```
Seq('MAIVMGR')
```

We can also write records back out to disk after filtering or editing.

```
biopython_filtered_path = chapter4_demo_dir / "no_ecori_records.fasta"
filtered_records = [record for record in bio_records if "GAATTC" not in record.seq]
SeqIO.write(filtered_records, biopython_filtered_path, "fasta")

biopython_filtered_path.name, biopython_filtered_path.exists()
```

```
('no_ecori_records.fasta', True)
```

And because FASTA is simple text, we can inspect the output directly.

```
biopython_filtered_path.read_text()
```

```
'>promoter_variant_1\nATGCGTACCGTTAG\n>coding_variant_1\nATGGCCATTGTAATGGGCCGCTGA\n'
```

This is one of the main benefits of SeqIO: the same read-process-write pattern works across many file formats.

7.24 GenBank: richer records with annotations and features

FASTA is excellent when all you need is a header and a sequence. But many synthetic biology tasks need more structure.

A GenBank record can include:

- accession-like identifiers

- organism or source information
- free-text description
- annotations about the whole record
- explicitly marked features such as promoters, CDSs, terminators, and origins

Let us create a tiny GenBank example and parse it with Biopython.

```
genbank_path = chapter4_demo_dir / "demo_construct.gb"

genbank_text = """LOCUS       DEMO0001                  39 bp    DNA     linear   SYN
01-JAN-2024
DEFINITION  Synthetic demonstration construct.
ACCESSION   DEMO0001
VERSION     DEMO0001.1
KEYWORDS    synthetic biology.
SOURCE      synthetic DNA construct
  ORGANISM  synthetic DNA construct
            other sequences; artificial sequences.
FEATURES             Location/Qualifiers
     source          1..39
                     /organism=\"synthetic DNA construct\"
                     /mol_type=\"other DNA\"
     promoter        1..9
                     /label=\"P_demo\"
     CDS             10..33
                     /gene=\"demoGFP\"
                     /product=\"demo protein\"
                     /codon_start=1
                     /translation=\"MAIVMGR\"
ORIGIN
        1 ttgacatat atggccatt gtaatgggcc gctgaaatata
//
"""

genbank_path.write_text(genbank_text)
genbank_record = SeqIO.read(genbank_path, "genbank")

genbank_record.id, len(genbank_record), genbank_record.description
```

```
('DEMO0001.1', 39, 'Synthetic demonstration construct')
```

Now we can inspect whole-record annotations.

```
sorted(genbank_record.annotations.keys())
```

```
['accessions',
 'data_file_division',
 'date',
 'keywords',
 'molecule_type',
 'organism',
 'sequence_version',
 'source',
 'taxonomy',
 'topology']
```

And we can inspect the features that were explicitly encoded in the file.

```
[(feature.type, int(feature.location.start), int(feature.location.end)) for feature
in genbank_record.features]
```

```
[('source', 0, 39), ('promoter', 0, 9), ('CDS', 9, 33)]
```

A GenBank feature is not just a text label. It has typed structure and a location on the sequence.

```
cds_feature = next(feature for feature in genbank_record.features if feature.type
== "CDS")
str(cds_feature.extract(genbank_record.seq)),      cds_feature.qualifiers["gene"][0],
cds_feature.qualifiers["translation"][0]
```

```
('ATGGCCATTGTAATGGGCCGCTGA', 'demoGFP', 'MAIVMGR')
```

This is where Biopython becomes especially powerful for synthetic biology.

A richly annotated record gives your code a standard way to talk about sequence parts:

- the whole construct lives in `genbank_record`
- the DNA lives in `genbank_record.seq`
- features live in `genbank_record.features`
- metadata and annotations live in consistent attributes

That is much more robust than passing around loosely formatted strings and hoping that everyone interprets them the same way.

7.25 Standardizing your own records

Biopython is not only for reading public formats. It is also useful for creating your own standardized sequence records before writing them to disk or passing them into later analysis steps.

```
from Bio.SeqFeature import SeqFeature, FeatureLocation

custom_record = SeqRecord(
    Seq("ATGGCCATTGTAATGGGCCGCTGA"),
    id="design_001",
    name="design_001",
    description="Custom design record generated in Python",
)
custom_record.annotations["molecule_type"] = "DNA"
custom_record.features = [
              SeqFeature(FeatureLocation(0,    len(custom_record.seq)),    type="CDS",
qualifiers={"gene": ["demo_gene"]})
]

custom_record.id, custom_record.features[0].type, str(custom_record.seq.translate())
```

```
('design_001', 'CDS', 'MAIVMGR*')
```

Once your data has been put into a `SeqRecord`, many operations become standardized:

- writing it to FASTA or GenBank
- slicing sequence regions while keeping semantics clear
- attaching feature information
- passing the object between modules instead of inventing custom ad hoc dictionaries

7.26 BLAST with Biopython

Sequence work often moves from *what is this sequence?* to *what does it resemble?*

That is where BLAST enters. Biopython helps in two main ways:

1. submitting BLAST searches
2. parsing BLAST outputs into structured Python objects

For small exploratory work, you can submit a query to NCBI directly from Python. The example below is shown for completeness but is not executed in this chapter because it depends on internet access and should be used with care.

```
from Bio.Blast import NCBIWWW

NCBIWWW.email = "your_email@example.com"

with  NCBIWWW.qblast("blastn",  "nt",  str(record_dict["coding_variant_1"].seq))  as
result_handle:
    blast_xml = result_handle.read()

# Save the XML for later parsing
(chapter4_demo_dir / "coding_variant_1_blast.xml").write_text(blast_xml)
```

```
1997
```

In teaching and in production pipelines, it is often better to separate the two tasks:

- run BLAST remotely or locally
- parse the output later in a reproducible script

Biopython's `SearchIO` module is designed for this second task.

Let us create a tiny BLAST-like tabular output file and parse it.

```
from Bio import SearchIO

blast_tab_path = chapter4_demo_dir / "toy_blast.tsv"
blast_tab_text = """query1  subjectA    100.000 12  0   0   1   12  5   16  1e-20   50.0
query1  subjectB    91.667  12  1   0   1   12  20  31  2e-10   40.0
query2  subjectA    87.500  8   1   0   1   8   30  37  1e-05   25.0
"""
blast_tab_path.write_text(blast_tab_text)

qresults = list(SearchIO.parse(blast_tab_path, "blast-tab"))
[(qresult.id, len(qresult)) for qresult in qresults]
```

```
[('query1', 2), ('query2', 1)]
```

Each query result contains one or more hits, and each hit contains one or more HSPs (high-scoring segment pairs).

```
first_qresult = qresults[0]
best_hit = first_qresult[0]
best_hsp = best_hit.hsps[0]

best_hit.id, best_hsp.evalue, best_hsp.ident_pct
```

```
('subjectA', 1e-20, 100.0)
```

We can inspect the coordinates as well.

```
best_hsp.query_start, best_hsp.query_end, best_hsp.hit_start, best_hsp.hit_end
```

```
(0, 12, 4, 16)
```

This object model matters because it standardizes BLAST parsing. Instead of manually splitting lines and remembering which column means what, you can work with named attributes on structured objects.

A small summarizer turns those structured results into a Python-friendly report.

```
def summarize_blast_qresults(qresults):
    rows = []
    for qresult in qresults:
        for hit in qresult:
            top_hsp = hit.hsps[0]
```

```
            rows.append(
                {
                    "query_id": qresult.id,
                    "hit_id": hit.id,
                    "evalue": top_hsp.evalue,
                    "bitscore": top_hsp.bitscore,
                    "identity_percent": round(top_hsp.ident_pct, 1),
                    "query_start": top_hsp.query_start,
                    "query_end": top_hsp.query_end,
                    "hit_start": top_hsp.hit_start,
                    "hit_end": top_hsp.hit_end,
                }
            )
    return rows

blast_summary_rows = summarize_blast_qresults(qresults)
blast_summary_rows
```

```
[{'query_id': 'query1',
  'hit_id': 'subjectA',
  'evalue': 1e-20,
  'bitscore': 50.0,
  'identity_percent': 100.0,
  'query_start': 0,
  'query_end': 12,
  'hit_start': 4,
  'hit_end': 16},
 {'query_id': 'query1',
  'hit_id': 'subjectB',
  'evalue': 2e-10,
  'bitscore': 40.0,
  'identity_percent': 91.7,
  'query_start': 0,
  'query_end': 12,
  'hit_start': 19,
  'hit_end': 31},
 {'query_id': 'query2',
  'hit_id': 'subjectA',
  'evalue': 1e-05,
  'bitscore': 25.0,
  'identity_percent': 87.5,
  'query_start': 0,
  'query_end': 8,
  'hit_start': 29,
  'hit_end': 37}]
```

That list of dictionaries is exactly the kind of object you might later convert into a table, save to CSV, or merge with other experimental metadata.

7.27 Why this matters in synthetic biology

Biopython does more than save typing.
It gives you **shared data models**.
That matters because synthetic biology work often moves across tools and stages:

- sequence design
- file exchange
- annotation
- search and comparison
- downstream analysis and reporting

If each stage invents its own record format, your code base becomes fragile. If instead you rely on standard objects like `SeqRecord`, `SeqIO`, and `SearchIO`, many operations become easier to automate and easier to reason about.

This is also a cultural shift.

Earlier in the chapter, we learned sequence logic by writing our own functions. With Biopython, we start to participate in a broader software ecosystem where records, annotations, and search results follow shared conventions.

That is a major step toward research code that other people can read, reuse, and extend.

7.28 Exercises

1. Write a function `has_homopolymer(seq, n)` that returns `True` if a sequence contains a run of the same base of length `n` or more.
2. Modify `find_present_sites` so that it returns both the enzyme name and the position of the match.
3. Write a function that counts codons in a coding sequence and returns the most common codon.
4. Extend `read_fasta` so that it also preserves the full FASTA header and splits out an identifier from the description.
5. Add a new screening rule to `passes_simple_design_screen`, such as a maximum homopolymer length or a required stop codon.
6. Create a second FASTA file with deliberately invalid input and confirm that your validation logic fails clearly.

7.29 Recap

In this chapter, you learned how to treat sequences as structured, biologically meaningful data.

You used Python to:

- index and slice sequences
- compute base composition and GC content
- validate DNA input
- compute complements and reverse complements
- transcribe DNA to RNA
- translate codons into protein sequences
- check simple ORF-like properties
- scan for motifs and restriction sites
- generate k-mers
- read FASTA records
- summarize and screen a small sequence library
- write a compact CSV report

This chapter marks an important transition in the book.

In the earlier chapters, Python was the subject. Here, biology and Python begin to meet directly.

From this point on, we will increasingly use code not only to learn programming concepts, but to express the kinds of reasoning that synthetic biology depends on.

8. Tabular Experimental Data

Most synthetic biology projects do not begin with a model.

They begin with a table.

You grow strains, induce cultures, measure fluorescence, record optical density, annotate constructs, and export the results from an instrument or a spreadsheet. Very quickly, your work becomes a problem of **tabular data management**.

That is why `pandas` is one of the most useful Python libraries for synthetic biology. It gives us a way to read, clean, reshape, summarize, and save experimental data without leaving Python.

In this chapter, we will introduce `pandas` through the kinds of tables that appear constantly in biology:

- fluorescence measurements
- growth measurements
- induction assays
- construct metadata
- replicate-level observations
- summary tables for downstream analysis

This chapter also introduces one of the most important habits for scientific computing: **tidy data**.

Once we introduce tidy data, we will use it as the default tabular format for the rest of the book.

8.1 Why tidy data matters

A table can be easy for a human to read but hard for code to analyze.

That tension appears everywhere in biology. Instrument exports, manually assembled spreadsheets, and presentation-ready summary tables are often arranged for human inspection rather than computation.

A tidy table follows a simple idea:

- each **row** is one observation
- each **column** is one variable
- each type of observational unit gets its own table

For a fluorescence induction experiment, one observation might be:

- one strain
- at one inducer concentration
- at one time point
- for one replicate

In a tidy table, those properties become columns.

That might sound abstract, so let us start with a non-tidy example.

8.2 A wide table is often the first thing you get

Suppose a plate reader exports fluorescence values like this.

```
import pandas as pd

wide_data = pd.DataFrame(
    {
        "time_h": [0, 2, 4],
        "WT_rep1": [120, 135, 140],
        "WT_rep2": [118, 132, 141],
        "Sensor_rep1": [125, 410, 980],
        "Sensor_rep2": [130, 430, 1020],
```

```
    }
)

wide_data
```

	time_h	WT_rep1	WT_rep2	Sensor_rep1	Sensor_rep2
0	0	120	118	125	130
1	2	135	132	410	430
2	4	140	141	980	1020

A human can read this table easily, but the column names are mixing several different variables together:

- strain identity
- replicate identity
- measurement value

That makes downstream analysis awkward.

For example, imagine asking any of these questions:

- what is the mean fluorescence for each strain at each time point?
- how much variation is there across replicates?
- can we merge this table with construct metadata?
- can we plot all strains with a common grammar later?

Those operations become easier when the table is tidy.

8.3 Reshaping wide data into tidy data

We can use `melt()` to convert the wide table into a longer, tidier format.

```
tidy_from_wide = wide_data.melt(
    id_vars="time_h",
    var_name="sample",
    value_name="fluorescence",
)

tidy_from_wide
```

	time_h	sample	fluorescence
0	0	WT_rep1	120
1	2	WT_rep1	135
2	4	WT_rep1	140
3	0	WT_rep2	118
4	2	WT_rep2	132
5	4	WT_rep2	141
6	0	Sensor_rep1	125
7	2	Sensor_rep1	410
8	4	Sensor_rep1	980
9	0	Sensor_rep2	130
10	2	Sensor_rep2	430
11	4	Sensor_rep2	1020

Now each row is closer to a single observation, but the `sample` column still combines two variables: strain and replicate.

We can separate them.

```
parsed          =          tidy_from_wide["sample"].str.extract(r"(?P<strain>.+)_rep(?
P<replicate>\d+)")

tidy_from_wide = pd.concat([tidy_from_wide, parsed], axis=1)
tidy_from_wide["replicate"] = tidy_from_wide["replicate"].astype(int)

tidy_from_wide = tidy_from_wide[["time_h", "strain", "replicate", "fluorescence"]]

tidy_from_wide
```

	time_h	strain	replicate	fluorescence
0	0	WT	1	120
1	2	WT	1	135
2	4	WT	1	140
3	0	WT	2	118
4	2	WT	2	132
5	4	WT	2	141
6	0	Sensor	1	125
7	2	Sensor	1	410
8	4	Sensor	1	980
9	0	Sensor	2	130
10	2	Sensor	2	430
11	4	Sensor	2	1020

This new table is much better for computation:

- one row is one measurement
- `time_h`, `strain`, and `replicate` are explicit variables
- `fluorescence` is the measured value

From this point onward in the book, we will prefer this style.

When we talk about experimental tables, assume that we want them in **tidy format** unless there is a strong reason not to.

8.4 Building a tidy experiment table

Let us create a slightly richer dataset. This time, each row will represent one observation from an induction experiment.

```
experiment = pd.DataFrame(
    [
        {"strain": "WT", "inducer_mM": 0.0, "time_h": 4, "replicate": 1, "od600": 0.82,
"fluorescence": 145},
        {"strain": "WT", "inducer_mM": 0.0, "time_h": 4, "replicate": 2, "od600": 0.80,
"fluorescence": 150},
        {"strain": "WT", "inducer_mM": 1.0, "time_h": 4, "replicate": 1, "od600": 0.81,
"fluorescence": 152},
        {"strain": "WT", "inducer_mM": 1.0, "time_h": 4, "replicate": 2, "od600": 0.79,
"fluorescence": 149},
        {"strain": "Sensor", "inducer_mM": 0.0, "time_h": 4, "replicate": 1, "od600":
```

```
0.77, "fluorescence": 210},
         {"strain": "Sensor", "inducer_mM": 0.0, "time_h": 4, "replicate": 2, "od600":
0.75, "fluorescence": 220},
         {"strain": "Sensor", "inducer_mM": 1.0, "time_h": 4, "replicate": 1, "od600":
0.78, "fluorescence": 920},
         {"strain": "Sensor", "inducer_mM": 1.0, "time_h": 4, "replicate": 2, "od600":
0.76, "fluorescence": 980},
        {"strain": "Amplifier", "inducer_mM": 0.0, "time_h": 4, "replicate": 1, "od600":
0.73, "fluorescence": 260},
        {"strain": "Amplifier", "inducer_mM": 0.0, "time_h": 4, "replicate": 2, "od600":
0.72, "fluorescence": 255},
        {"strain": "Amplifier", "inducer_mM": 1.0, "time_h": 4, "replicate": 1, "od600":
0.74, "fluorescence": 1380},
        {"strain": "Amplifier", "inducer_mM": 1.0, "time_h": 4, "replicate": 2, "od600":
0.71, "fluorescence": 1415},
    ]
)

experiment
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence
0	WT	0.0	4	1	0.82	145
1	WT	0.0	4	2	0.80	150
2	WT	1.0	4	1	0.81	152
3	WT	1.0	4	2	0.79	149
4	Sensor	0.0	4	1	0.77	210
5	Sensor	0.0	4	2	0.75	220
6	Sensor	1.0	4	1	0.78	920
7	Sensor	1.0	4	2	0.76	980
8	Amplifier	0.0	4	1	0.73	260
9	Amplifier	0.0	4	2	0.72	255
10	Amplifier	1.0	4	1	0.74	1380
11	Amplifier	1.0	4	2	0.71	1415

This is the kind of structure that ages well.

A tidy table like this is easy to:

- filter by strain or condition
- normalize measurements
- group replicates
- compute summaries
- merge with metadata
- write back to disk

8.5 Inspecting a DataFrame

A `pandas` table is called a **DataFrame**.

Some of the first things we usually check are the size, column names, and data types.

```
experiment.shape
```

```
(12, 6)
```

```
experiment.columns.tolist()
```

```
['strain', 'inducer_mM', 'time_h', 'replicate', 'od600', 'fluorescence']
```

```
experiment.dtypes
```

```
strain              str
inducer_mM      float64
time_h            int64
replicate         int64
od600           float64
fluorescence      int64
dtype: object
```

And for a quick preview:

```
experiment.head()
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence
0	WT	0.0	4	1	0.82	145
1	WT	0.0	4	2	0.80	150
2	WT	1.0	4	1	0.81	152
3	WT	1.0	4	2	0.79	149
4	Sensor	0.0	4	1	0.77	210

This may feel routine, but it is part of good scientific hygiene. A surprising amount of debugging is simply discovering that a column has the wrong type or an unexpected name.

8.6 Selecting columns

Column selection is straightforward.

```
experiment["fluorescence"]
```

```
0      145
1      150
2      152
3      149
4      210
5      220
6      920
7      980
8      260
9      255
10    1380
11    1415
Name: fluorescence, dtype: int64
```

To select multiple columns, pass a list of names.

```
experiment[["strain", "inducer_mM", "fluorescence"]]
```

	strain	inducer_mM	fluorescence
0	WT	0.0	145
1	WT	0.0	150

	strain	inducer_mM	fluorescence
2	WT	1.0	152
3	WT	1.0	149
4	Sensor	0.0	210
5	Sensor	0.0	220
6	Sensor	1.0	920
7	Sensor	1.0	980
8	Amplifier	0.0	260
9	Amplifier	0.0	255
10	Amplifier	1.0	1380
11	Amplifier	1.0	1415

This is useful when you want to focus on a subset of variables without modifying the original table.

8.7 Filtering rows

Because our table is tidy, filtering becomes very natural.

```
sensor_only = experiment[experiment["strain"] == "Sensor"]
sensor_only
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence
4	Sensor	0.0	4	1	0.77	210
5	Sensor	0.0	4	2	0.75	220
6	Sensor	1.0	4	1	0.78	920
7	Sensor	1.0	4	2	0.76	980

We can combine conditions too.

```
induced_sensor = experiment[
    (experiment["strain"] == "Sensor") & (experiment["inducer_mM"] == 1.0)
]

induced_sensor
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence
6	Sensor	1.0	4	1	0.78	920
7	Sensor	1.0	4	2	0.76	980

That one line already expresses a biologically meaningful question: show me the induced measurements for the sensor strain.

Sorting is equally common.

```
experiment.sort_values(["strain", "inducer_mM", "replicate"])
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence
8	Amplifier	0.0	4	1	0.73	260
9	Amplifier	0.0	4	2	0.72	255

	strain	inducer_mM	time_h	replicate	od600	fluorescence
10	Amplifier	1.0	4	1	0.74	1380
11	Amplifier	1.0	4	2	0.71	1415
4	Sensor	0.0	4	1	0.77	210
5	Sensor	0.0	4	2	0.75	220
6	Sensor	1.0	4	1	0.78	920
7	Sensor	1.0	4	2	0.76	980
0	WT	0.0	4	1	0.82	145
1	WT	0.0	4	2	0.80	150
2	WT	1.0	4	1	0.81	152
3	WT	1.0	4	2	0.79	149

8.8 Adding derived columns

Raw fluorescence is often less informative than fluorescence normalized by culture density.
A common first pass is to divide by OD600.

```
experiment = experiment.assign(
    norm_fluorescence=experiment["fluorescence"] / experiment["od600"]
)

experiment[["strain", "inducer_mM", "replicate", "norm_fluorescence"]]
```

	strain	inducer_mM	replicate	norm_fluorescence
0	WT	0.0	1	176.829268
1	WT	0.0	2	187.500000
2	WT	1.0	1	187.654321
3	WT	1.0	2	188.607595
4	Sensor	0.0	1	272.727273
5	Sensor	0.0	2	293.333333
6	Sensor	1.0	1	1179.487179
7	Sensor	1.0	2	1289.473684
8	Amplifier	0.0	1	356.164384
9	Amplifier	0.0	2	354.166667
10	Amplifier	1.0	1	1864.864865
11	Amplifier	1.0	2	1992.957746

This is a good example of why tidy data helps. Each row already contains the variables required for the calculation, so adding a derived column is simple and transparent.

8.9 Grouping and summarizing replicates

Biological experiments almost always involve replicate measurements.
A tidy table makes grouped summaries very convenient.

```
summary = (
    experiment.groupby(["strain", "inducer_mM"])
    .agg(
        n=("norm_fluorescence", "size"),
        mean_norm_fluorescence=("norm_fluorescence", "mean"),
        sd_norm_fluorescence=("norm_fluorescence", "std"),
        mean_od600=("od600", "mean"),
    )
    .reset_index()
)

summary
```

	strain	inducer_mM	n	mean_norm_fluorescence	sd_norm_fluorescence	mean_od600
0	Amplifier	0.0	2	355.165525	1.412599	0.725
1	Amplifier	1.0	2	1928.911306	90.575345	0.725
2	Sensor	0.0	2	283.030303	14.570685	0.760
3	Sensor	1.0	2	1234.480432	77.772203	0.770
4	WT	0.0	2	182.164634	7.545347	0.810
5	WT	1.0	2	188.130958	0.674066	0.800

This summary table is also tidy.

Each row now represents one summarized condition rather than one raw replicate. That is still perfectly tidy, because the observational unit has changed. The important thing is that the rows and columns remain explicit.

This distinction is worth remembering:

- a **replicate-level tidy table** has one row per measurement
- a **condition-level tidy summary** has one row per summarized condition

Both are tidy. They simply describe different units of analysis.

8.10 Quantifying induction

Once we have grouped summaries, we can compute simple biological metrics such as fold induction.

```
baseline = summary[summary["inducer_mM"] == 0.0][
    ["strain", "mean_norm_fluorescence"]
].rename(columns={"mean_norm_fluorescence": "baseline_norm_fluorescence"})

summary_with_baseline = summary.merge(baseline, on="strain")
summary_with_baseline["fold_induction"] = (
    summary_with_baseline["mean_norm_fluorescence"]
    / summary_with_baseline["baseline_norm_fluorescence"]
)

summary_with_baseline
```

	strain	inducer_mM	n	mean_norm_fluorescence	sd_norm_fluorescence	mean_od600	baseline_norm_fluorescence	fold_induction
0	Amplifier	0.0	2	355.165525	1.412599	0.725	355.165525	1.000000
1	Amplifier	1.0	2	1928.911306	90.575345	0.725	355.165525	5.431021
2	Sensor	0.0	2	283.030303	14.570685	0.760	283.030303	1.000000
3	Sensor	1.0	2	1234.480432	77.772203	0.770	283.030303	4.361655
4	WT	0.0	2	182.164634	7.545347	0.810	182.164634	1.000000

	strain	inducer_mM	n	mean_norm_fluorescence	sd_norm_fluorescence	mean_od600	baseline_norm_fluorescence	fold_induction
5	WT	1.0	2	188.130958	0.674066	0.800	182.164634	1.032752

Now we have a compact summary that answers a real synthetic biology question: how strongly does each strain respond to inducer?

8.11 Missing values happen

Real experiments are not perfectly complete. Wells fail, cultures contaminate, measurements are dropped, and metadata go missing.

`pandas` uses special missing values such as `NaN` to represent absent entries.

```
with_missing = experiment.copy()
with_missing.loc[3, "od600"] = None
with_missing.loc[8, "fluorescence"] = None

with_missing
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence	norm_fluorescence
0	WT	0.0	4	1	0.82	145.0	176.829268
1	WT	0.0	4	2	0.80	150.0	187.500000
2	WT	1.0	4	1	0.81	152.0	187.654321
3	WT	1.0	4	2	NaN	149.0	188.607595
4	Sensor	0.0	4	1	0.77	210.0	272.727273
5	Sensor	0.0	4	2	0.75	220.0	293.333333
6	Sensor	1.0	4	1	0.78	920.0	1179.487179
7	Sensor	1.0	4	2	0.76	980.0	1289.473684
8	Amplifier	0.0	4	1	0.73	NaN	356.164384
9	Amplifier	0.0	4	2	0.72	255.0	354.166667
10	Amplifier	1.0	4	1	0.74	1380.0	1864.864865
11	Amplifier	1.0	4	2	0.71	1415.0	1992.957746

We can ask how many missing values appear in each column.

```
with_missing.isna().sum()
```

```
strain               0
inducer_mM           0
time_h               0
replicate            0
od600                1
fluorescence         1
norm_fluorescence    0
dtype: int64
```

Sometimes the correct action is to remove incomplete rows.

```
complete_cases = with_missing.dropna(subset=["od600", "fluorescence"])
complete_cases
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence	norm_fluorescence
0	WT	0.0	4	1	0.82	145.0	176.829268

	strain	inducer_mM	time_h	replicate	od600	fluorescence	norm_fluorescence
1	WT	0.0	4	2	0.80	150.0	187.500000
2	WT	1.0	4	1	0.81	152.0	187.654321
4	Sensor	0.0	4	1	0.77	210.0	272.727273
5	Sensor	0.0	4	2	0.75	220.0	293.333333
6	Sensor	1.0	4	1	0.78	920.0	1179.487179
7	Sensor	1.0	4	2	0.76	980.0	1289.473684
9	Amplifier	0.0	4	2	0.72	255.0	354.166667
10	Amplifier	1.0	4	1	0.74	1380.0	1864.864865
11	Amplifier	1.0	4	2	0.71	1415.0	1992.957746

Other times we may want to keep the rows but mark that some analysis cannot yet be performed.

The important point is not to ignore missingness. Missing values are part of the data-generating process and often reflect something biologically or experimentally meaningful.

8.12 Reading tidy data from a CSV file

In practice, we usually load data from files rather than typing them directly into Python.

Here is a small CSV example using an in-memory text buffer.

```
from io import StringIO

csv_text = """strain,inducer_mM,time_h,replicate,od600,fluorescence
WT,0.0,4,1,0.82,145
WT,0.0,4,2,0.80,150
Sensor,1.0,4,1,0.78,920
Sensor,1.0,4,2,0.76,980
"""

loaded = pd.read_csv(StringIO(csv_text))
loaded
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence
0	WT	0.0	4	1	0.82	145
1	WT	0.0	4	2	0.80	150
2	Sensor	1.0	4	1	0.78	920
3	Sensor	1.0	4	2	0.76	980

On disk, the equivalent workflow would look like this:

```
from pathlib import Path

example_path = Path("data") / "induction_results.csv"
example_path
```

```
PosixPath('data/induction_results.csv')
```

If the file exists, we would read it with:

```
# pd.read_csv(example_path)
```

In a real project, the most important thing is that the CSV itself should already be organized as a tidy table whenever possible.

8.13 Merging measurement tables with metadata

Experiments often involve more than one table.
For example, one table may contain measurements, while another contains construct metadata.

```
metadata = pd.DataFrame(
    {
        "strain": ["WT", "Sensor", "Amplifier"],
        "plasmid": ["pControl", "pSensor", "pAmp"],
        "reporter": ["none", "GFP", "GFP"],
        "host": ["E. coli", "E. coli", "E. coli"],
    }
)

metadata
```

	strain	plasmid	reporter	host
0	WT	pControl	none	E. coli
1	Sensor	pSensor	GFP	E. coli
2	Amplifier	pAmp	GFP	E. coli

Because both tables have a `strain` column, we can merge them.

```
annotated = experiment.merge(metadata, on="strain", how="left")
annotated
```

	strain	inducer_mM	time_h	replicate	od600	fluorescence	norm_fluorescence	plasmid	reporter	host
0	WT	0.0	4	1	0.82	145	176.829268	pControl	none	E. coli
1	WT	0.0	4	2	0.80	150	187.500000	pControl	none	E. coli
2	WT	1.0	4	1	0.81	152	187.654321	pControl	none	E. coli
3	WT	1.0	4	2	0.79	149	188.607595	pControl	none	E. coli
4	Sensor	0.0	4	1	0.77	210	272.727273	pSensor	GFP	E. coli
5	Sensor	0.0	4	2	0.75	220	293.333333	pSensor	GFP	E. coli
6	Sensor	1.0	4	1	0.78	920	1179.487179	pSensor	GFP	E. coli
7	Sensor	1.0	4	2	0.76	980	1289.473684	pSensor	GFP	E. coli
8	Amplifier	0.0	4	1	0.73	260	356.164384	pAmp	GFP	E. coli
9	Amplifier	0.0	4	2	0.72	255	354.166667	pAmp	GFP	E. coli
10	Amplifier	1.0	4	1	0.74	1380	1864.864865	pAmp	GFP	E. coli

	strain	inducer_mM	time_h	replicate	od600	fluorescence	norm_fluorescence	plasmid	reporter	host
11	Amplifier	1.0	4	2	0.71	1415	1992.957746	pAmp	GFP	E. coli

This is one of the major reasons to preserve tidy structure. Joins become much easier when variables are explicit and consistently named.

8.14 Selecting one table shape for downstream work

Once you begin analyzing tidy data, it is tempting to keep making presentation-friendly versions of the table. That is fine for slides or papers, but for computation it is better to keep a **canonical tidy table** and derive other forms when needed.

For example, if you ever need a wide table for reporting, you can create it from the tidy version.

```
wide_summary = summary.pivot(
    index="strain",
    columns="inducer_mM",
    values="mean_norm_fluorescence",
)

wide_summary
```

inducer_mM	0.0	1.0
strain		
Amplifier	355.165525	1928.911306
Sensor	283.030303	1234.480432
WT	182.164634	188.130958

That is a useful display, but it is not the format we want to keep as the primary analytical table.

A good default workflow is:

1. clean the raw data
2. convert to tidy format
3. do all analysis in tidy format
4. create wide or presentation-specific views only at the end

We will follow that pattern throughout the rest of this book.

8.15 Saving processed data

After cleaning and summarizing a dataset, it is often helpful to save the result for later chapters, figures, or reports.

```
output_dir = Path("results")
output_dir.mkdir(exist_ok=True)

output_path = output_dir / "induction_summary.csv"
summary_with_baseline.to_csv(output_path, index=False)

output_path
```

```
PosixPath('results/induction_summary.csv')
```

That small step turns an analysis from a one-off calculation into a reproducible workflow artifact.

8.16 A small end-to-end example

Let us combine the main ideas of the chapter into a compact workflow.

```
workflow_result = (
    experiment
    .assign(norm_fluorescence=lambda df: df["fluorescence"] / df["od600"])
    .groupby(["strain", "inducer_mM"])
    .agg(mean_norm_fluorescence=("norm_fluorescence", "mean"))
    .reset_index()
    .sort_values(["strain", "inducer_mM"])
)

workflow_result
```

	strain	inducer_mM	mean_norm_fluorescence
0	Amplifier	0.0	355.165525
1	Amplifier	1.0	1928.911306
2	Sensor	0.0	283.030303
3	Sensor	1.0	1234.480432
4	WT	0.0	182.164634
5	WT	1.0	188.130958

This pipeline works cleanly because the underlying table is tidy.

That is the theme to carry forward.

As datasets become larger and models become more sophisticated, tidy organization continues to pay off.

8.17 Recap

In this chapter, we learned how to:

- represent biological experiments as `pandas` DataFrames
- distinguish wide tables from tidy tables
- reshape data into tidy format
- inspect columns, dimensions, and data types
- filter rows and select variables
- create derived columns such as normalized fluorescence
- summarize replicate-level data by condition
- handle missing values explicitly
- merge measurements with metadata
- save processed outputs for reproducible analysis

Most importantly, we established a convention for the rest of the book:

From here onward, tabular experimental data should be assumed to be in tidy format unless stated otherwise.

8.18 Exercises

1. Add a new column called `log10_norm_fluorescence` using base-10 logarithms. You may need the `math` module or `numpy`.
2. Extend the experiment table by adding a second time point, such as `time_h = 8`, and compute condition summaries grouped by both time and inducer.
3. Create a metadata table that includes a `promoter` column and merge it with the experiment table.
4. Starting from a wide table with columns like `Sensor_0mM_rep1` and `Sensor_1mM_rep1`, convert it into a tidy table with separate columns for strain, inducer, and replicate.

5. Save both the replicate-level tidy table and the condition-level summary table to separate CSV files in a `results/` directory.

9. Networks, Circuits, and Graphs

Synthetic biology is full of relationships.

A transcription factor activates a promoter. A repressor blocks expression of a reporter. A plasmid depends on a specific backbone. A design workflow may require one assembly step before another. A gene circuit may contain a cascade, a feedback loop, or an incoherent feed-forward motif.

Tables are still useful here, but in this chapter we will introduce another way to think about biological structure: **graphs**.

A graph is a collection of **nodes** connected by **edges**.

In synthetic biology, nodes might represent:

- genes
- proteins
- promoters
- guide RNAs
- plasmids
- strains
- assembly steps
- analysis stages

Edges represent relationships between those things.

Examples include:

- activation
- repression
- binding
- dependency
- derivation
- assembly order
- information flow

Graphs help us move from "what values are in this table?" to "how is this system connected?"

In this chapter, we will learn how to represent biological relationships in Python using **tidy tables** and **NetworkX**.

9.1 Graphs as tidy data

In Chapter 5, we introduced **tidy data** and agreed to use it as the default tabular format from that point onward.

That convention continues here.

When we represent a network in a table, we will usually use one of two tidy tables:

1. a **node table**, where each row is one node
2. an **edge table**, where each row is one edge

This is a very practical habit.

A tidy edge table is easy to:

- read from CSV
- inspect with `pandas`
- filter by interaction type
- merge with metadata
- save after curation
- convert into a graph object when needed

Likewise, a tidy node table makes it easy to attach metadata like:

- part type

- sequence length
- host organism
- plasmid name
- copy number class
- fluorescent protein color
- design status

So although we are introducing graph thinking, we are not abandoning tidy data.
We are adding a graph layer on top of it.

9.2 Installing NetworkX

The most widely used pure-Python graph library is `networkx`.
If you need to install it, the command is:

```
pip install networkx
```

We will also use `pandas`, which we already introduced in the previous chapter.

9.3 A first regulatory network

Let us begin with a simple gene regulation example.
Suppose we have a sensor circuit with these relationships:

- `AraC` activates `pBAD`
- `pBAD` drives expression of `GFP`
- `LacI` represses `pLac`
- `pLac` drives expression of `RFP`
- `TetR` represses `pTet`
- `pTet` drives expression of `LacI`

We can represent those relationships as a tidy edge table.

```
import pandas as pd

edge_table = pd.DataFrame(
    [
        {"source": "AraC", "target": "pBAD", "interaction": "activation", "sign": 1},
        {"source": "pBAD", "target": "GFP", "interaction": "expression", "sign": 1},
        {"source": "LacI", "target": "pLac", "interaction": "repression", "sign": -1},
        {"source": "pLac", "target": "RFP", "interaction": "expression", "sign": 1},
        {"source": "TetR", "target": "pTet", "interaction": "repression", "sign": -1},
        {"source": "pTet", "target": "LacI", "interaction": "expression", "sign": 1},
    ]
)

edge_table
```

	source	target	interaction	sign
0	AraC	pBAD	activation	1
1	pBAD	GFP	expression	1
2	LacI	pLac	repression	−1
3	pLac	RFP	expression	1
4	TetR	pTet	repression	−1
5	pTet	LacI	expression	1

This table is tidy because:

- each row is one interaction

- each column is one variable
- the observational unit is consistent

That is the first habit to keep.

Even when we plan to use a graph library later, it is often best to **store and exchange network data as tidy tables**.

9.4 Building a directed graph from a tidy edge table

Many biological networks are **directed**.

That means an edge has a direction.

If `AraC` regulates `pBAD`, then the edge points from `AraC` to `pBAD`. The reverse is not automatically true.

We can build a directed graph with NetworkX.

```
import networkx as nx

G = nx.from_pandas_edgelist(
    edge_table,
    source="source",
    target="target",
    edge_attr=["interaction", "sign"],
    create_using=nx.DiGraph(),
)

G
```

```
<networkx.classes.digraph.DiGraph at 0x7fe421fe5950>
```

The result is a `DiGraph`, which stands for **directed graph**.

We can inspect its basic properties.

```
G.number_of_nodes(), G.number_of_edges()
```

```
(8, 6)
```

```
sorted(G.nodes())
```

```
['AraC', 'GFP', 'LacI', 'RFP', 'TetR', 'pBAD', 'pLac', 'pTet']
```

```
list(G.edges(data=True))
```

```
[('AraC', 'pBAD', {'interaction': 'activation', 'sign': 1}),
 ('pBAD', 'GFP', {'interaction': 'expression', 'sign': 1}),
 ('LacI', 'pLac', {'interaction': 'repression', 'sign': -1}),
 ('pLac', 'RFP', {'interaction': 'expression', 'sign': 1}),
 ('TetR', 'pTet', {'interaction': 'repression', 'sign': -1}),
 ('pTet', 'LacI', {'interaction': 'expression', 'sign': 1})]
```

A graph object is useful because it lets us ask structural questions more naturally than a plain table can.

9.5 Nodes, edges, predecessors, and successors

In a directed graph:

- a node can have **incoming** edges
- a node can have **outgoing** edges

For a regulatory network, we often care about:

- what does this regulator affect?
- what controls this promoter or gene?

NetworkX gives us direct methods for that.

```
list(G.successors("AraC"))
```

```
['pBAD']
```

```
list(G.successors("pBAD"))
```

```
['GFP']
```

```
list(G.predecessors("pLac"))
```

```
['LacI']
```

This language is worth learning.

If you say that node `A` has `B` as a successor, you are saying there is a directed edge from `A` to `B`.

9.6 Using a node table for metadata

Edges tell us how nodes are connected. Node tables tell us what the nodes are.

Let us create a tidy node table.

```
node_table = pd.DataFrame(
    [
        {"node": "AraC", "kind": "protein", "role": "regulator"},
        {"node": "pBAD", "kind": "promoter", "role": "input promoter"},
        {"node": "GFP", "kind": "protein", "role": "reporter"},
        {"node": "LacI", "kind": "protein", "role": "repressor"},
        {"node": "pLac", "kind": "promoter", "role": "regulated promoter"},
        {"node": "RFP", "kind": "protein", "role": "reporter"},
        {"node": "TetR", "kind": "protein", "role": "repressor"},
        {"node": "pTet", "kind": "promoter", "role": "regulated promoter"},
    ]
)

node_table
```

	node	kind	role
0	AraC	protein	regulator
1	pBAD	promoter	input promoter
2	GFP	protein	reporter
3	LacI	protein	repressor
4	pLac	promoter	regulated promoter
5	RFP	protein	reporter
6	TetR	protein	repressor
7	pTet	promoter	regulated promoter

We can attach this metadata to the graph.

```
node_attributes = node_table.set_index("node").to_dict(orient="index")
nx.set_node_attributes(G, node_attributes)

G.nodes["GFP"]
```

```
{'kind': 'protein', 'role': 'reporter'}
```

```
G.nodes["pBAD"]
```

```
{'kind': 'promoter', 'role': 'input promoter'}
```

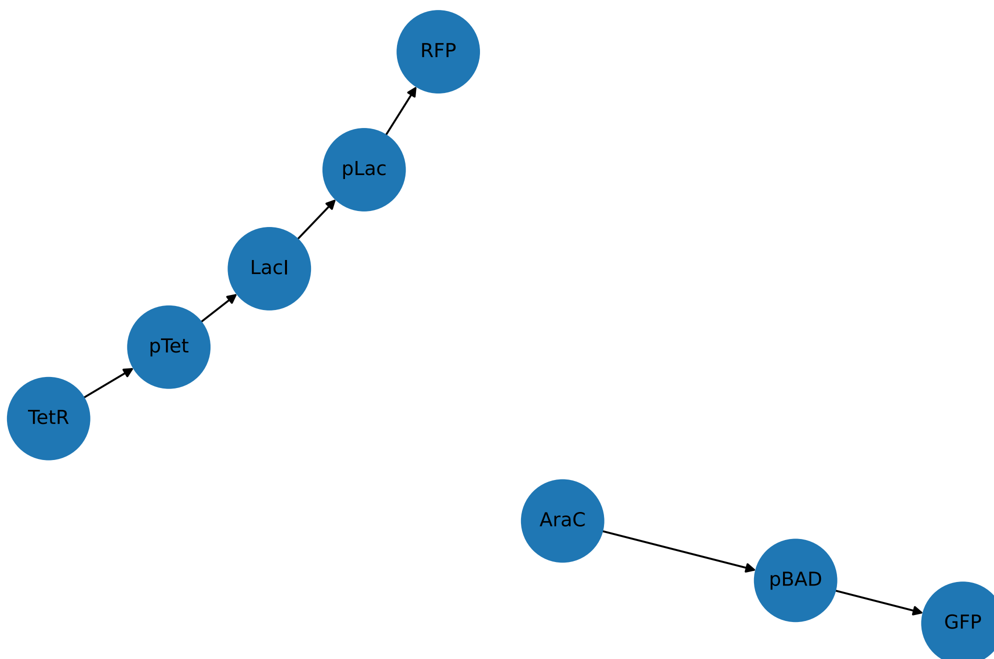
This is one of the reasons tidy node tables are powerful.

They let us curate metadata in a spreadsheet-like format, then move that metadata into a graph object for analysis.

9.7 Drawing a small circuit

Graphs can be inspected numerically, but sometimes a simple diagram helps.

```
import matplotlib.pyplot as plt

pos = nx.spring_layout(G, seed=7)

plt.figure(figsize=(8, 5))
nx.draw(
    G,
    pos,
    with_labels=True,
    node_size=1800,
    arrows=True,
    font_size=10,
)
plt.title("A small directed regulatory network")
plt.show()
```



This is not a publication-quality biological diagram, but it is very useful for quick inspection.

It answers questions like:

- are the expected nodes present?
- is the directionality correct?

- did we accidentally omit an interaction?
- does the network have disconnected parts?

For quick exploratory work, simple graph drawings are often enough.

9.8 Keeping edge metadata tidy

Because we built the graph from a tidy edge table, the interaction metadata remains accessible.

```
for source, target, data in G.edges(data=True):
    print(f"{source} -> {target}: {data['interaction']} (sign={data['sign']})")
```

```
AraC -> pBAD: activation (sign=1)
pBAD -> GFP: expression (sign=1)
LacI -> pLac: repression (sign=-1)
pLac -> RFP: expression (sign=1)
TetR -> pTet: repression (sign=-1)
pTet -> LacI: expression (sign=1)
```

A good rule is this:

- **curate in tidy tables**
- **analyze in graph objects**
- **export results back to tidy tables when useful**

That rhythm scales well.

9.9 Finding paths through a circuit

A path is a sequence of connected edges.
In synthetic biology, paths are useful when thinking about:

- signal flow through a regulatory cascade
- design dependencies
- propagation of control
- multi-step assembly or computation

Let us build a slightly richer network with a clear cascade.

```
cascade_edges = pd.DataFrame(
    [
        {"source": "Input", "target": "TF1", "interaction": "activation", "sign": 1},
        {"source": "TF1", "target": "TF2", "interaction": "activation", "sign": 1},
      {"source": "TF2", "target": "Reporter", "interaction": "activation", "sign": 1},
     {"source": "TF1", "target": "Reporter", "interaction": "repression", "sign": -1},
    ]
)

cascade = nx.from_pandas_edgelist(
    cascade_edges,
    source="source",
    target="target",
    edge_attr=["interaction", "sign"],
    create_using=nx.DiGraph(),
)

list(cascade.edges(data=True))
```

```
[('Input', 'TF1', {'interaction': 'activation', 'sign': 1}),
 ('TF1', 'TF2', {'interaction': 'activation', 'sign': 1}),
 ('TF1', 'Reporter', {'interaction': 'repression', 'sign': -1}),
 ('TF2', 'Reporter', {'interaction': 'activation', 'sign': 1})]
```

We can ask for the shortest directed path from `Input` to `Reporter`.

```
nx.shortest_path(cascade, source="Input", target="Reporter")
```

```
['Input', 'TF1', 'Reporter']
```

We can also list all simple paths.

```
list(nx.all_simple_paths(cascade, source="Input", target="Reporter"))
```

```
[['Input', 'TF1', 'TF2', 'Reporter'], ['Input', 'TF1', 'Reporter']]
```

Now we can see something interesting.

There are two routes from input to output:

- `Input -> TF1 -> Reporter`
- `Input -> TF1 -> TF2 -> Reporter`

That is already the structure of a small network motif.

9.10 Feed-forward logic

A **feed-forward loop** appears when one upstream regulator affects an output both directly and indirectly through an intermediate node.

Our example has exactly that shape.

We can inspect the signed influence of each path by multiplying the edge signs along the path.

```
def path_sign(graph, path):
    sign = 1
    for a, b in zip(path[:-1], path[1:]):
        sign *= graph.edges[a, b]["sign"]
    return sign

paths = list(nx.all_simple_paths(cascade, source="Input", target="Reporter"))

[(path, path_sign(cascade, path)) for path in paths]
```

```
[(['Input', 'TF1', 'TF2', 'Reporter'], 1), (['Input', 'TF1', 'Reporter'], -1)]
```

This is a simple but powerful idea.

Once a regulatory network is encoded as a graph with edge attributes, we can write small functions that reason about:

- activation vs repression
- path length
- redundancy
- conflicting paths
- logic motifs

That is much harder to do robustly if the structure only exists informally in our heads.

9.11 Detecting feedback loops

Many synthetic circuits and natural regulatory systems contain feedback.

Feedback can stabilize, destabilize, amplify, or oscillate depending on the system.

Let us create a tiny feedback example.

```
feedback_edges = pd.DataFrame(
    [
        {"source": "LuxR", "target": "pLux", "interaction": "activation", "sign": 1},
        {"source": "pLux", "target": "LuxI", "interaction": "expression", "sign": 1},
        {"source": "LuxI", "target": "AHL", "interaction": "synthesis", "sign": 1},
```

```
            {"source": "AHL", "target": "LuxR", "interaction": "binding_activation",
"sign": 1},
    ]
)

feedback = nx.from_pandas_edgelist(
    feedback_edges,
    source="source",
    target="target",
    edge_attr=["interaction", "sign"],
    create_using=nx.DiGraph(),
)

list(nx.simple_cycles(feedback))
```

```
[['LuxI', 'AHL', 'LuxR', 'pLux']]
```

A directed cycle is a clean structural definition of feedback.

If your graph contains a directed cycle, then some chain of influence returns to an earlier node.

That does **not** tell you the dynamics by itself, but it does tell you that feedback is structurally present.

9.12 When an adjacency matrix is useful

Although tidy edge tables are usually the best storage format, sometimes it is useful to convert a graph into a matrix.

An **adjacency matrix** records whether each node connects to each other node.

```
adjacency = nx.to_pandas_adjacency(cascade, dtype=int)
adjacency
```

	Input	TF1	TF2	Reporter
Input	0	1	0	0
TF1	0	0	1	1
TF2	0	0	0	1
Reporter	0	0	0	0

This representation is useful for:

- matrix-based methods
- exporting to certain analysis tools
- checking connectivity patterns visually
- teaching the connection between graphs and linear algebra

But for most data handling, a tidy edge table remains easier to work with.

That is why our default workflow remains:

- tidy tables for storage and curation
- graph objects for structural analysis
- matrices only when needed

9.13 Converting a graph back into a tidy edge table

Sometimes we start with a graph, perform analysis, and then want to save the result.

We can always recover a tidy edge table.

```
edge_export = nx.to_pandas_edgelist(cascade)
edge_export
```

	source	target	interaction	sign
0	Input	TF1	activation	1
1	TF1	TF2	activation	1
2	TF1	Reporter	repression	−1
3	TF2	Reporter	activation	1

That means graphs do not lock us into a special format.
They are another working representation, not the final destination.

9.14 Assembly dependencies as a directed acyclic graph

Not every graph in synthetic biology is a regulatory network.
Graphs are also useful for **workflows** and **dependencies**.
Suppose a cloning workflow has these relationships:

- the reporter cassette must be assembled before the full plasmid
- the backbone must be prepared before the full plasmid
- the plasmid must be sequence-verified before transformation
- transformation must happen before induction testing

We can encode that too.

```
dependency_table = pd.DataFrame(
    [
        {"source": "Reporter cassette", "target": "Assembled plasmid", "dependency":
"required_before"},
        {"source": "Prepared backbone", "target": "Assembled plasmid", "dependency":
"required_before"},
      {"source": "Assembled plasmid", "target": "Sequence verification", "dependency":
"required_before"},
        {"source": "Sequence verification", "target": "Transformation", "dependency":
"required_before"},
            {"source":  "Transformation",  "target":  "Induction  test",  "dependency":
"required_before"},
    ]
)

dependencies = nx.from_pandas_edgelist(
    dependency_table,
    source="source",
    target="target",
    edge_attr=["dependency"],
    create_using=nx.DiGraph(),
)

list(dependencies.edges(data=True))
```

```
[('Reporter cassette', 'Assembled plasmid', {'dependency': 'required_before'}),
 ('Assembled plasmid',
  'Sequence verification',
  {'dependency': 'required_before'}),
 ('Prepared backbone', 'Assembled plasmid', {'dependency': 'required_before'}),
 ('Sequence verification',
  'Transformation',
  {'dependency': 'required_before'}),
 ('Transformation', 'Induction test', {'dependency': 'required_before'})]
```

This graph is intended to have no directed cycles. That kind of graph is called a **directed acyclic graph**, or **DAG**.

When a graph is a DAG, we can compute a valid execution order.

```
list(nx.topological_sort(dependencies))
```

```
['Reporter cassette',
 'Prepared backbone',
 'Assembled plasmid',
 'Sequence verification',
 'Transformation',
 'Induction test']
```

This is a beautiful example of why graph thinking matters.

The same library can help with:

- regulatory structure
- workflow ordering
- assembly planning
- analysis pipelines
- data provenance

9.15 Detecting impossible workflows

If a dependency graph contains a cycle, the workflow is impossible as written.

For example, if step A depends on step B, step B depends on step C, and step C depends on step A, then nothing can start.

Let us create an intentionally broken workflow.

```
broken_dependency_table = pd.DataFrame(
    [
        {"source": "Assemble plasmid", "target": "Verify plasmid"},
        {"source": "Verify plasmid", "target": "Transform cells"},
        {"source": "Transform cells", "target": "Assemble plasmid"},
    ]
)

broken = nx.from_pandas_edgelist(
    broken_dependency_table,
    source="source",
    target="target",
    create_using=nx.DiGraph(),
)

nx.is_directed_acyclic_graph(broken)
```

```
False
```

Because this graph is not acyclic, a topological sort would fail.

That kind of check can prevent subtle mistakes in automation pipelines and project planning.

9.16 Merging tidy metadata with graph results

Because we began with tidy data, we can summarize graph structure and merge it back into tables.

For example, we can compute the in-degree and out-degree of each node in our first graph.

```
degree_summary = pd.DataFrame(
    {
        "node": list(G.nodes()),
        "in_degree": [G.in_degree(node) for node in G.nodes()],
        "out_degree": [G.out_degree(node) for node in G.nodes()],
    }
)
```

```
node_summary = node_table.merge(degree_summary, on="node", how="left")
node_summary.sort_values(["kind", "node"])
```

	node	kind	role	in_degree	out_degree
1	pBAD	promoter	input promoter	1	1
4	pLac	promoter	regulated promoter	1	1
7	pTet	promoter	regulated promoter	1	1
0	AraC	protein	regulator	0	1
2	GFP	protein	reporter	1	0
3	LacI	protein	repressor	1	1
5	RFP	protein	reporter	1	0
6	TetR	protein	repressor	0	1

This is exactly the kind of workflow that scales well:

1. store the network in tidy tables
2. convert to a graph
3. compute structural properties
4. bring those results back into tidy tables
5. continue analysis with `pandas`

That pattern will keep appearing throughout computational biology.

9.17 Choosing between a table and a graph

A good practical question is:
When should I use a table, and when should I use a graph?
Use a **tidy table** when you want to:

- store curated interactions
- edit metadata
- merge with experimental measurements
- save or exchange data
- filter and summarize observations

Use a **graph object** when you want to:

- follow paths
- detect cycles
- inspect predecessors and successors
- compute connectivity measures
- reason about motifs and dependencies

In practice, most good workflows use both.

9.18 Exercises

1. Create a tidy edge table for a repressilator-like circuit with three repressors in a cycle.
2. Convert that table into a directed graph and confirm that it contains a directed cycle.
3. Add a tidy node table with metadata such as `kind`, `host`, or `copy_number_class`.
4. Write a function that returns all direct targets of a regulator.
5. Write a function that counts how many activating and repressing edges exist in a graph.
6. Build a dependency graph for a DBTL workflow and compute a valid topological order.
7. Export one of your graphs back into a tidy edge table and save it as CSV.

9.19 Recap

In this chapter, we introduced graph thinking for synthetic biology.

The most important ideas are:

- a graph contains nodes and edges
- many biological relationships are naturally directed
- tidy **edge tables** and **node tables** are the best default storage format
- `networkx` lets us convert tidy tables into graph objects for analysis
- graph objects help us inspect paths, cycles, feedback, motifs, and workflow dependencies
- after graph analysis, we can move results back into tidy tables for further work

That final point matters a lot.

We are not replacing tidy data. We are extending it.

From here onward, when we deal with network structure, pathway logic, or design dependencies, we will still prefer tidy tables as the standard exchange format, and use graph objects as computational tools on top of them.

III

Part III - Models, Standards, and Design Automation

10 Modeling Gene Expression 103
10.1 Why model gene expression? 103
10.2 Variables, parameters, and rates 103
10.3 The simplest expression model: production and loss 104
10.4 Computing the analytical solution 104
10.5 Visualizing the approach to steady state 105
10.6 Simulating the same model with Euler's method 106
10.7 Time scales matter 108
10.8 Building a tidy summary table 109
10.9 A two-stage model: mRNA and protein 110
10.10 Simulating the two-stage model 110
10.11 Comparing promoter strengths 112
10.12 Induction and Hill functions 113
10.13 Coding Hill functions 114
10.14 Linking induction to protein production 115
10.15 Parameter sweeps should also be tidy 116
10.16 Simulating an induction time course 118
10.17 A note on deterministic vs stochastic models . 119
10.18 Choosing the right level of model complexity . 119
10.19 Exercises 120
10.20 Recap 120

11 SBOL and Tooling 121
11.1 Why not stop at FASTA or GenBank? 121
11.2 SBOL as a design language 122
11.3 The tooling layers we will use 122
11.4 Installing the packages 123
11.5 A first SBOL document with `pySBOL3` 123
11.6 What just happened? 125
11.7 Representing function, not only sequence 126
11.8 Writing the design to disk 127
11.9 Visualizing the design with VisBOL 127
11.10 Programmable visualization with DNAplotlib . 128
11.11 Using `SBOL-utilities` to reduce boilerplate . . 129
11.12 Converting older sequence formats into SBOL . 130
11.13 A practical mindset for using SBOL 132
11.14 Recommended workflow 132
11.15 Exercises 132
11.16 Recap 132

12 Design-Build-Test-Learn 135
12.1 Why DBTL matters computationally 135
12.2 One DBTL cycle as a data transformation pipeline 135
12.3 A thesis-derived software stack for closing the loop 136
12.4 Three levels of workflow closure 137
12.5 The software architecture of a closed loop . . . 137
12.6 Using LOICA at the design stage 138
12.7 LOICA as a Python lesson 140
12.8 Using PUDU at the build stage 140
12.9 PUDU as a Python lesson 141
12.10 Using Flapjack for test and learn 141
12.11 A worked conceptual loop 143
12.12 Why standards matter here 144
12.13 A practical teaching strategy 144
12.14 What students should learn from this chapter . 145
12.15 Minimal installation notes 145
12.16 Exercises 145
12.17 Closing thought 146

10. Modeling Gene Expression

Synthetic biology is not only about storing data and drawing circuits.
It is also about asking quantitative questions.

- How fast will a reporter accumulate?
- What happens if a protein degrades more quickly?
- How strongly does a promoter respond to an inducer?
- Will two conditions separate cleanly, or overlap too much to be useful?
- What parameters matter most?

Those are modeling questions.
A model is a simplified mathematical description of a biological system.
A good model does not capture everything. It captures the things that matter for the question we are asking.
In this chapter, we will build simple models of gene expression in Python.
Our goals are practical:

- translate a biological story into equations
- simulate those equations numerically
- store simulated outputs in **tidy data** format
- compare conditions and parameters systematically
- build intuition for expression dynamics, not just algebra

This chapter is not a full course in dynamical systems.
It is a working introduction for synthetic biologists who want to think quantitatively and write code that supports that thinking.

10.1 Why model gene expression?

Experiments tell us what happened.
Models help us reason about what could happen.
That matters because synthetic biology often involves design choices before an experiment is run.
For example, you may want to compare:

- a strong promoter vs a weak promoter
- a stable protein vs a destabilized one
- a low-copy plasmid vs a high-copy plasmid
- a tightly repressed promoter vs a leaky promoter
- a steep response curve vs a shallow one

A model lets us explore those alternatives quickly.
It does **not** replace experiments.
Instead, it helps with:

- intuition building
- experimental planning
- parameter sensitivity analysis
- identifying unrealistic expectations
- communicating assumptions clearly

In synthetic biology, even simple models can be extremely useful.

10.2 Variables, parameters, and rates

A model usually contains three kinds of ingredients.

10.2.1 Variables

Variables change over time.
Examples:

- mRNA concentration
- protein concentration
- inducer concentration
- cell density

10.2.2 Parameters

Parameters are fixed for one simulation.
Examples:

- transcription rate
- translation rate
- degradation rate
- Hill coefficient
- dissociation constant

10.2.3 Rules of change

Rules of change tell us how variables evolve.
For gene expression, a common idea is:

- molecules are produced
- molecules are removed or diluted

That is the core of many useful models.

10.3 The simplest expression model: production and loss

Let `P` be the amount of protein.
A minimal model says:
[= - P]
where:

- `\beta` is the production rate
- `\gamma` is the loss rate
- `P` is the protein amount

The biological story is simple.
Protein is continuously produced, but it is also lost through degradation or dilution during growth.
This model already teaches something important.
At steady state, production and loss balance each other.
If:
[= 0]
then:
[P^* =]
So the steady-state level depends on both production and loss.
A stronger promoter can increase expression, but so can a lower loss rate.

10.4 Computing the analytical solution

For the simple production-loss model, there is a closed-form solution.
If the initial amount is zero, then:
[P(t) = (1 - e^{-t})]
Let us compute that in Python.

```
import numpy as np
import pandas as pd
```

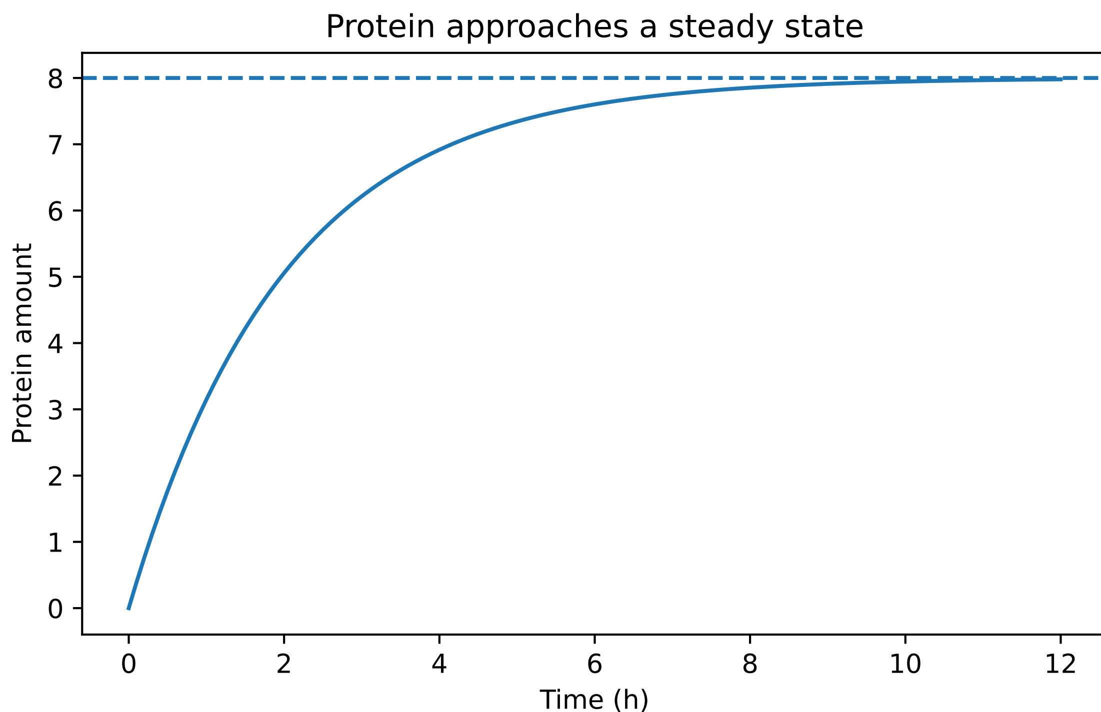
```python
import matplotlib.pyplot as plt

beta = 4.0
gamma = 0.5

time = np.linspace(0, 12, 121)
protein = beta / gamma * (1 - np.exp(-gamma * time))

analytic_df = pd.DataFrame(
    {
        "time_h": time,
        "species": "protein",
        "value": protein,
        "model": "analytic",
        "condition": "baseline",
    }
)

analytic_df.head()
```

	time_h	species	value	model	condition
0	0.0	protein	0.000000	analytic	baseline
1	0.1	protein	0.390165	analytic	baseline
2	0.2	protein	0.761301	analytic	baseline
3	0.3	protein	1.114336	analytic	baseline
4	0.4	protein	1.450154	analytic	baseline

This is a **tidy time-course table**.
Each row is one observation of one variable at one time under one condition.
That is the convention we will keep using.
Even simulated data should be stored in tidy format when possible.

10.5 Visualizing the approach to steady state

```python
plt.figure(figsize=(7, 4))
plt.plot(analytic_df["time_h"], analytic_df["value"])
plt.axhline(beta / gamma, linestyle="--")
plt.xlabel("Time (h)")
plt.ylabel("Protein amount")
plt.title("Protein approaches a steady state")
plt.show()
```



The dashed line is the steady-state value `\beta / \gamma`.
The curve rises quickly at first, then slows as loss catches up to production.
That shape appears everywhere in biology.

10.6 Simulating the same model with Euler's method

Many biological models do **not** have neat analytical solutions.
So we need a numerical method.
A very common first method is **Euler's method**.
The idea is simple.
If the current state is `P`, and the rate of change is:
[= f(P)]
then over a small time step `dt`, we approximate:
[P_{next} = P + dt f(P)]
For our model:
[P_{next} = P + dt(- P)]
Let us implement that.

```
def simulate_protein(beta, gamma, t_end=12, dt=0.1, p0=0.0, condition="baseline"):
    times = np.arange(0, t_end + dt, dt)
    p = p0
    records = []

    for t in times:
        records.append(
            {
                "time_h": t,
                "species": "protein",
                "value": p,
                "condition": condition,
                "model": "euler",
            }
        )
        dp_dt = beta - gamma * p
        p = p + dt * dp_dt
```

```
    return pd.DataFrame(records)

euler_df = simulate_protein(beta=4.0, gamma=0.5)
euler_df.head()
```

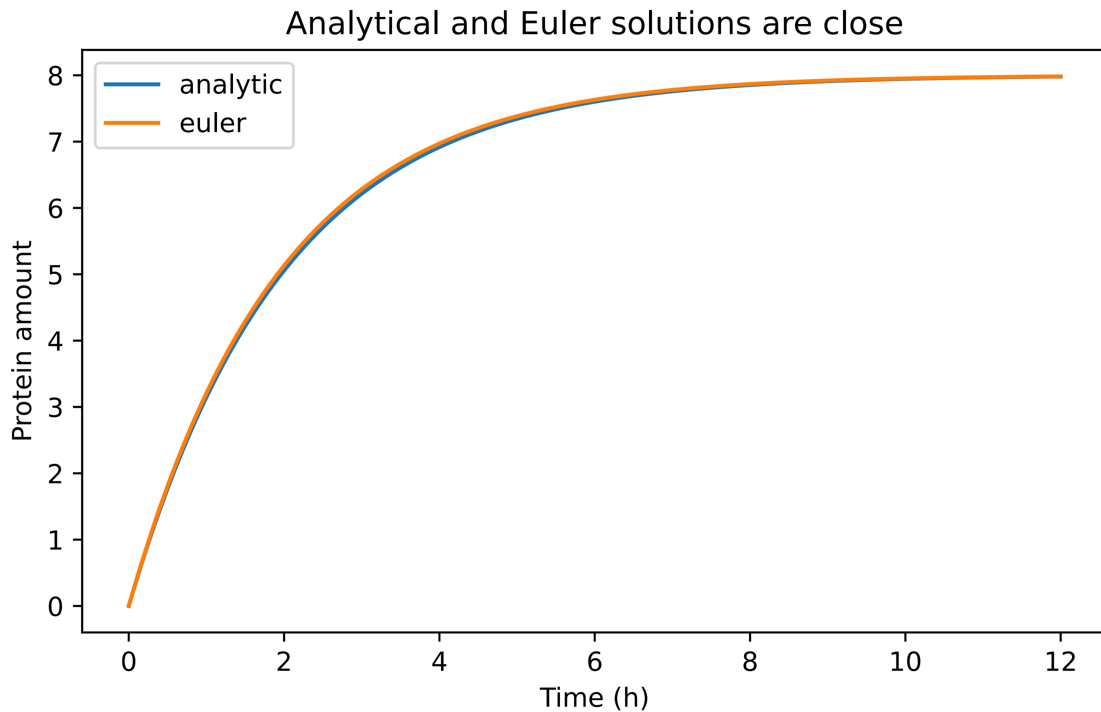
	time_h	species	value	condition	model
0	0.0	protein	0.00000	baseline	euler
1	0.1	protein	0.40000	baseline	euler
2	0.2	protein	0.78000	baseline	euler
3	0.3	protein	1.14100	baseline	euler
4	0.4	protein	1.48395	baseline	euler

We can compare the numerical and analytical solutions.

```
comparison_df = pd.concat([analytic_df, euler_df], ignore_index=True)
comparison_df.head()
```

	time_h	species	value	model	condition
0	0.0	protein	0.000000	analytic	baseline
1	0.1	protein	0.390165	analytic	baseline
2	0.2	protein	0.761301	analytic	baseline
3	0.3	protein	1.114336	analytic	baseline
4	0.4	protein	1.450154	analytic	baseline

```
plt.figure(figsize=(7, 4))

for model_name, subset in comparison_df.groupby("model"):
    plt.plot(subset["time_h"], subset["value"], label=model_name)

plt.xlabel("Time (h)")
plt.ylabel("Protein amount")
plt.title("Analytical and Euler solutions are close")
plt.legend()
plt.show()
```



For a small enough `dt`, Euler's method does a good job here.
That is one of the first practical lessons of modeling:

- the biology matters
- the mathematics matters
- the numerical method matters too

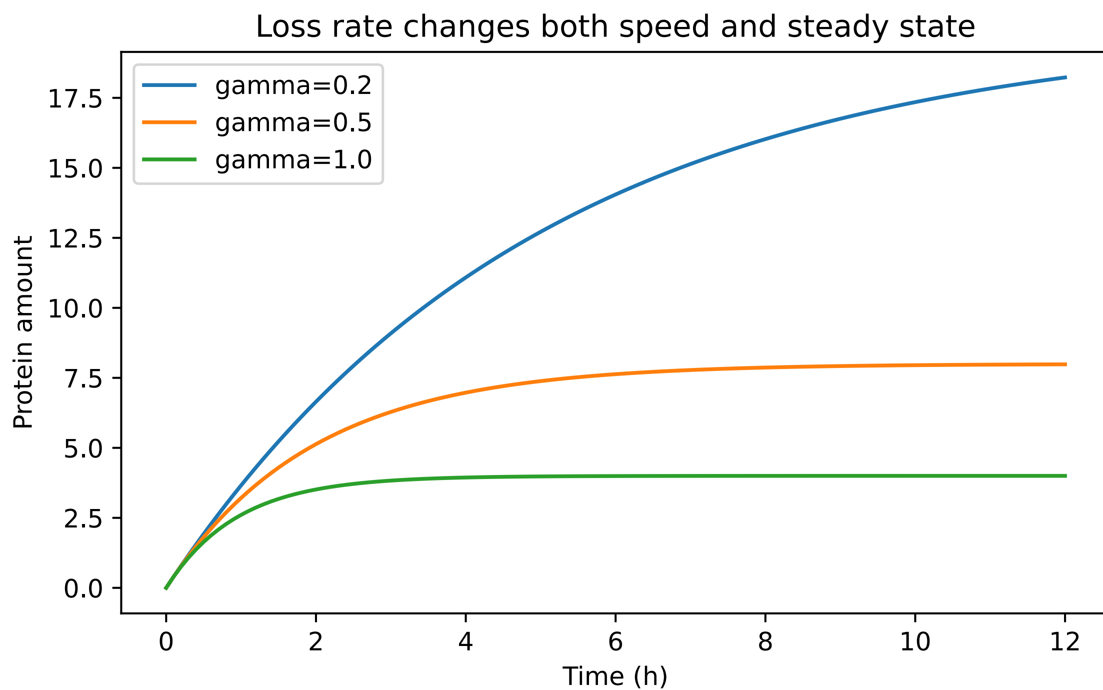
10.7 Time scales matter

The parameter `\gamma` controls how quickly the system responds.
If `\gamma` is large, the system adjusts quickly.
If `\gamma` is small, the system changes slowly.
We can compare several degradation or dilution rates.

```
rate_sweep = pd.concat(
    [
        simulate_protein(beta=4.0, gamma=0.2, condition="gamma=0.2"),
        simulate_protein(beta=4.0, gamma=0.5, condition="gamma=0.5"),
        simulate_protein(beta=4.0, gamma=1.0, condition="gamma=1.0"),
    ],
    ignore_index=True,
)

rate_sweep.head()
```

	time_h	species	value	condition	model
0	0.0	protein	0.000000	gamma=0.2	euler
1	0.1	protein	0.400000	gamma=0.2	euler
2	0.2	protein	0.792000	gamma=0.2	euler
3	0.3	protein	1.176160	gamma=0.2	euler
4	0.4	protein	1.552637	gamma=0.2	euler

```
plt.figure(figsize=(7, 4))
for condition, subset in rate_sweep.groupby("condition"):
    plt.plot(subset["time_h"], subset["value"], label=condition)

plt.xlabel("Time (h)")
plt.ylabel("Protein amount")
plt.title("Loss rate changes both speed and steady state")
plt.legend()
plt.show()
```



Notice what changed.
When `\gamma` increases:

- the system responds faster
- the steady-state level falls because `\beta / \gamma` becomes smaller

That is a powerful design idea.
Destabilizing a protein can make a system faster, but often at the cost of lower signal.

10.8 Building a tidy summary table

Because our simulations are already tidy, summarizing them is straightforward.
Let us extract the final simulated value for each condition.

```
final_values = (
    rate_sweep.sort_values("time_h")
    .groupby(["condition", "species"], as_index=False)
    .tail(1)
    .loc[:, ["condition", "species", "value"]]
    .rename(columns={"value": "final_value"})
    .reset_index(drop=True)
)

final_values
```

	condition	species	final_value
0	gamma=0.5	protein	7.983021
1	gamma=1.0	protein	3.999987
2	gamma=0.2	protein	18.229243

This is exactly why tidy data is useful after Chapter 5.
We do not need a special case for simulated results.
We can filter, group, summarize, and merge them the same way we handled experimental data.

10.9 A two-stage model: mRNA and protein

The one-variable model is useful, but gene expression usually has at least two conceptual stages:

1. transcription produces mRNA
2. translation produces protein from mRNA

A simple two-stage model is:
[= - _m m]
[= k_{tl} m - _p p]
where:

- `m` is mRNA amount
- `p` is protein amount
- `\alpha` is transcription rate
- `\delta_m` is mRNA loss rate
- `k_{tl}` is translation rate
- `\delta_p` is protein loss rate

This model helps us separate fast RNA dynamics from slower protein accumulation.

10.10 Simulating the two-stage model

```
def simulate_mrna_protein(
    alpha,
    delta_m,
    k_tl,
    delta_p,
    t_end=12,
    dt=0.05,
    m0=0.0,
    p0=0.0,
    condition="baseline",
):
    times = np.arange(0, t_end + dt, dt)
    m = m0
    p = p0
    records = []

    for t in times:
           records.append({"time_h": t, "species": "mRNA", "value": m, "condition":
condition})
         records.append({"time_h": t, "species": "protein", "value": p, "condition":
condition})

        dm_dt = alpha - delta_m * m
        dp_dt = k_tl * m - delta_p * p

        m = m + dt * dm_dt
        p = p + dt * dp_dt
```

```python
    return pd.DataFrame(records)

two_stage_df = simulate_mrna_protein(
    alpha=6.0,
    delta_m=2.0,
    k_tl=3.0,
    delta_p=0.4,
)

two_stage_df.head()
```

	time_h	species	value	condition
0	0.00	mRNA	0.00	baseline
1	0.00	protein	0.00	baseline
2	0.05	mRNA	0.30	baseline
3	0.05	protein	0.00	baseline
4	0.10	mRNA	0.57	baseline

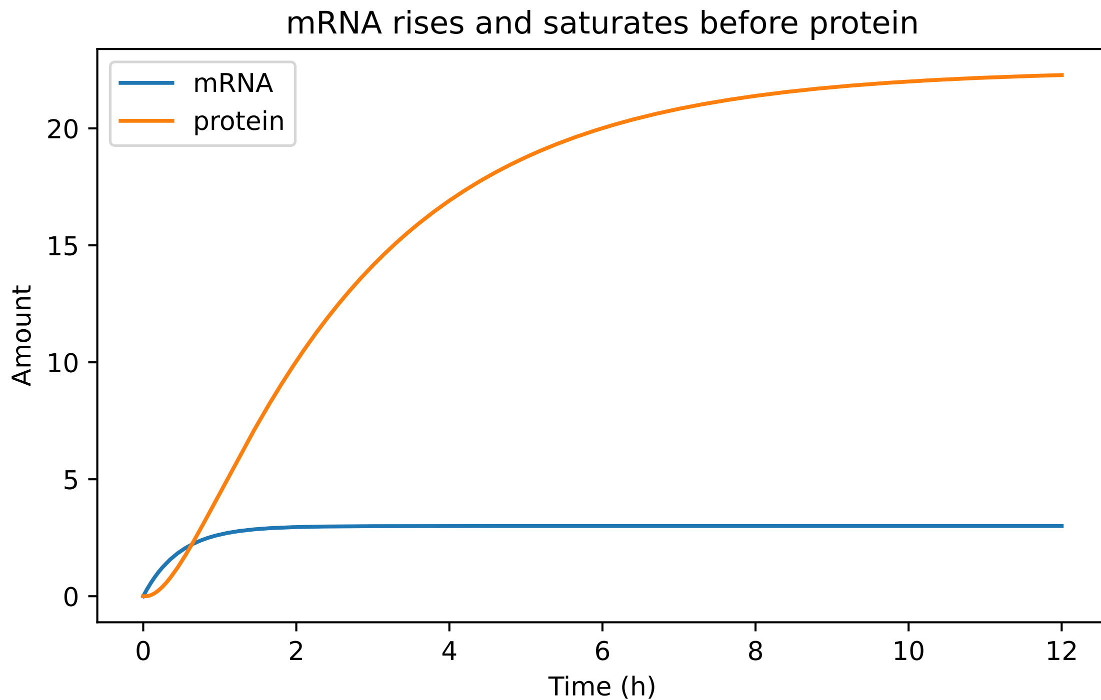
Again, this is tidy data.

Each row contains:

- one time point
- one species
- one value
- one condition

That structure is flexible enough for plotting and analysis.

```python
plt.figure(figsize=(7, 4))
for species, subset in two_stage_df.groupby("species"):
    plt.plot(subset["time_h"], subset["value"], label=species)

plt.xlabel("Time (h)")
plt.ylabel("Amount")
plt.title("mRNA rises and saturates before protein")
plt.legend()
plt.show()
```



Typically, mRNA responds faster because its turnover is faster.
Protein often lags behind.
That delay matters when designing dynamic circuits.

10.11 Comparing promoter strengths

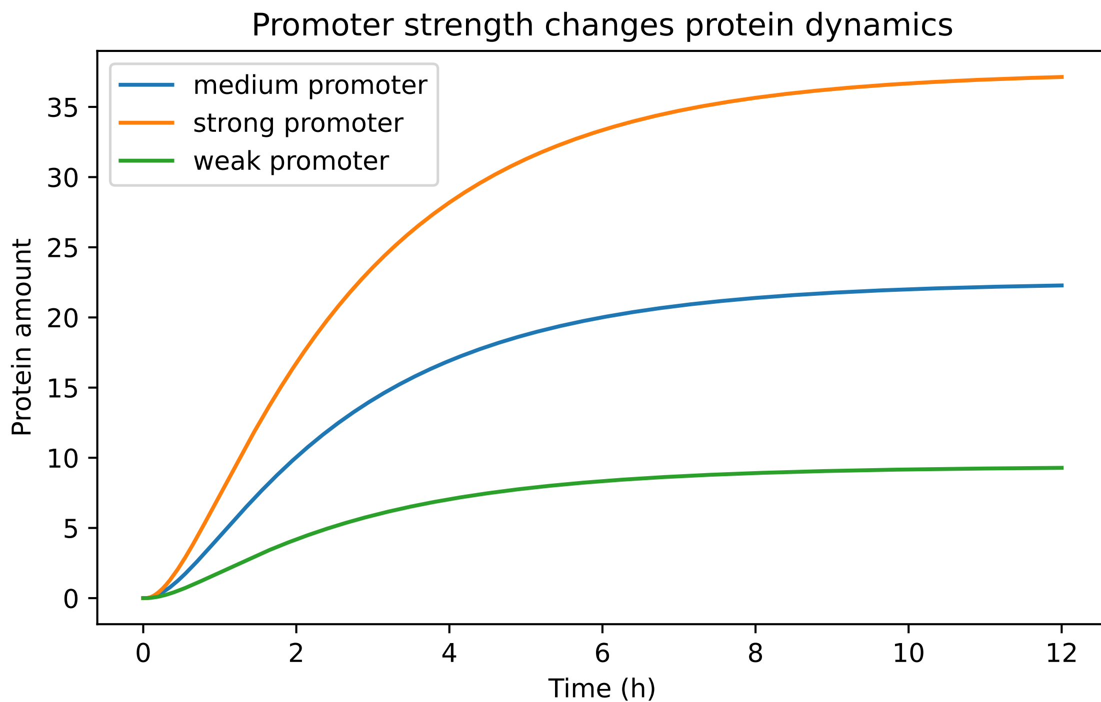
Suppose we want to compare a weak, medium, and strong promoter.
In this simple model, we can do that by changing `\alpha`, the transcription rate.

```
promoter_sweep = pd.concat(
    [
        simulate_mrna_protein(2.5, 2.0, 3.0, 0.4, condition="weak promoter"),
        simulate_mrna_protein(6.0, 2.0, 3.0, 0.4, condition="medium promoter"),
        simulate_mrna_protein(10.0, 2.0, 3.0, 0.4, condition="strong promoter"),
    ],
    ignore_index=True,
)

protein_only = promoter_sweep[promoter_sweep["species"] == "protein"]
protein_only.head()
```

	time_h	species	value	condition
1	0.00	protein	0.000000	weak promoter
3	0.05	protein	0.000000	weak promoter
5	0.10	protein	0.018750	weak promoter
7	0.15	protein	0.054000	weak promoter
9	0.20	protein	0.103733	weak promoter

```
plt.figure(figsize=(7, 4))
for condition, subset in protein_only.groupby("condition"):
    plt.plot(subset["time_h"], subset["value"], label=condition)
```

```
plt.xlabel("Time (h)")
plt.ylabel("Protein amount")
plt.title("Promoter strength changes protein dynamics")
plt.legend()
plt.show()
```



This is a very common modeling use case.

Before building three constructs, we can explore whether the expected response differences are likely to be large enough to matter.

10.12 Induction and Hill functions

So far, production has been constant.

But many synthetic biology systems are regulated.

A promoter may respond to an inducer, a repressor, or an activator.

A common approximation is the **Hill function**.

10.12.1 Activation

A simple activation function is:

[(x) =]

where:

- `x` is inducer or activator concentration
- `K` is the half-response concentration
- `n` is the Hill coefficient

10.12.2 Repression

A simple repression function is:

[(x) =]

These functions are not perfect descriptions of every promoter.

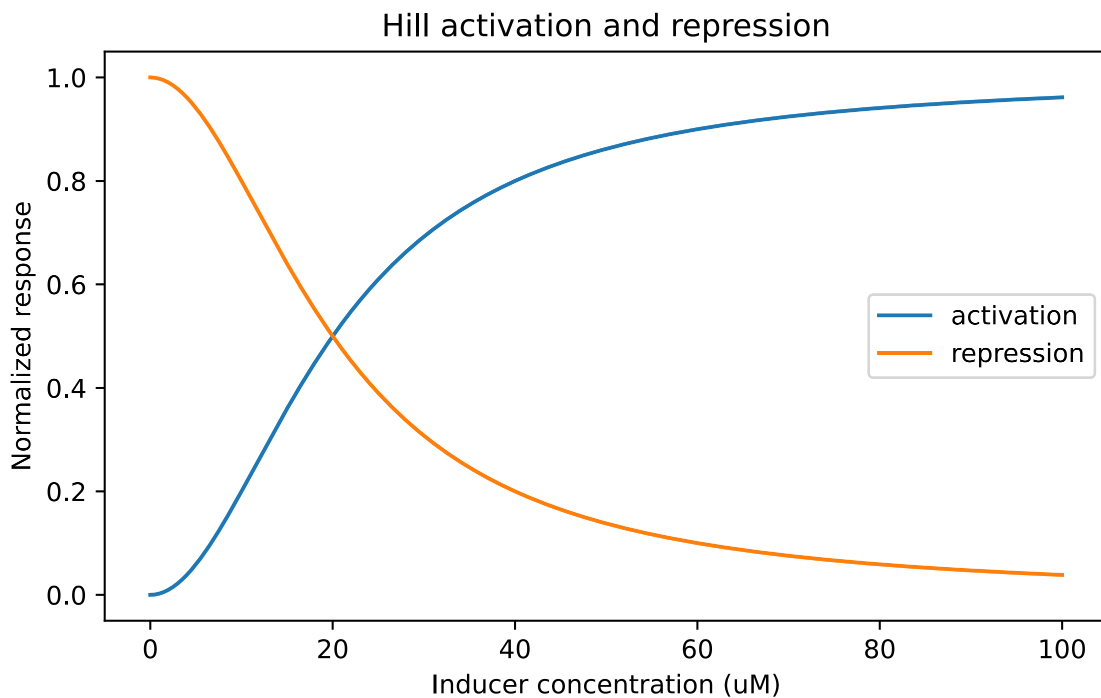
But they are extremely useful approximations.

10.13 Coding Hill functions

```
def hill_activation(x, K, n):
    x = np.asarray(x, dtype=float)
    return (x**n) / (K**n + x**n)


def hill_repression(x, K, n):
    x = np.asarray(x, dtype=float)
    return (K**n) / (K**n + x**n)

inducer = np.linspace(0, 100, 201)
activation = hill_activation(inducer, K=20, n=2)
repression = hill_repression(inducer, K=20, n=2)

hill_df = pd.DataFrame(
    {
        "inducer_uM": np.concatenate([inducer, inducer]),
        "response": np.concatenate([activation, repression]),
        "relationship": ["activation"] * len(inducer) + ["repression"] * len(inducer),
    }
)

hill_df.head()
```

	inducer_uM	response	relationship
0	0.0	0.000000	activation
1	0.5	0.000625	activation
2	1.0	0.002494	activation
3	1.5	0.005594	activation
4	2.0	0.009901	activation

```
plt.figure(figsize=(7, 4))
for relationship, subset in hill_df.groupby("relationship"):
    plt.plot(subset["inducer_uM"], subset["response"], label=relationship)

plt.xlabel("Inducer concentration (uM)")
plt.ylabel("Normalized response")
plt.title("Hill activation and repression")
plt.legend()
plt.show()
```



A Hill coefficient of `n = 1` gives a gentler curve.
A larger `n` makes the response steeper.
That can matter a lot when designing threshold-like behavior.

10.14 Linking induction to protein production

We can plug a Hill activation function into our expression model.

Suppose a promoter has a maximum transcription rate `alpha_max`, and induction controls what fraction of that maximum is active.

Then we might write:

[(x) = _{max}]

Let us compute a steady-state dose-response curve for protein.

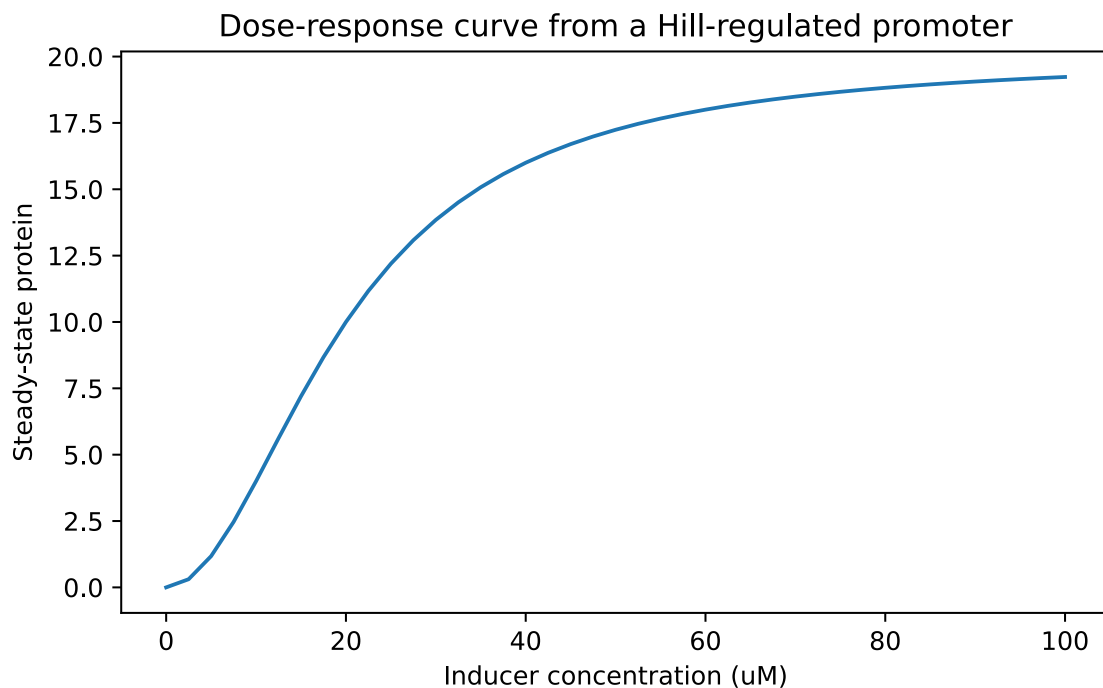
In the simple production-loss model, steady state is `P^* = \beta / \gamma`.

If we treat `\beta` as an inducer-dependent production term, we can sweep across inducer values.

```
alpha_max = 12.0
gamma = 0.6
inducer_values = np.linspace(0, 100, 41)

response_records = []
for x in inducer_values:
    beta_x = alpha_max * hill_activation(x, K=20, n=2)
    p_ss = beta_x / gamma
    response_records.append(
        {
            "inducer_uM": x,
            "steady_state_protein": p_ss,
            "K": 20,
            "hill_n": 2,
        }
    )

dose_response_df = pd.DataFrame(response_records)
dose_response_df.head()
```

	inducer_uM	steady_state_protein	K	hill_n
0	0.0	0.000000	20	2
1	2.5	0.307692	20	2
2	5.0	1.176471	20	2
3	7.5	2.465753	20	2
4	10.0	4.000000	20	2

```
plt.figure(figsize=(7, 4))
plt.plot(dose_response_df["inducer_uM"], dose_response_df["steady_state_protein"])
plt.xlabel("Inducer concentration (uM)")
plt.ylabel("Steady-state protein")
plt.title("Dose-response curve from a Hill-regulated promoter")
plt.show()
```



This is a classic synthetic biology use case.
You can think of the model as a way to connect:

- inducer concentration
- promoter activity
- expression level

10.15 Parameter sweeps should also be tidy

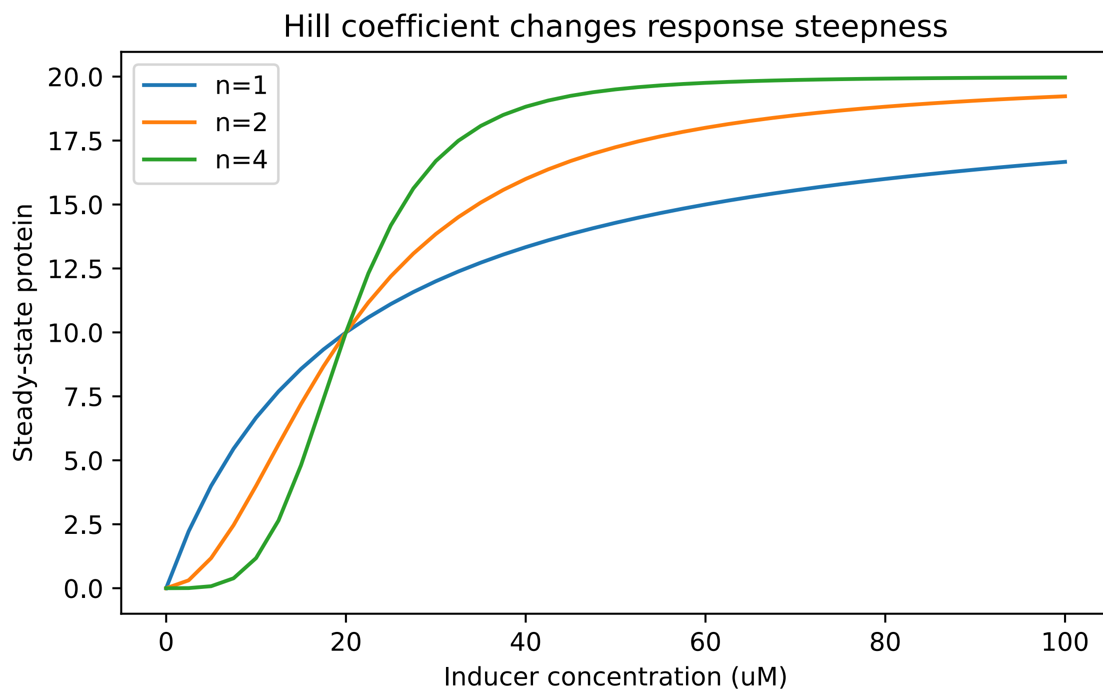
From this point onward, parameter sweeps should be stored tidily too.
Let us compare different Hill coefficients.

```
parameter_sweep_records = []

for n_value in [1, 2, 4]:
    for x in inducer_values:
        beta_x = alpha_max * hill_activation(x, K=20, n=n_value)
        p_ss = beta_x / gamma
        parameter_sweep_records.append(
```

```
            {
                "inducer_uM": x,
                "steady_state_protein": p_ss,
                "hill_n": n_value,
                "K": 20,
                "parameter": "hill_n",
            }
        )

parameter_sweep_df = pd.DataFrame(parameter_sweep_records)
parameter_sweep_df.head()
```

	inducer_uM	steady_state_protein	hill_n	K	parameter
0	0.0	0.000000	1	20	hill_n
1	2.5	2.222222	1	20	hill_n
2	5.0	4.000000	1	20	hill_n
3	7.5	5.454545	1	20	hill_n
4	10.0	6.666667	1	20	hill_n

```
plt.figure(figsize=(7, 4))
for hill_n, subset in parameter_sweep_df.groupby("hill_n"):
                   plt.plot(subset["inducer_uM"],     subset["steady_state_protein"],
label=f"n={hill_n}")

plt.xlabel("Inducer concentration (uM)")
plt.ylabel("Steady-state protein")
plt.title("Hill coefficient changes response steepness")
plt.legend()
plt.show()
```



Now the sweep lives in a tidy table, so we can summarize it easily.
For example, what inducer concentration first reaches at least half-maximal output?

```
half_max_summary = []
max_output = parameter_sweep_df["steady_state_protein"].max()

for hill_n, subset in parameter_sweep_df.groupby("hill_n"):
    threshold = subset["steady_state_protein"].max() / 2
    above = subset[subset["steady_state_protein"] >= threshold]
    first_crossing = above["inducer_uM"].min()
    half_max_summary.append({"hill_n": hill_n, "half_max_inducer_uM": first_crossing})

half_max_df = pd.DataFrame(half_max_summary)
half_max_df
```

	hill_n	half_max_inducer_uM
0	1	15.0
1	2	20.0
2	4	20.0

10.16 Simulating an induction time course

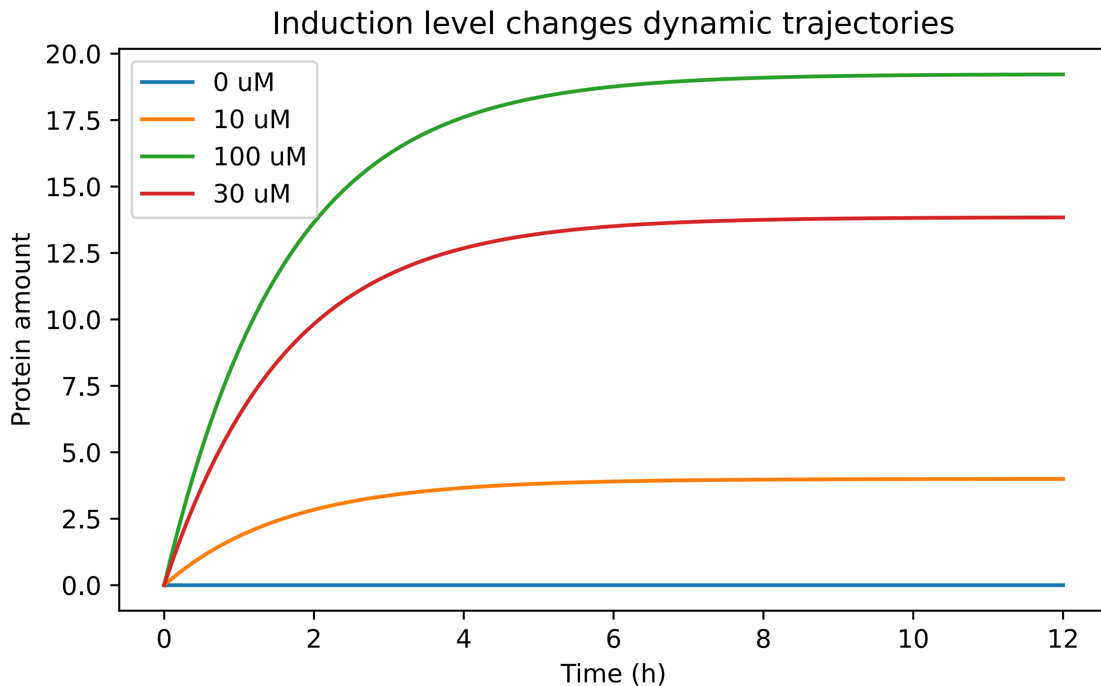
A dose-response curve tells us steady-state behavior.
But sometimes we care about dynamics after induction.
We can simulate a protein model where the production rate depends on inducer concentration.

```
def simulate_induced_protein(inducer_uM, alpha_max, K, n, gamma, t_end=12, dt=0.1,
condition=None):
    if condition is None:
        condition = f"inducer={inducer_uM}"

    beta = alpha_max * hill_activation(inducer_uM, K=K, n=n)
        return simulate_protein(beta=beta, gamma=gamma, t_end=t_end, dt=dt,
condition=condition)

induction_timecourse_df = pd.concat(
    [
        simulate_induced_protein(0, 12.0, 20, 2, 0.6, condition="0 uM"),
        simulate_induced_protein(10, 12.0, 20, 2, 0.6, condition="10 uM"),
        simulate_induced_protein(30, 12.0, 20, 2, 0.6, condition="30 uM"),
        simulate_induced_protein(100, 12.0, 20, 2, 0.6, condition="100 uM"),
    ],
    ignore_index=True,
)

induction_timecourse_df.head()
```

	time_h	species	value	condition	model
0	0.0	protein	0.0	0 uM	euler
1	0.1	protein	0.0	0 uM	euler
2	0.2	protein	0.0	0 uM	euler
3	0.3	protein	0.0	0 uM	euler
4	0.4	protein	0.0	0 uM	euler

```
plt.figure(figsize=(7, 4))
for condition, subset in induction_timecourse_df.groupby("condition"):
```

```
    plt.plot(subset["time_h"], subset["value"], label=condition)

plt.xlabel("Time (h)")
plt.ylabel("Protein amount")
plt.title("Induction level changes dynamic trajectories")
plt.legend()
plt.show()
```



This kind of result helps answer practical design questions.
For example:

- is basal expression acceptably low?
- does the induced state separate enough from the uninduced state?
- how long must we wait before measuring fluorescence?

10.17 A note on deterministic vs stochastic models

Everything we have done so far is **deterministic**.
That means the model gives the same answer every time for the same parameters and initial conditions.
Real gene expression is often noisy.
At low copy number, stochastic effects can matter a lot.
So why start with deterministic models?
Because they are often the right first step.
They help us understand:

- the average behavior
- the role of parameters
- the structure of a design
- whether a system is even plausible before adding more realism

Later, you may want stochastic simulation, Bayesian inference, or parameter estimation from data.
But deterministic models are the right foundation.

10.18 Choosing the right level of model complexity

Not every project needs the same model.
A useful rule of thumb is:

- start with the simplest model that can answer your question
- add complexity only when the simpler model fails meaningfully

For example:
Use a one-variable production-loss model when you care about:

- rough accumulation times
- steady-state scaling
- promoter strength comparisons

Use a two-stage mRNA-protein model when you care about:

- transcription vs translation separately
- delays between RNA and protein
- RNA instability

Use regulated production with Hill functions when you care about:

- induction curves
- repression curves
- threshold behavior
- dose-response tuning

A model is not better because it has more equations.
A model is better when it is better matched to the question.

10.19 Exercises

1. Change the degradation rate `gamma` in the one-variable model and measure how long it takes to reach 90% of steady state.
2. Modify the two-stage model so that the initial mRNA amount is nonzero. How does that change early protein accumulation?
3. Compare two proteins with the same production rate but different degradation rates. Which one is faster? Which one reaches a higher steady state?
4. Repeat the Hill-function sweep with different values of `K`. How does `K` shift the response curve?
5. Build a tidy parameter sweep over both `K` and `n`, and summarize the final steady-state protein values.
6. Add a constant leak term to the induced production model and examine basal expression.
7. Save one of your simulated tidy tables to CSV for later analysis.

10.20 Recap

In this chapter, we introduced practical modeling of gene expression in Python.
The most important ideas are:

- models turn biological stories into quantitative rules
- production-loss models already explain steady states and response times
- Euler's method lets us simulate models numerically
- simulated outputs should be stored in **tidy data** format just like experimental data
- two-stage mRNA-protein models add biological realism while staying manageable
- Hill functions are a practical way to model activation and repression
- tidy parameter sweeps make it easy to compare design choices systematically

From this point onward, we will treat simulated tables the same way we treat experimental ones:

- each row should represent one observation
- variables should live in columns
- conditions and parameters should be explicit
- tidy data remains the default format for downstream analysis

That consistency is important.
It means the same Python tools can support both modeling and experiments, which is exactly what we want in synthetic biology.

11. SBOL and Tooling

Synthetic biology is not only about sequences.

It is also about **design exchange**.

A FASTA file can tell us what the bases are. A GenBank record can tell us more, including annotations and sequence features. But engineered biological systems are usually more than a sequence record.

We often need to represent:

- what the parts are
- how they are composed
- what roles they play
- what interactions they participate in
- what constraints or assumptions define the design
- how a design moves between tools in the design-build-test-learn cycle

That is where **SBOL**, the **Synthetic Biology Open Language**, becomes important.

For synthetic biology, SBOL should be the default exchange format whenever we care about both **structure** and **function**, not just raw sequence. FASTA and GenBank are still useful, but they are best treated as sequence-oriented formats, not as the canonical representation of an engineered design.

In this chapter, we will do four things:

- understand why SBOL matters
- learn the core object model of **pySBOL3**
- use **SBOL-utilities** to reduce boilerplate and bridge older sequence formats into SBOL
- visualize designs with **VisBOL** and **DNAplotlib**

By the end, you should see SBOL not as an abstract standard, but as a practical data structure for Python-based synthetic biology and a natural foundation for standardized design visualization.

11.1 Why not stop at FASTA or GenBank?

FASTA is extremely simple.

That simplicity is also its limitation.

A FASTA record usually gives us an identifier and a sequence. That is enough for alignment, primer design, BLAST searches, and many sequence-processing tasks. It is not enough to represent a complete design intent.

GenBank is richer.

A GenBank file can include annotated features such as promoters, coding sequences, and terminators. That makes it much more informative than FASTA for sequence-centric work. But GenBank is still centered on the idea of a sequence record with annotations layered on top.

Synthetic biology often needs more than that.

We may want to represent that a promoter regulates a coding sequence, that a protein inhibits another component, that a construct is part of a larger system, or that a design is intended to move into another software tool for simulation, build planning, repository upload, or metadata capture.

SBOL was developed for exactly this problem.

SBOL is meant to standardize how designs are represented so that tools can exchange information without inventing a new private format for each project.

A useful practical rule is this:

- use **FASTA** when you only need sequence strings
- use **GenBank** when you need sequence plus familiar annotations
- use **SBOL** when you need a standardized representation of engineered biological design, especially when **structure, function, composition, and interoperability** all matter

11.2 SBOL as a design language

One helpful way to think about SBOL is that it gives us a common language for design objects.
Instead of passing around only strings, we can pass around structured objects such as:

- a DNA component
- a protein component
- a promoter role
- a transcriptional unit as an engineered region
- interactions such as repression or production
- documents that collect many related design objects together

This matters because software becomes much easier to connect when tools agree on the meaning of those objects.
That is why SBOL shows up naturally in standards-driven workflows, repositories such as SynBioHub, and design automation tools.

11.3 The tooling layers we will use

In this chapter we will use two related packages.

11.3.1 `pySBOL3`

This is the core Python interface to SBOL version 3.
It gives us the low-level object model directly:

- `Document`
- `Component`
- `Sequence`
- `SubComponent`
- `Interaction`
- `Participation`
- `Constraint`

When you want precise control, `pySBOL3` is the main tool.

11.3.2 `SBOL-utilities`

This package sits one layer higher.
It provides helper functions for common operations, including:

- creating standard biological parts
- assembling simple engineered regions
- converting between formats such as FASTA, GenBank, and SBOL
- supporting common synthetic biology workflows

In practice, many projects use both layers together.
Use `pySBOL3` when you want explicit control over the data model. Use `SBOL-utilities` when you want convenience and interoperability helpers.

11.3.3 VisBOL

VisBOL is useful when your source of truth is already an SBOL design and you want to inspect or communicate that design visually using standardized glyphs.
It is especially good for:

- quickly checking whether a design looks structurally right
- rendering SBOL-native diagrams without manually defining shapes
- exporting clean figures for notes, talks, or manuscripts

11.3.4 DNAplotlib

DNAplotlib is useful when your source of truth is a Python analysis workflow and you want fine-grained control over how the diagram is drawn.

It is especially good for:

- building publication-style figures programmatically
- overlaying or aligning design diagrams with analysis outputs
- customizing colors, labels, part geometry, and regulation arcs

A good working habit is to think of these tools as complementary rather than competing.

- **SBOL + VisBOL** is a strong path for standardized design exchange and standardized visualization
- **Python objects + DNAplotlib** is a strong path for flexible figure generation inside analysis notebooks and scripts

11.4 Installing the packages

A minimal installation looks like this:

```
pip install sbol3 sbol-utilities biopython
```

If you also want to reproduce the DNAplotlib examples later in the chapter, install it as well:

```
pip install dnaplotlib
```

VisBOL is typically used as a separate visualization tool rather than as a Python package inside the same notebook workflow, so we will treat it as an external viewer for SBOL files.

11.5 A first SBOL document with `pySBOL3`

We will start with a constitutive GFP transcriptional unit.

The goal is not biological realism in every nucleotide. The goal is to understand the data model.

We will create:

- a promoter
- an RBS
- a coding sequence
- a terminator
- an engineered region that contains them in order

```python
from pathlib import Path

import pandas as pd
import sbol3

artifacts = Path("outputs/ch08")
artifacts.mkdir(parents=True, exist_ok=True)

sbol3.set_namespace("https://example.org/python-for-synthetic-biology")

doc = sbol3.Document()


def make_dna_component(doc, name, role, seq_text):
    component = sbol3.Component(name, sbol3.SBO_DNA, roles=[role])
    sequence = sbol3.Sequence(
        f"{name}_seq",
        elements=seq_text,
        encoding=sbol3.IUPAC_DNA_ENCODING,
    )
    component.sequences = [sequence]
    doc.add(sequence)
```

```
    doc.add(component)
    return component


promoter = make_dna_component(
    doc,
    "pConst",
    sbol3.SO_PROMOTER,
    "ttgacagctagctcagtcctaggtataatgctagc",
)
rbs = make_dna_component(doc, "BCD2", sbol3.SO_RBS, "aaaggagg")
gfp = make_dna_component(doc, "GFP", sbol3.SO_CDS, "atggtgagcaagggcgaggag")
terminator = make_dna_component(
    doc,
    "T1",
    sbol3.SO_TERMINATOR,
    "tttttattgctagttattgctagc",
)

tu = sbol3.Component(
    "TU_constitutive_gfp",
    sbol3.SBO_DNA,
    roles=[sbol3.SO_ENGINEERED_REGION],
)

for part in [promoter, rbs, gfp, terminator]:
    tu.features.append(sbol3.SubComponent(part))

for left, right in zip(tu.features[:-1], tu.features[1:]):
    tu.constraints.append(sbol3.Constraint(sbol3.SBOL_PRECEDES, left, right))

doc.add(tu)

manual_path = artifacts / "manual_tu.nt"
doc.write(manual_path, sbol3.SORTED_NTRIPLES)

component_inventory = pd.DataFrame(
    {
       "display_id": [obj.display_id for obj in [promoter, rbs, gfp, terminator, tu]],
        "type": ["DNA", "DNA", "DNA", "DNA", "DNA region"],
        "role": [
            "promoter",
            "RBS",
            "CDS",
            "terminator",
            "engineered region",
        ],
    }
)

component_inventory
```

	display_id	type	role
0	pConst	DNA	promoter
1	BCD2	DNA	RBS
2	GFP	DNA	CDS
3	T1	DNA	terminator

	display_id	type	role
4	TU_constitutive_gfp	DNA region	engineered region

One pattern emphasized in introductory SBOL tutorials is that a `Document` is not a black box. You should get used to inspecting its contents early and often.

```
top_level_inventory = pd.DataFrame(
    {
        "display_id": [getattr(obj, "display_id", None) for obj in doc.objects],
        "python_class": [type(obj).__name__ for obj in doc.objects],
        "identity": [obj.identity for obj in doc.objects],
    }
)

top_level_inventory
```

	display_id	python_class	identity
0	pConst_seq	Sequence	https://example.org/python-for-synthetic-biolo...
1	pConst	Component	https://example.org/python-for-synthetic-biolo...
2	BCD2_seq	Sequence	https://example.org/python-for-synthetic-biolo...
3	BCD2	Component	https://example.org/python-for-synthetic-biolo...
4	GFP_seq	Sequence	https://example.org/python-for-synthetic-biolo...
5	GFP	Component	https://example.org/python-for-synthetic-biolo...
6	T1_seq	Sequence	https://example.org/python-for-synthetic-biolo...
7	T1	Component	https://example.org/python-for-synthetic-biolo...
8	TU_constitutive_gfp	Component	https://example.org/python-for-synthetic-biolo...

That inspection step is extremely useful when you are learning the model or debugging a larger document exported from another tool.

That table is a **tidy inventory** of the main design objects: one row per object, one column per variable.

The important point is not the exact nucleotide sequence. It is the fact that the design is now represented by structured SBOL objects instead of a single anonymous string.

11.6 What just happened?

Several SBOL ideas appeared in a compact example.

The coding style above is close to the pattern used in many introductory `pySBOL3` tutorials:

1. set a namespace
2. create a `Document`
3. build top-level objects like `Component` and `Sequence`
4. connect them through features, references, and constraints
5. inspect the resulting document before writing it to disk

That sequence of steps is worth internalizing because it scales from toy examples to larger design libraries.

11.6.1 `Document`

A `Document` is the container for SBOL objects.

It is the thing you read from disk, write to disk, and pass between tools.

11.6.2 `Component`

A `Component` represents a biological design object.

Here, our promoter, RBS, CDS, terminator, and complete transcriptional unit are all components.

11.6.3 `Sequence`

A `Sequence` stores the actual sequence text.

This matters conceptually. The sequence is not the same thing as the design object. A component may refer to a sequence, but the component also carries type and role information.

11.6.4 `SubComponent`

A `SubComponent` says that one component occurs inside another.

That is how the transcriptional unit contains the promoter, RBS, CDS, and terminator.

11.6.5 `Constraint`

A `Constraint` lets us say that one part precedes another.

That is how we capture order in the engineered region.

This is already a major step beyond FASTA. We are not only storing bases. We are storing a design structure.

11.7 Representing function, not only sequence

SBOL is especially valuable when we go beyond sequence layout and start representing function.

Here is a minimal example of a sensor-like design where:

- a protein `LacI` is represented explicitly
- a promoter and coding region are placed inside a system
- interactions are added to state repression and genetic production

This is not yet a full mechanistic model. It is a structured functional description.

```
laci = sbol3.Component("LacI", sbol3.SBO_PROTEIN)
doc.add(laci)

sensor = sbol3.Component("lac_sensor", sbol3.SBO_FUNCTIONAL_ENTITY)

sensor_promoter = sbol3.SubComponent(promoter)
sensor_output = sbol3.SubComponent(gfp)

sensor.features.extend([sensor_promoter, sensor_output])
sensor.constraints.append(sbol3.Constraint(sbol3.SBOL_PRECEDES,      sensor_promoter,
sensor_output))

repression = sbol3.Interaction(
    sbol3.SBO_INHIBITION,
    participations=[
        sbol3.Participation([sbol3.SBO_INHIBITOR], laci),
        sbol3.Participation([sbol3.SBO_INHIBITED], sensor_promoter),
    ],
)

production = sbol3.Interaction(
    sbol3.SBO_GENETIC_PRODUCTION,
    participations=[
        sbol3.Participation([sbol3.SBO_TEMPLATE], sensor_output),
        sbol3.Participation([sbol3.SBO_PRODUCT], laci),
    ],
)

sensor.interactions.extend([repression, production])
doc.add(sensor)

interaction_table = pd.DataFrame(
    {
```

```
            "interaction_type": [i.types[0].split(":")[-1] if ":" in i.types[0] else
i.types[0] for i in sensor.interactions],
         "n_participants": [len(i.participations) for i in sensor.interactions],
    }
)

interaction_table
```

	interaction_type	n_participants
0	0000169	2
1	0000589	2

Now we have crossed the line from **annotated sequence** into **design semantics**.

That is the key educational leap of SBOL.

You are no longer asking only, “what is this sequence?” You are also asking, “what role does this object play?” and “how does it relate to other objects in the design?”

11.8 Writing the design to disk

An SBOL document can be serialized to disk in RDF-based formats.

```
sensor_path = artifacts / "sensor_design.nt"
doc.write(sensor_path, sbol3.SORTED_NTRIPLES)

{
    "file": str(sensor_path),
    "exists": sensor_path.exists(),
    "n_top_level_objects": len(list(doc.objects)),
}
```

```
{'file': 'outputs/ch08/sensor_design.nt',
 'exists': True,
 'n_top_level_objects': 11}
```

The exact serialization format is less important than the principle.

Once a design is encoded as SBOL, it can be:

- stored in a repository
- exchanged across tools
- inspected programmatically
- enriched with more structure or metadata later

11.9 Visualizing the design with VisBOL

Once a design exists as an SBOL document, the simplest visualization workflow is often to open that file in a tool that already understands SBOL semantics.

That is the role of VisBOL.

A practical workflow looks like this:

1. build or export an SBOL document from Python
2. write it to disk in an SBOL serialization format
3. load that file into VisBOL
4. inspect whether the structure, orientation, and composition match your intent
5. export a figure when you want a quick standards-oriented diagram

In other words, VisBOL is best thought of as a **viewer and renderer for SBOL-native designs**.

If the design file is already the source of truth, this is often the fastest path from model to figure.

```
visbol_ready_path = artifacts / "visbol_ready_design.nt"
doc.write(visbol_ready_path, sbol3.SORTED_NTRIPLES)

{
    "file_for_visbol": str(visbol_ready_path),
    "exists": visbol_ready_path.exists(),
}
```

```
{'file_for_visbol': 'outputs/ch08/visbol_ready_design.nt', 'exists': True}
```

This chunk is intentionally simple.

The key idea is that VisBOL does not require us to redraw the design by hand. It consumes the standardized SBOL representation directly.

11.10 Programmable visualization with DNAplotlib

Sometimes standardized viewing is not enough.

You may want to:

- match a figure style used in a paper
- control colors and labels precisely
- line up a design diagram with experimental plots
- render many design variants inside the same Python workflow

That is where DNAplotlib becomes useful.

Where VisBOL starts from an SBOL file, DNAplotlib usually starts from a Python description of the design to be drawn. The common pattern is to define a list of part dictionaries and then render them with a `DNARenderer`.

The example below is marked as not executed because DNAplotlib is an optional dependency and may not be installed in every environment. The important thing is to see the workflow.

```python
import matplotlib.pyplot as plt
import dnaplotlib as dpl

design = [
    {"type": "Promoter", "name": "pTet", "fwd": True, "opts": {"label": "pTet"}},
    {"type": "RBS", "name": "BCD2", "fwd": True},
    {"type": "CDS", "name": "GFP", "fwd": True, "opts": {"label": "GFP"}},
    {"type": "Terminator", "name": "T1", "fwd": True},
]

regulations = [
    {"type": "Repression", "from_part": 2, "to_part": 0, "opts": {"label": "LacI"}},
]

dr = dpl.DNARenderer()
part_renderers = dr.SBOL_part_renderers()
reg_renderers = dr.std_reg_renderers()

fig, ax = plt.subplots(figsize=(10, 2))
start, end = dr.renderDNA(
    ax,
    design,
    part_renderers,
    regs=regulations,
    reg_renderers=reg_renderers,
)

ax.set_xlim([start - 10, end + 10])
ax.set_ylim([-25, 25])
ax.set_aspect("equal")
```

```
ax.axis("off")
fig.tight_layout()
plt.show()
```

This representation is more manual than VisBOL, but it is also more flexible.

You can think of the difference like this:

- **VisBOL** is excellent when you want a standards-aware rendering of the SBOL design itself
- **DNAplotlib** is excellent when you want a programmable publication figure inside a Python workflow

One very effective pattern is to keep SBOL as the canonical design representation, then derive a smaller plotting-oriented representation from it for custom figures.

11.11 Using `SBOL-utilities` to reduce boilerplate

Writing `pySBOL3` objects directly is powerful, but it can feel verbose.

That is where `SBOL-utilities` helps.

The package provides helper constructors for common biological parts and common workflows.

Here we will rebuild a transcriptional unit using helper functions rather than writing each piece by hand.

```
from sbol_utilities.component import promoter as util_promoter
from sbol_utilities.component import rbs as util_rbs
from sbol_utilities.component import cds as util_cds
from sbol_utilities.component import terminator as util_terminator
from sbol_utilities.component import engineered_region

helper_doc = sbol3.Document()

helper_parts = []
for factory, name, seq in [
    (util_promoter, "pTet", "ttgacaattaatcatcggctcgtataatgtgtgga"),
    (util_rbs, "BCD2_helper", "aaaggagg"),
    (util_cds, "mCherry", "atggtgagcaagggcgaggag"),
    (
        util_terminator,
        "B0015",
        "ccgctgagcaataactagcataaccccttggggcctctaaacgggtcttgaggggttttttg",
    ),
]:
    component, sequence = factory(name, seq)
    helper_doc.add(component)
    helper_doc.add(sequence)
    helper_parts.append(component)

helper_tu = engineered_region("TU_mCherry", helper_parts)
helper_doc.add(helper_tu)

helper_path = artifacts / "helper_tu.nt"
helper_doc.write(helper_path, sbol3.SORTED_NTRIPLES)

pd.DataFrame(
    {
                          "quantity":   [len(helper_parts),   len(helper_tu.features),
len(helper_tu.constraints)],
    },
    index=["input parts", "features in engineered region", "ordering constraints"],
)
```

	quantity
input parts	4
features in engineered region	4
ordering constraints	1

This example shows a pattern you will probably use often in real work.

- use `pySBOL3` when you need direct control
- use `SBOL-utilities` when you want a more ergonomic layer for common tasks

The two are complementary.

11.12 Converting older sequence formats into SBOL

A realistic lab does not begin from a perfect SBOL-native world.
You may receive:

- a FASTA file from a collaborator
- a GenBank record from a plasmid repository
- a directory full of mixed sequence files from older projects

One practical reason to use `SBOL-utilities` is that it helps bridge those formats into SBOL.

11.12.1 FASTA to SBOL

```
from sbol_utilities.conversion import convert_from_fasta, convert_to_fasta

fasta_path = artifacts / "toy.fasta"
fasta_path.write_text(">gfp\nATGGTGAGCAAGGGCGAGGAG\n")

fasta_doc = convert_from_fasta(str(fasta_path), "https://example.org/fasta-demo")

fasta_summary = pd.DataFrame(
    {
        "object_type": [type(obj).__name__ for obj in fasta_doc.objects],
        "identity": [obj.identity for obj in fasta_doc.objects],
    }
)

fasta_summary
```

	object_type	identity
0	Sequence	https://example.org/fasta-demo/gfp_sequence
1	Component	https://example.org/fasta-demo/gfp

That conversion gives us an SBOL document that tools can work with directly.
We can also export back out again when needed.

```
roundtrip_fasta = artifacts / "roundtrip.fasta"
convert_to_fasta(fasta_doc, str(roundtrip_fasta))

roundtrip_fasta.read_text()
```

```
'>gfp\nATGGTGAGCAAGGGCGAGGAG\n'
```

11.12.2 GenBank to SBOL

For a GenBank example, we will first create a tiny GenBank record with Biopython and then convert it.

```
from Bio import SeqIO
from Bio.Seq import Seq
from Bio.SeqFeature import FeatureLocation, SeqFeature
from Bio.SeqRecord import SeqRecord
from sbol_utilities.conversion import convert_from_genbank, convert_to_genbank

record = SeqRecord(
    Seq("ATGGTGAGCAAGGGCGAGGAGTAA"),
    id="toy_plasmid",
    name="toy_plasmid",
    description="toy plasmid",
)
record.annotations["molecule_type"] = "DNA"
record.features = [
    SeqFeature(FeatureLocation(0, 24), type="CDS", qualifiers={"label": ["GFP"]})
]

genbank_path = artifacts / "toy.gb"
SeqIO.write(record, genbank_path, "genbank")

genbank_doc = convert_from_genbank(
    str(genbank_path),
    "https://example.org/genbank-demo",
    allow_genbank_online=True,
)

pd.DataFrame(
    {
        "object_type": [type(obj).__name__ for obj in genbank_doc.objects],
        "identity": [obj.identity for obj in genbank_doc.objects],
    }
)
```

	object_type	identity
0	Component	https://example.org/genbank-demo/toy_plasmid
1	Sequence	https://example.org/genbank-demo/toy_plasmid_seq

And we can export back to GenBank when needed.

```
roundtrip_genbank = artifacts / "roundtrip.gb"
records = convert_to_genbank(
    genbank_doc,
    str(roundtrip_genbank),
    allow_genbank_online=True,
)

{
    "roundtrip_file": str(roundtrip_genbank),
    "n_records": len(records),
    "first_record_id": records[0].id,
}
```

```
{'roundtrip_file': 'outputs/ch08/roundtrip.gb',
 'n_records': 1,
 'first_record_id': 'toy_plasmid'}
```

This is a good example of how to think about the formats together.

FASTA and GenBank do not need to disappear.

But if your workflow is moving toward standardization, automation, and interoperability, they should usually become **boundary formats**, while SBOL becomes the **canonical internal representation** of the design.

11.13 A practical mindset for using SBOL

At first, SBOL can feel like extra work.

Why not just keep using strings and GenBank files?

The answer is that standards pay off when projects become larger, more collaborative, or more automated.

SBOL becomes especially valuable when you want to:

- move designs between tools without hand-written adapters
- keep structure and function together in one representation
- connect sequence design to metadata, repositories, simulation, or build planning
- represent systems, not only isolated records
- write reusable code that operates on standardized design objects

If you are working alone on one plasmid, FASTA or GenBank might feel enough.

If you want reproducible, standards-driven synthetic biology software, SBOL is the better long-term choice.

11.14 Recommended workflow

A practical educational workflow looks like this:

1. start with simple sequence manipulation when needed
2. convert important designs into SBOL early
3. use `pySBOL3` for explicit modeling of components and interactions
4. use `SBOL-utilities` to reduce repetitive code and bridge formats
5. treat FASTA and GenBank as import/export formats, not as the richest source of design truth

This mirrors a broader pattern in computational biology.

Raw strings are convenient. Structured objects scale better.

11.15 Exercises

1. Create an SBOL document for a transcriptional unit containing a promoter, RBS, coding sequence, and terminator for a reporter of your choice.
2. Add a protein regulator and encode a repression interaction in SBOL.
3. Convert a small FASTA file into SBOL and inspect the generated top-level objects.
4. Convert a simple GenBank record into SBOL and then export it back to GenBank.
5. Extend one of the examples so that the resulting SBOL document contains two transcriptional units rather than one.

11.16 Recap

In this chapter, we moved from sequence-centric thinking to design-centric thinking.

The main ideas are:

- SBOL is the right format when we need standardized representations of both structure and function
- `pySBOL3` exposes the SBOL 3 data model directly in Python
- `SBOL-utilities` makes common tasks easier and helps bridge older sequence formats into SBOL
- VisBOL gives us a standards-aware way to inspect and communicate SBOL-native designs
- DNAplotlib gives us a programmable way to build highly customized design figures inside Python workflows
- FASTA and GenBank remain useful, but SBOL is the better canonical format for interoperable synthetic biology tooling

This chapter also changes the mental model we will use in the rest of the book.

When a design matters as an engineered object rather than just a nucleotide string, we should now think first in terms of **SBOL documents, components, features, interactions, and standardized exchange**.

12. Design-Build-Test-Learn

Synthetic biology is often introduced as a cycle:

design -> build -> test -> learn

That is true.

But for Python users, DBTL is also something more specific.

It is a **data pipeline**. Each stage consumes structured information, transforms it, and produces new structured information for the next stage. If the stages do not agree on representations, metadata, and interfaces, the loop breaks. If they do agree, then the loop can become reproducible, automatable, and eventually scalable.

This chapter uses three tools from your thesis work as a concrete case study in closing the loop:

- **LOICA** for design, simulation, and characterization
- **PUDU** for build planning and liquid-handling automation
- **Flapjack** for experimental data management, querying, visualization, and analysis

Together they illustrate an important lesson for synthetic biology software:

> the hard part is not only making a useful tool for one stage, but making the outputs of one stage become usable inputs for the next.

12.1 Why DBTL matters computationally

In many labs, DBTL is still partly manual. A researcher sketches a design, builds DNA constructs, runs a plate-reader experiment, exports a spreadsheet, makes a plot, and then updates the next design by hand.

That workflow can work at small scale. But it becomes fragile as soon as we want to:

- compare many candidate designs
- track metadata consistently
- reuse past experimental results
- automate liquid handling
- parameterize models from data
- rerun the same analysis months later

From a Python perspective, DBTL is the problem of building **composable software objects and file formats** for biological engineering.

That is why the previous chapter on SBOL matters so much here. DBTL becomes much easier to automate when designs, build plans, and metadata are represented in machine-readable ways rather than buried in screenshots, notebooks, or informal spreadsheets.

12.2 One DBTL cycle as a data transformation pipeline

One useful way to think about the cycle is to focus on the main artifact produced at each stage.

```
import pandas as pd

pipeline = pd.DataFrame(
    {
        "stage": ["design", "build", "test", "learn"],
        "main_artifact": [
            "design objects and models",
            "assembly plan and protocol",
            "measurements plus metadata",
            "parameters, summaries, and ranked designs",
        ],
```

```
        "python_question": [
            "How do I represent a genetic network?",
            "How do I turn a design into executable instructions?",
            "How do I query and analyze assay data?",
            "How do I feed the results back into the next design?",
        ],
    }
)

pipeline
```

	stage	main_artifact	python_question
0	design	design objects and models	How do I represent a genetic network?
1	build	assembly plan and protocol	How do I turn a design into executable instruc...
2	test	measurements plus metadata	How do I query and analyze assay data?
3	learn	parameters, summaries, and ranked designs	How do I feed the results back into the next d...

This is simple, but it captures the core software idea of the chapter. Each stage is not just an activity. It is a transformation from one data structure into another.

12.3 A thesis-derived software stack for closing the loop

Your thesis frames this nicely. The work focuses on creating DBTL workflows for engineering synthetic genetic network dynamics, and emphasizes that many existing tools do not cover the whole cycle or connect cleanly across stages. The resulting framework is modular, standards-aware, and intended to support simulated, manual, and automated workflows.

The three tools in this chapter fit into the loop like this:

12.3.1 LOICA: design and model-driven iteration

LOICA is a Python package for designing, modeling, and characterizing genetic networks. The package README describes it as a tool for designing, modeling, and characterizing genetic networks, with support for synthetic data generation, communication with Flapjack, and SBOL I/O.

In the thesis, LOICA is used as the design-stage engine that can generate network models, run simulations, export standards-based representations, and later absorb characterization results back into the design loop.

12.3.2 PUDU: build planning and liquid-handling automation

PUDU is a Python package for liquid-handling robot control in synthetic biology workflows. Its README presents it as a way to make robot programming small and simple, and recommends a workflow based on developing and simulating protocols locally before running them on an OT-2.

In the thesis, PUDU sits at the build stage and turns standardized build information into executable assembly, transformation, and plate-setup workflows, including metadata capture.

12.3.3 Flapjack: test and learn

Flapjack is the data-management and analysis side of the loop. Its paper describes it as a system for organizing, querying, plotting, and analyzing characterization data, with a REST API and Python package, and with measurements linked to design metadata through SBOL.

In the thesis, Flapjack closes the loop by storing simulated or experimental measurements, supporting visualization and analysis, and enabling characterization results to feed back into LOICA.

12.4 Three levels of workflow closure

A nice feature of the thesis work is that it does not treat DBTL as all-or-nothing. Instead, it shows three progressively more connected workflows:

1. a **simulated** DBTL workflow
2. a workflow with **manual build** but software-assisted design and learn
3. a workflow with **automated build** using liquid handling

That progression is pedagogically useful. It shows students that they do not need a robot to understand DBTL. They can start with simulation, then add richer metadata and experimental data, and then later add automation.

```
workflow_levels = pd.DataFrame(
    {
        "workflow": [
            "simulated DBTL",
            "manual build workflow",
            "automated build workflow",
        ],
        "design": ["LOICA", "LOICA", "LOICA"],
       "build": ["PUDU simulated", "manual execution + PUDU planning", "PUDU + OT-2"],
        "test_learn": ["Flapjack on simulated data", "Flapjack on plate-reader data",
"Flapjack on automated experimental data"],
    }
)

workflow_levels
```

	workflow	design	build	test_learn
0	simulated DBTL	LOICA	PUDU simulated	Flapjack on simulated data
1	manual build workflow	LOICA	manual execution + PUDU planning	Flapjack on plate-reader data
2	automated build workflow	LOICA	PUDU + OT-2	Flapjack on automated experimental data

12.5 The software architecture of a closed loop

A compact way to summarize the architecture is this:

```
workflow_graph = {
    "LOICA": ["SBOL", "Flapjack"],
    "SBOL": ["sbol-utilities", "PUDU"],
    "PUDU": ["metadata", "lab automation", "Flapjack"],
    "Flapjack": ["analysis", "characterization", "LOICA"],
}
workflow_graph
```

```
{'LOICA': ['SBOL', 'Flapjack'],
 'SBOL': ['sbol-utilities', 'PUDU'],
 'PUDU': ['metadata', 'lab automation', 'Flapjack'],
 'Flapjack': ['analysis', 'characterization', 'LOICA']}
```

This is of course a simplification. But it shows an important design principle:

- **LOICA** produces structured designs and models
- **SBOL and SBOL-utilities** provide the standards bridge
- **PUDU** turns build plans into executable build actions
- **Flapjack** stores and analyzes measurements
- the outputs from **Flapjack** inform the next round of **LOICA** models

That is what people mean when they say a DBTL loop is *closed*.

12.6 Using LOICA at the design stage

LOICA is a good teaching example because it treats a genetic network as a composition of Python objects. That makes it useful not only biologically but also computationally. It helps readers see that a design tool is really an **object model** plus a **simulation interface** plus a **data interface**.

The official LOICA notebooks show a pattern like this:

- connect to Flapjack
- create or retrieve data objects such as study, vector, and signal
- instantiate a `GeneticNetwork`
- add `Reporter` and `Operator` objects
- create a metabolism context
- simulate an `Assay`
- upload the results
- characterize the operator from the resulting data

Below is a simplified version of that pattern based on the LOICA `Source` notebook.

```
from loica import *
from flapjack import *
import getpass
import matplotlib.pyplot as plt

fj = Flapjack(url_base="localhost:8000")
fj.log_in(
    username=input("Flapjack username: "),
    password=getpass.getpass("Password: "),
)

study = fj.create("study", name="Loica testing", description="Test study")
dna = fj.create("dna", name="source")
vector = fj.create("vector", name="source", dnas=dna.id)
sfp = fj.create("signal", name="SFP", color="green")

network = GeneticNetwork(vector=vector.id[0])
reporter = Reporter(
    name="SFP",
    color="green",
    degradation_rate=0,
    init_concentration=0,
    signal_id=sfp.id[0],
)
network.add_reporter(reporter)

source = Source(output=reporter, rate=10)
network.add_operator(source)

plt.figure(figsize=(3, 3), dpi=150)
network.draw()
```

Even if you do not run this immediately, several useful ideas appear here.

First, the biological design is represented as a set of interacting objects rather than as one large opaque script. Second, the design already knows how to connect to data objects in Flapjack. Third, the design can be visualized and simulated before anything is built.

That is exactly why a design tool matters in DBTL. It lets us reason about candidate systems before spending experimental effort on them.

12.6.1 Adding simulation context

LOICA does not only define a network. It also defines the context in which that network operates. In the tutorial notebook, this is done with a simulated metabolism, samples, and an assay.

```
def growth_rate(t):
    return gompertz_growth_rate(t, 0.01, 1, 1, 0.5)


def biomass(t):
    return gompertz(t, 0.01, 1, 1, 0.5)


metab = SimulatedMetabolism(biomass, growth_rate)

media = fj.create("media", name="loica", description="Simulated loica media")
strain = fj.create("strain", name="loica", description="Loica test strain")

samples = []
for _ in range(5):
    sample = Sample(
        genetic_network=network,
        metabolism=metab,
        media=media.id[0],
        strain=strain.id[0],
    )
    samples.append(sample)

biomass_signal = fj.create("signal", name="SOD", description="Simulated OD")

assay = Assay(
    samples,
    n_measurements=100,
    interval=0.24,
    name="Loica constitutive expression",
    description="Simulated constitutive gene generated by LOICA",
    biomass_signal_id=biomass_signal.id[0],
)

assay.run()
```

This is a powerful computational idea. The same environment that stores experimental measurements can also store simulated measurements. That means the line between *test* and *design* becomes more fluid. Simulation can be used as a first-pass filter before a wet-lab experiment ever begins.

12.6.2 Uploading and characterizing the result

A closed loop needs feedback. That is where characterization comes in.

```
assay.upload(fj, study=study.id[0])

source.characterize(
    fj,
    vector=vector.id,
    media=media.id,
    strain=strain.id,
    signal=sfp.id,
    biomass_signal=biomass_signal.id,
)

source.rate
```

This pattern is central to the thesis: simulate or measure behavior, store the result in Flapjack, characterize a design object, and use the resulting parameters in future designs. The thesis explicitly describes LOICA designs being parameterized from Flapjack data and then reused in later genetic network designs.

12.7 LOICA as a Python lesson

LOICA is not just a domain tool. It is also a good software design lesson.

It teaches that:

- biological abstractions can be modeled as Python classes
- simulation should be attached to meaningful domain objects
- design objects become more useful when they connect to real data
- standards such as SBOL let those objects leave one tool and enter another

That combination of **object-oriented abstraction** and **standards-based interoperability** is one of the strongest themes in your thesis work.

12.8 Using PUDU at the build stage

Design is only half the story. At some point, a design has to become a build plan.

That translation step is often where synthetic biology workflows become messy. A diagram is not a protocol. A list of parts is not yet a robotic workflow. A GenBank file is not automatically enough information for assembly, transformation, or plate setup.

PUDU is interesting because it treats build-stage automation as a Python problem:

- define the build inputs clearly
- encode them as structured data
- generate instructions for a robot and a human
- capture useful metadata while doing so

The thesis describes PUDU as a build-stage tool that can use standard build plans to simulate and automate DNA assembly, transformation, and test setup, while producing metadata that can feed later stages.

12.8.1 A minimal Loop assembly example

The thesis gives a very compact example of a PUDU assembly specification: a dictionary of parts by role, passed into a Loop assembly protocol class. That same pattern also appears in the PUDU codebase.

```
from opentrons import protocol_api
from pudu.assembly import Loop_assembly

assemblies = {
    "promoter": ["GVP0008", "GVP0010", "GVP0012", "GVP0013", "GVP0015", "GVP0016"],
    "rbs": "B0034",
    "cds": "sfGFP",
    "terminator": "B0015",
    "receiver": "Odd_1",
}


def run(protocol: protocol_api.ProtocolContext):
    pudu_loop_assembly = Loop_assembly(assemblies=[assemblies])
    pudu_loop_assembly.run(protocol)
```

This is a deceptively important example. The build plan is encoded as a Python dictionary. That means we can generate it programmatically, validate it, transform it, version-control it, and connect it to standards-based descriptions produced earlier in the workflow.

12.8.2 Simulating a protocol before running it

The PUDU README recommends simulating protocols locally before transferring the script to the OT-2, for example with `opentrons_simulate ./scripts/run_Loop_assembly.py`.

That is a wonderful engineering lesson for students. Before we run a biological build, we can run a **software build** of the protocol itself.

In other words, the build stage has its own design-test cycle.

12.8.3 PUDU and human-readable output

One subtle but practical detail from both the thesis and the repository is that PUDU is not only about robot execution. It also produces instructions that humans can use for deck setup, labeling, and verification.

That matters because many real labs are hybrid environments. Some steps are automated and some are still manual. Good software should support that reality rather than assume a perfectly robotic lab.

12.8.4 A plate-setup style example

The thesis also describes using PUDU to turn a set of transformed samples into a plate layout for kinetic measurements. A compact version looks like this:

```
from opentrons import protocol_api
from pudu.transform import Plate_samples


def run(protocol: protocol_api.ProtocolContext):
    pudu_plate_samples = Plate_samples(
        samples=["s1", "s2", "s3", "s4", "s5", "s6"],
        starting_slot=13,
    )
    pudu_plate_samples.run(protocol)
```

This is a good reminder that *build* and *test setup* are often tightly coupled. If we do not track where samples go, what they contain, and how they were prepared, then the test stage becomes difficult to analyze later.

12.9 PUDU as a Python lesson

PUDU teaches several important software lessons:

- protocols become easier to reuse when they are parameterized by structured inputs
- robotic automation is easier to trust when it can be simulated first
- metadata capture is not an afterthought; it is part of the protocol design
- a good automation tool should help both humans and machines

This is exactly why the thesis places so much emphasis on standard build plans and metadata representation.

12.10 Using Flapjack for test and learn

The test stage is where designs meet evidence. But raw measurements alone are not enough. We also need to know:

- what construct was measured
- in what host strain
- in what medium
- under what inducer conditions
- in which assay
- with which reporter and biomass signals

That is what makes Flapjack so important in this ecosystem. It does not just store numbers. It stores measurements **in context**.

The Flapjack paper emphasizes that characterization depends on connecting measurement data with metadata and part composition, and that the tool provides an interactive frontend, a REST API, and a Python package for external integration.

12.10.1 Connecting with pyFlapjack

The pyFlapjack notebook examples use a simple but expressive pattern:

- create a Flapjack client
- log in
- retrieve studies, vectors, media, strains, and signals
- request measurements or analyzed data as data frames
- plot or further analyze the results in Python

```
import getpass
from flapjack import Flapjack

user = input("Flapjack username: ")
passwd = getpass.getpass()

fj = Flapjack("flapjack.rudge-lab.org:8000")
fj.log_in(username=user, password=passwd)

study = fj.get("study", name="Context effects")
vector = fj.get("vector", name="pAAA")
media = fj.get("media", name="M9-glucose")
strain = fj.get("strain", name="MG1655z1")
od = fj.get("signal", name="OD")

raw = fj.measurements(
    study=study.id,
    media=media.id,
    strain=strain.id,
    vector=vector.id,
)

analysis = fj.analysis(
    study=study.id,
    media=media.id,
    strain=strain.id,
    vector=vector.id,
    type="Background Correct",
    biomass_signal=od.id,
)
```

For teaching, this is a beautiful example because the return values are data frames. That means students can move directly into the general Python data stack they already know:

- pandas for wrangling
- matplotlib or plotly for plots
- numpy for numerical work
- scipy for fitting and inference

12.10.2 Plotting measurements and analysis results

The notebook examples also show that Flapjack can either return data frames for custom plotting or create more opinionated plots directly.

```
raw[raw.Signal == "OD"].plot(x="Time", y="Measurement", style=".")

fig = fj.plot(
    assay=study.id,
    media=media.id,
    normalize="None",
    subplots="Signal",
    markers="Vector",
```

```
    plot="Mean",
)
fig.show()
```

And for analysis-derived summaries:

```
cfp = fj.get("signal", search="CFP")

fig = fj.plot(
    assay=study.id,
    media=media.id,
    type="Mean Expression",
    biomass_signal=od.id,
    ref_signal=cfp.id,
    normalize="None",
    subplots="Signal",
    markers="Vector",
    plot="Mean",
)
fig.show()
```

The thesis chapter on DBTL describes the same general pattern: experimental data are uploaded to Flapjack, inspected visually, analyzed using the inverse characterization method, summarized, and then used to characterize components for the next design round.

12.10.3 Getting tabular outputs for further learning

One of the strongest computational ideas in Flapjack is that analysis results can be requested as tables rather than only viewed in a browser. That means the learn stage can become programmable.

```
df = fj.analysis(
    assay=[study.id[0]],
    media=media.id,
    type="Alpha",
    biomass_signal=od.id,
    ref_signal=cfp.id,
)

df.head()
```

Once the result is a tidy data frame, the rest of the learn stage becomes ordinary Python. We can rank variants, compare conditions, fit curves, build predictors, or feed summary parameters back into LOICA.

12.10.4 Flapjack as the bridge between test and design

This is the central point. Flapjack is not only a place to *look at plots*. It is part of the feedback architecture.

Your thesis explicitly describes using Flapjack analysis tools to inspect raw and processed data, compute mean expression and biomass, and then use LOICA to characterize source operators from the resulting data, closing the cycle.

That is exactly what the learn stage should do. It should not end in a figure. It should end in an updated design model.

12.11 A worked conceptual loop

It helps to summarize the whole workflow as one conceptual program:

```
cycle = [
    "1. Define candidate genetic networks in LOICA",
    "2. Export or translate design information into SBOL-aware build representations",
    "3. Use PUDU to simulate or execute assembly and test setup",
    "4. Upload simulated or experimental measurements to Flapjack",
```

```
    "5. Query and analyze results with pyFlapjack",
    "6. Characterize model components and update the next LOICA design",
]
cycle
```

```
['1. Define candidate genetic networks in LOICA',
 '2. Export or translate design information into SBOL-aware build representations',
 '3. Use PUDU to simulate or execute assembly and test setup',
 '4. Upload simulated or experimental measurements to Flapjack',
 '5. Query and analyze results with pyFlapjack',
 '6. Characterize model components and update the next LOICA design']
```

That list is the heart of the chapter.

Notice that the same high-level logic works in all three modes:

- all-software simulation
- software plus manual wet-lab build
- software plus robotic build

The details change, but the dataflow stays recognizable.

12.12 Why standards matter here

Without standards, each handoff between tools becomes custom glue code. That is expensive, fragile, and hard to maintain.

The thesis repeatedly emphasizes that standards-aware representations, especially SBOL, are what make these tools modular and connectable across the workflow. The same work also notes that standardized inputs and outputs are essential to reducing gaps between DBTL stages.

That is why the previous chapter belongs directly before this one. SBOL is not just a documentation format. It is part of the software architecture that allows LOICA, PUDU, Flapjack, and related tools to exchange meaningful information.

12.13 A practical teaching strategy

One useful way to teach this chapter is to present the tools in increasing order of laboratory commitment.

12.13.1 1. Start with simulated DBTL

Students can first learn the loop without a wet lab:

- create a small LOICA design
- simulate an assay
- store or inspect the resulting measurements
- compute a summary from the simulated data

This makes DBTL concrete without requiring equipment.

12.13.2 2. Add real experimental data

Next, students can use exported plate-reader data and metadata:

- upload or query measurements in Flapjack
- compare raw and analyzed traces
- derive simple metrics such as mean expression or fitted parameters

Now the *test* and *learn* stages become real.

12.13.3 3. Add automated build concepts

Finally, students can inspect how a design becomes a robotic protocol:

- define assemblies as structured data
- simulate the protocol

- inspect plate layouts or reagent mappings
- reason about metadata capture and reproducibility

Now the whole loop is visible.

12.14 What students should learn from this chapter

Biologically, students should see that DBTL is the core organizing workflow of engineering biology.

Computationally, they should learn five deeper lessons.

12.14.1 1. A workflow is a chain of data structures

Each stage of DBTL should produce artifacts that another stage can consume. That is a software design problem, not only a biological one.

12.14.2 2. Abstractions matter

LOICA works because it introduces a design abstraction for genetic networks. PUDU works because it introduces a protocol abstraction for automated build. Flapjack works because it introduces a data model for assay measurements and metadata.

12.14.3 3. Metadata is part of the science

If we cannot reconstruct what was measured, under which conditions, and from which design, then the learn stage becomes weak.

12.14.4 4. Simulation and experiment should talk to each other

A mature workflow does not keep model code and assay data in separate worlds. It lets them update one another.

12.14.5 5. Standards make software ecosystems possible

A single tool can be useful. A connected set of tools can change how a lab works.

12.15 Minimal installation notes

If you want to experiment with these tools directly, the package names are:

```
pip install loica
pip install pudupy
pip install pyflapjack
```

The official package pages list these names on PyPI, while the repository documentation points readers to notebooks and tutorials for learning the APIs.

Depending on your environment, you may also need:

- an accessible Flapjack instance
- credentials for API access
- Opentrons tooling if you want to simulate or run PUDU protocols on an OT-2

12.16 Exercises

1. Write a Python dictionary that represents a tiny DBTL workflow with four keys: `design`, `build`, `test`, and `learn`. What artifact would you store at each key?
2. Take one of the LOICA examples in this chapter and identify which objects represent **structure**, which represent **context**, and which represent **measurement**.
3. Modify the PUDU assembly dictionary to represent three promoters instead of six. How would you generate such dictionaries automatically from a design table in `pandas`?

4. Imagine you already have a `pyFlapjack` data frame with columns `Vector`, `Signal`, and `Expression`. Write a short `pandas` snippet to rank the top three constructs by expression.
5. Sketch a function called `next_round_designs(results_df)` that would take learning-stage results and propose which constructs to build next.

12.17 Closing thought

DBTL is often drawn as a circle.

For software, though, it is better to think of it as a **loop with memory**. Each cycle should leave behind better metadata, better models, better reusable code, and better structured knowledge for the next round.

That is what makes the combination of LOICA, PUDU, and Flapjack so instructive. They do not just occupy different DBTL stages. They show how Python, standards, metadata, and automation can work together to make the loop tighter, more reproducible, and more genuinely engineerable.

IV Part IV - Case Studies from Real Lab Software

13 Advanced Computational Biology . 149
13.1 What makes this chapter different from earlier modeling chapters? . . . 149
13.2 The biological story . . . 149
13.3 Growth profiles are spatial data . . . 150
13.4 Building the growth profile in Python . . . 150
13.5 Visualizing the growth profile . . . 151
13.6 From growth profile to spatial phase differences . . . 151
13.7 A toy spatial oscillator . . . 152
13.8 Converting a kymograph into tidy data . . . 153
13.9 Plotting a kymograph . . . 154
13.10 Static rings versus traveling waves . . . 155
13.11 Extracting simple summaries from the toy model . . . 156
13.12 How maximal expression changes wavelength . 157
13.13 How degradation changes temporal progression . . . 158
13.14 Why kymographs are such a powerful representation . . . 159
13.15 Thinking in layers: full model, reduced model, analysis pipeline . . . 159
13.16 What the repository teaches about computational workflow . . . 160
13.17 A computational biologist’s checklist for spatial circuit models . . . 160
13.18 A small extension: exporting tidy simulation summaries . . . 161
13.19 What this chapter does *not* capture . . . 161
13.20 Exercises . . . 162
13.21 Recap . . . 162

14 Myers Lab Case Studies . . . 163
14.1 Draft focus . . . 163
14.2 Candidate tools . . . 163
14.3 Draft note . . . 163

15 Personal Projects . . . 165
15.1 Draft focus . . . 165
15.2 Candidate threads . . . 165
15.3 Draft note . . . 165

13. Advanced Computational Biology

Synthetic biology becomes **advanced computational biology** when we stop modeling one well-mixed tube and start modeling how genetic circuits interact with space, growth, mechanics, and time.

That transition matters because many important biological phenomena are not only molecular.

They are also spatial.

Cells grow in colonies, tissues, channels, chambers, and gradients. Those physical settings create differences in growth rate, dilution, crowding, and signal exposure. A circuit that looks simple in a single-cell ordinary differential equation can produce rich multicellular behavior once it is embedded in a spatial setting.

This chapter is built around the paper *Novel Tunable Spatio-Temporal Patterns From a Simple Genetic Oscillator Circuit* and the accompanying `SpatialOscillator` repository from RudgeLab. The paper is a strong example of the kind of computational thinking synthetic biologists should learn:

- connect a gene circuit to a physical environment
- reduce a complicated simulation into a smaller explanatory model
- use parameter scans to discover design rules
- turn simulation outputs into interpretable summaries such as kymographs, wave speeds, and wavelengths

Our goal in this chapter is not to reproduce the full paper exactly.

Instead, we will do something more useful pedagogically:

1. understand the biological and modeling ideas behind the work
2. build a smaller educational model in Python
3. store outputs in **tidy format**
4. interpret spatial patterns through plots and parameter sweeps
5. extract general lessons for advanced computational biology

13.1 What makes this chapter different from earlier modeling chapters?

Earlier in the book we mostly modeled one variable, one circuit, or one well-mixed system.

Here we move into **multi-scale reasoning**.

The main idea from the paper is that a genetic oscillator does not operate in isolation. Its behavior changes because cells at different positions in a colony experience different growth conditions. Mechanical constraints produce a non-uniform growth profile, and because growth dilutes proteins, that spatial growth profile changes the effective dynamics of the repressilator.

That coupling produces visible spatial patterns.

This is the key conceptual move:

- **gene circuit dynamics** set the oscillatory machinery
- **growth and mechanics** change the local timescale of that machinery
- **space** turns those local differences into patterns we can see

That is advanced computational biology in one sentence.

13.2 The biological story

The paper studies a repressilator-like oscillator in a growing colony. A repressilator is a three-node negative feedback loop in which each repressor inhibits the next one in the cycle. When tuned appropriately, the system oscillates.

In a spatially growing colony, not all cells grow at the same rate. Cells near the edge continue growing more rapidly, while cells deeper inside the colony become growth-limited. That difference matters because protein concentrations are affected not only by degradation but also by dilution during growth.

So even if the circuit topology is unchanged, the *effective* dynamics of the oscillator depend on position in the colony.

The paper shows that this simple idea is enough to produce tunable spatio-temporal patterns:

- **static rings** when effective degradation is too low
- **traveling waves** when degradation is sufficiently strong
- **longer wavelengths** when maximal expression is higher

The important lesson is not only about the repressilator.

It is about workflow. The authors combine:

- a physical picture of colony growth
- an analytical approximation for growth and velocity
- a simplified one-dimensional model
- more detailed individual-based simulations
- parameter scans and kymographs

That is exactly the kind of layered modeling approach synthetic biologists should learn.

13.3 Growth profiles are spatial data

A central result of the paper is that growth rate decreases approximately exponentially with distance from the colony edge. Let us start there.

We will use a simple exponential growth profile:

[(r) = _0 e^{-r/r_0}]

where:

- `\mu(r)` is the local growth rate
- `r` is distance from the colony edge
- `\mu_0` is the maximal growth rate at the edge
- `r_0` is a characteristic decay length

This is already a useful computational biology lesson.

A spatial model often starts by replacing "space" with one carefully chosen coordinate. Here that coordinate is **distance from the colony edge**. That turns a messy two-dimensional biological problem into something we can reason about.

13.4 Building the growth profile in Python

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


def growth_rate_profile(distance_from_edge, r0=8.23, mu0=1.0):
    """Exponential growth profile measured from the colony edge."""
    return mu0 * np.exp(-distance_from_edge / r0)


distance = np.linspace(0, 40, 200)
growth_rate = growth_rate_profile(distance)

growth_df = pd.DataFrame(
    {
        "distance_from_edge": distance,
        "growth_rate": growth_rate,
        "profile": "colony_edge_decay",
    }
)

growth_df.head()
```

	distance_from_edge	growth_rate	profile
0	0.000000	1.000000	colony_edge_decay
1	0.201005	0.975872	colony_edge_decay
2	0.402010	0.952327	colony_edge_decay
3	0.603015	0.929350	colony_edge_decay
4	0.804020	0.906927	colony_edge_decay

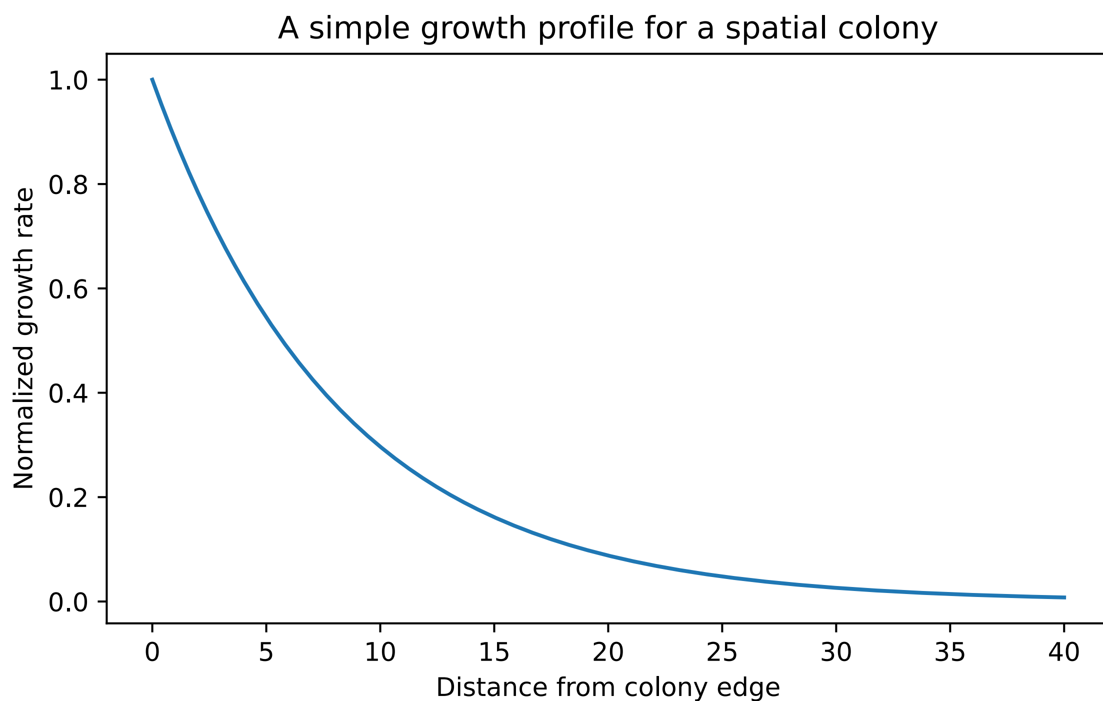
This is a tidy table. Each row is one measurement at one distance for one profile.

That is the format we will keep using.

Even when we compute a whole spatial field or a whole simulation, we should always know how to convert it back into tidy tabular data.

13.5 Visualizing the growth profile

```
plt.figure(figsize=(7, 4))
plt.plot(growth_df["distance_from_edge"], growth_df["growth_rate"])
plt.xlabel("Distance from colony edge")
plt.ylabel("Normalized growth rate")
plt.title("A simple growth profile for a spatial colony")
plt.show()
```



This curve captures an important biological asymmetry. Cells close to the edge grow quickly. Cells far from the edge grow slowly.

That means a circuit coupled to dilution or growth-dependent expression will not behave uniformly across the colony.

13.6 From growth profile to spatial phase differences

The full paper uses a reduced one-dimensional model plus more detailed individual-based simulations. Here we will build a smaller educational model that captures the same qualitative design logic.

The idea is simple.

We assume that each position in the colony hosts an oscillator, but that space introduces a **phase lag** because cells deeper in the colony experience less growth and therefore a different effective timescale.

We also let the temporal oscillation rate depend on a degradation-like parameter `gamma`.

This is **not** the full repressilator model. It is a compact teaching model that helps us reason about:

- why static rings can appear
- why traveling waves can appear
- why one parameter mostly changes temporal behavior
- why another mostly changes spatial spacing

That is often how advanced modeling works in practice. You first build a complex model to understand the mechanism, then build a smaller one to explain the mechanism.

13.7 A toy spatial oscillator

We will encode two design ideas inspired by the paper.

First, stronger degradation should make the pattern evolve faster in time.

Second, stronger maximal expression should produce a longer spatial wavelength.

We translate those ideas into a reduced phase model.

```
def spatial_phase_profile(distance_from_edge, alpha=1e4, r0=8.23):
    """
    Build a spatial phase lag from the cumulative growth deficit.

    Larger alpha gives longer spatial wavelength by reducing the rate at
    which phase changes across space.
    """
    mu = growth_rate_profile(distance_from_edge, r0=r0)
    growth_deficit = 1.0 - mu
    cumulative_deficit = np.cumsum(growth_deficit)
    cumulative_deficit = cumulative_deficit / cumulative_deficit.max()

    alpha_scale = 1 / (1 + 0.35 * np.log10(alpha))
    spatial_phase = 2 * np.pi * 3.0 * alpha_scale * cumulative_deficit
    return spatial_phase


def angular_frequency(gamma):
    """Temporal oscillation rate used in the toy model."""
    if gamma <= 0:
        return 0.0
    return 2 * np.pi * (0.04 + 0.22 * gamma)


def simulate_toy_spatial_oscillator(
    alpha=1e4,
    gamma=0.4,
    r0=8.23,
    n_r=160,
    n_t=240,
    t_max=40,
):
    distance = np.linspace(0, 40, n_r)
    time = np.linspace(0, t_max, n_t)

    mu = growth_rate_profile(distance, r0=r0)
    phase_space = spatial_phase_profile(distance, alpha=alpha, r0=r0)
    omega = angular_frequency(gamma)

    signal = 0.5 * (1 + np.sin(omega * time[:, None] - phase_space[None, :]))
```

```
    return {
        "time_h": time,
        "distance_from_edge": distance,
        "growth_rate": mu,
        "phase_space": phase_space,
        "omega": omega,
        "signal": signal,
        "alpha": alpha,
        "gamma": gamma,
    }


simulation = simulate_toy_spatial_oscillator(alpha=1e4, gamma=0.4)
simulation["signal"].shape
```

```
(240, 160)
```

Our signal is a matrix.

- rows are time points
- columns are spatial positions
- values are normalized expression levels

This matrix is often called a **kymograph array**. A kymograph is one of the most useful visual tools in advanced computational biology, because it shows how a spatial profile changes over time.

13.8 Converting a kymograph into tidy data

Matrices are convenient for simulation. Tidy tables are better for analysis.

We should be able to move between the two.

```
def kymograph_to_tidy(simulation_dict):
    matrix = simulation_dict["signal"]
    time = simulation_dict["time_h"]
    distance = simulation_dict["distance_from_edge"]

    tidy = (
        pd.DataFrame(matrix, index=time, columns=np.round(distance, 3))
        .rename_axis(index="time_h", columns="distance_from_edge")
        .stack()
        .rename("normalized_expression")
        .reset_index()
    )

    tidy["alpha"] = simulation_dict["alpha"]
    tidy["gamma"] = simulation_dict["gamma"]
    return tidy


kymo_df = kymograph_to_tidy(simulation)
kymo_df.head()
```

	time_h	distance_from_edge	normalized_expression	alpha	gamma
0	0.0	0.000	0.500000	10000.0	0.4
1	0.0	0.252	0.499069	10000.0	0.4
2	0.0	0.503	0.497236	10000.0	0.4
3	0.0	0.755	0.494527	10000.0	0.4

	time_h	distance_from_edge	normalized_expression	alpha	gamma
4	0.0	1.006	0.490970	10000.0	0.4

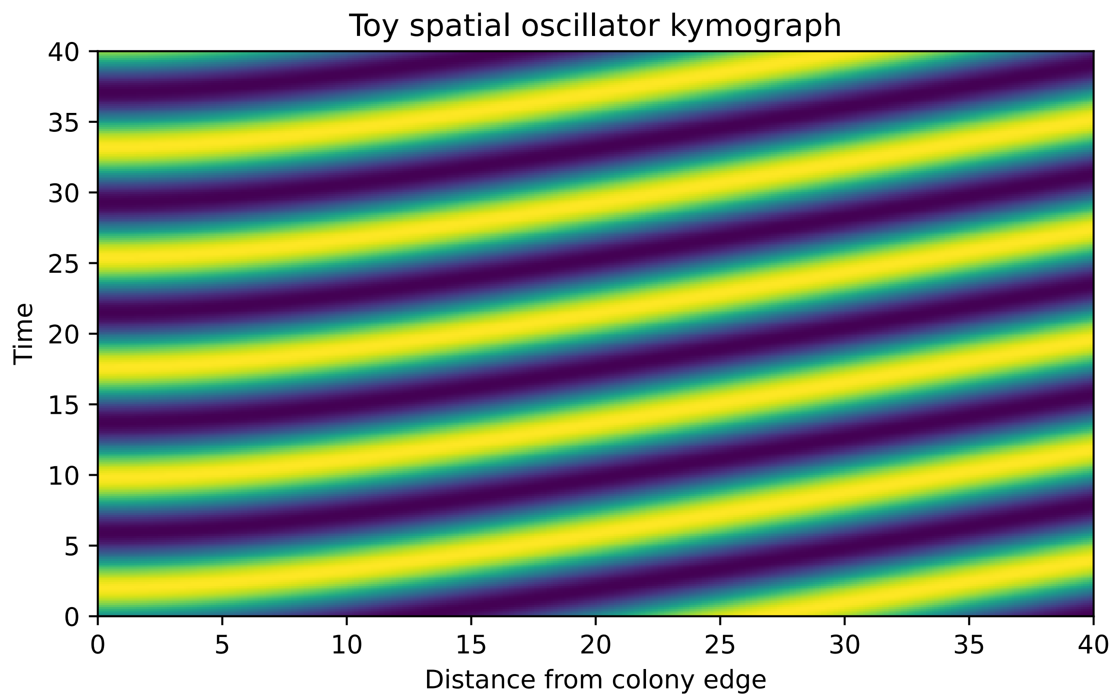
This is the same principle we used for time courses and plate-reader data.

The tidy version makes it easy to:

- filter one condition
- compute derived summaries
- join with metadata
- export to CSV
- reuse the same plotting logic later

13.9 Plotting a kymograph

```
def plot_kymograph(simulation_dict, ax=None, title=None):
    if ax is None:
        ax = plt.gca()

    image = ax.imshow(
        simulation_dict["signal"],
        aspect="auto",
        origin="lower",
        extent=[
            simulation_dict["distance_from_edge"].min(),
            simulation_dict["distance_from_edge"].max(),
            simulation_dict["time_h"].min(),
            simulation_dict["time_h"].max(),
        ],
    )
    ax.set_xlabel("Distance from colony edge")
    ax.set_ylabel("Time")
    if title is not None:
        ax.set_title(title)
    return image


plt.figure(figsize=(7, 4))
plot_kymograph(simulation, title="Toy spatial oscillator kymograph")
plt.show()
```



A kymograph is worth learning to read carefully.

- **horizontal stripes** suggest spatial structure that is not moving in time
- **diagonal stripes** suggest traveling waves
- **wider spacing between stripes** suggests longer wavelength
- **more rapid repetition in time** suggests faster oscillation or faster wave progression

Those visual clues are extremely useful when you are interpreting simulation results.

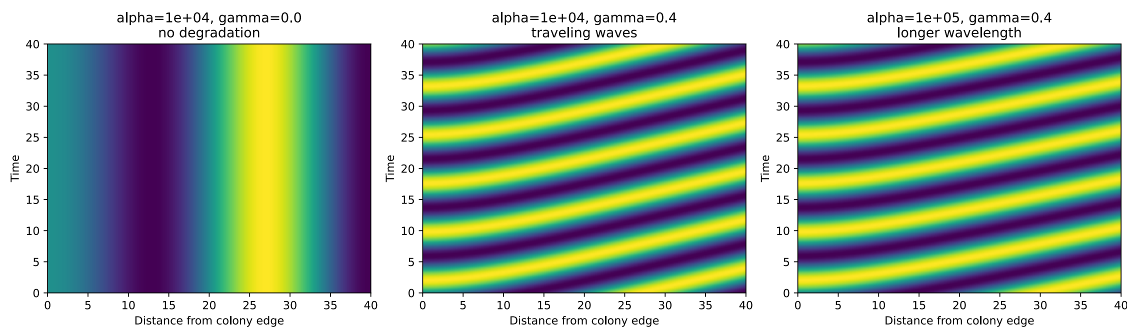
13.10 Static rings versus traveling waves

Now let us compare three conditions.

1. no degradation-like term
2. moderate degradation with moderate expression
3. moderate degradation with stronger expression

```
conditions = [
    {"alpha": 1e4, "gamma": 0.0, "label": "no degradation"},
    {"alpha": 1e4, "gamma": 0.4, "label": "traveling waves"},
    {"alpha": 1e5, "gamma": 0.4, "label": "longer wavelength"},
]

fig, axes = plt.subplots(1, 3, figsize=(14, 4), constrained_layout=True)

for ax, condition in zip(axes, conditions):
    sim = simulate_toy_spatial_oscillator(
        alpha=condition["alpha"],
        gamma=condition["gamma"],
    )
    plot_kymograph(
        sim,
        ax=ax,
                                                title=f"alpha={condition['alpha']:.0e},
gamma={condition['gamma']:.1f}\n{condition['label']}",
    )
```

```
plt.show()
```



This is the central visual lesson of the chapter.

When `gamma = 0`, our toy model produces a **static spatial pattern**. The stripes do not move over time. That corresponds to the intuition behind the fixed-ring regime.

When `gamma` is positive, the pattern begins to move in time, creating **traveling waves**.

When we increase `alpha`, the stripes become more widely spaced in space. That corresponds to a **longer wavelength**.

Again, this is a reduced teaching model, not a literal recreation of every equation in the paper. But it captures the same computational design message: one parameter mainly changes temporal progression, while another mainly changes spatial spacing.

13.11 Extracting simple summaries from the toy model

Advanced computational biology is not only about generating beautiful simulations.

It is also about turning those simulations into quantitative summaries.

Let us extract two simple interpretable features:

- an approximate spatial wavelength from the phase gradient
- the temporal oscillation rate from `gamma`

```
def estimate_wavelength(distance_from_edge, phase_space):
    phase_gradient = np.gradient(phase_space, distance_from_edge)
    return 2 * np.pi / phase_gradient.mean()


rows = []
alphas = [1e3, 1e4, 1e5]
gammas = [0.0, 0.2, 0.4, 0.6, 0.8]

for alpha in alphas:
    for gamma in gammas:
        sim = simulate_toy_spatial_oscillator(alpha=alpha, gamma=gamma)
        rows.append(
            {
                "alpha": alpha,
                "gamma": gamma,
                "estimated_wavelength": estimate_wavelength(
                    sim["distance_from_edge"],
                    sim["phase_space"],
                ),
                "oscillation_rate": sim["omega"] / (2 * np.pi),
            }
        )

scan_df = pd.DataFrame(rows)
scan_df
```

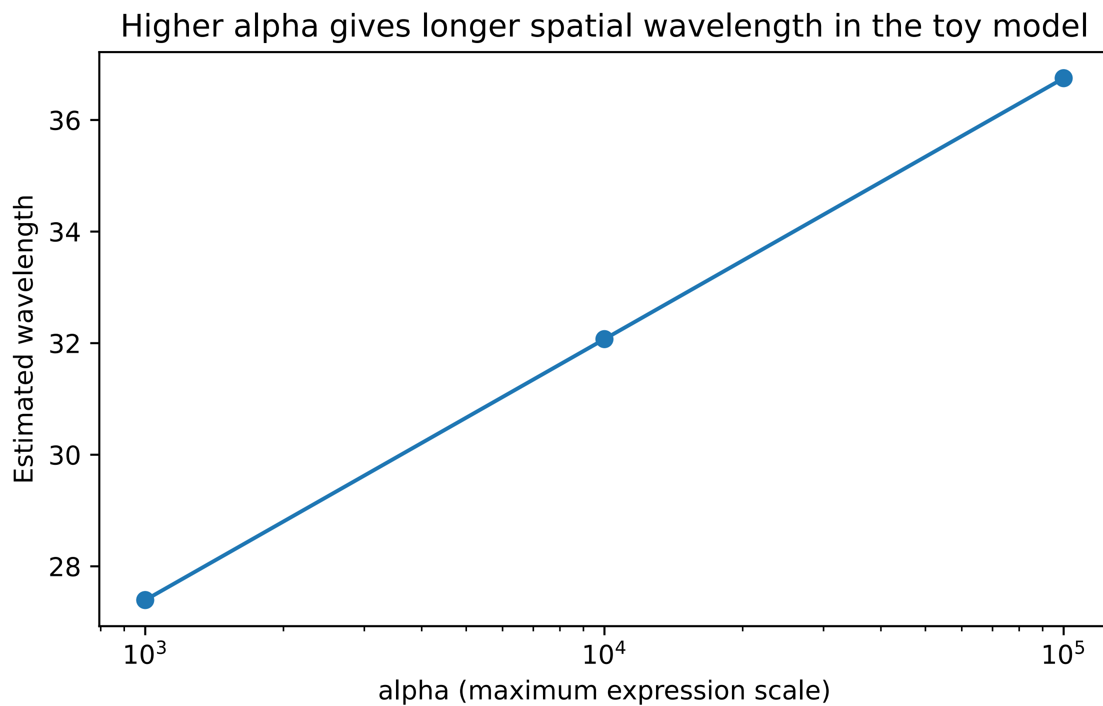
	alpha	gamma	estimated_wavelength	oscillation_rate
0	1000.0	0.0	27.395004	0.000
1	1000.0	0.2	27.395004	0.084
2	1000.0	0.4	27.395004	0.128
3	1000.0	0.6	27.395004	0.172
4	1000.0	0.8	27.395004	0.216
5	10000.0	0.0	32.072200	0.000
6	10000.0	0.2	32.072200	0.084
7	10000.0	0.4	32.072200	0.128
8	10000.0	0.6	32.072200	0.172
9	10000.0	0.8	32.072200	0.216
10	100000.0	0.0	36.749396	0.000
11	100000.0	0.2	36.749396	0.084
12	100000.0	0.4	36.749396	0.128
13	100000.0	0.6	36.749396	0.172
14	100000.0	0.8	36.749396	0.216

Notice that this is tidy parameter-scan output.

Each row is one simulation condition. That makes it easy to plot and compare.

13.12 How maximal expression changes wavelength

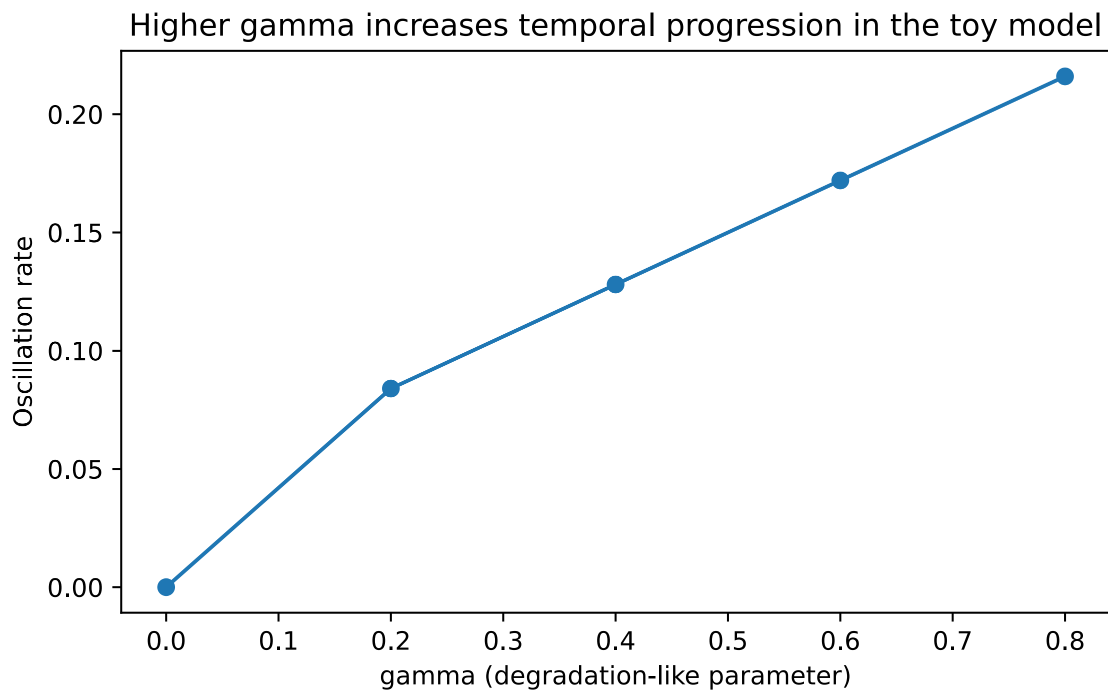
```
wavelength_summary = (
    scan_df.groupby("alpha", as_index=False)["estimated_wavelength"]
    .mean()
    .sort_values("alpha")
)

plt.figure(figsize=(7, 4))
plt.plot(wavelength_summary["alpha"],     wavelength_summary["estimated_wavelength"],
marker="o")
plt.xscale("log")
plt.xlabel("alpha (maximum expression scale)")
plt.ylabel("Estimated wavelength")
plt.title("Higher alpha gives longer spatial wavelength in the toy model")
plt.show()
```



This is the same type of question the paper asks, just in a much smaller model.
How does a biological design parameter change an interpretable spatial feature?
That is what parameter scans are for.

13.13 How degradation changes temporal progression

```
rate_summary = (
    scan_df.groupby("gamma", as_index=False)["oscillation_rate"]
    .mean()
    .sort_values("gamma")
)

plt.figure(figsize=(7, 4))
plt.plot(rate_summary["gamma"], rate_summary["oscillation_rate"], marker="o")
plt.xlabel("gamma (degradation-like parameter)")
plt.ylabel("Oscillation rate")
plt.title("Higher gamma increases temporal progression in the toy model")
plt.show()
```



This mirrors the central design rule from the paper.
A degradation-related parameter controls how fast the spatial pattern evolves.

13.14 Why kymographs are such a powerful representation

Kymographs deserve special attention because they show up often in advanced computational biology.
They are useful whenever you have:

- a one-dimensional spatial coordinate plus time
- a radial coordinate plus time
- a moving front plus time
- a sequence position plus cycle number
- a lineage position plus time

In this chapter, the kymograph lets us compress a complicated spatial oscillator into a plot we can reason about visually.
That is part of the art of good computational biology.
Not every output should stay in raw array form. Sometimes the right representation is the thing that makes the mechanism obvious.

13.15 Thinking in layers: full model, reduced model, analysis pipeline

The real power of the paper is not just the repressilator result.
It is the **layered modeling workflow**.

13.15.1 Layer 1: a biological mechanism

The biological idea is that growth is non-uniform in space, and growth affects protein dilution.

13.15.2 Layer 2: a physical profile

The spatial setting generates a structured growth profile. In the paper this is linked to colony mechanics and physical constraints.

13.15.3 Layer 3: a reduced mathematical model

The authors derive a simplified one-dimensional model that captures how oscillator timing changes across the colony.

13.15.4 Layer 4: a richer computational model

The study also compares these ideas to more detailed individual-based simulations. That richer layer is where spatial mechanics and colony geometry become more realistic.

13.15.5 Layer 5: analysis outputs

The final interpretation happens through summaries such as:

- growth profiles
- kymographs
- wave speed
- wavelength
- parameter scans

This is a very general pattern. It applies far beyond this specific paper.

13.16 What the repository teaches about computational workflow

The `SpatialOscillator` repository is also pedagogically useful. Its top-level structure separates **models** from **analysis**, which is a habit worth copying.

That separation is valuable because simulation and interpretation are different tasks.

- **model code** generates states, trajectories, and raw outputs
- **analysis code** computes summaries, makes figures, and compares conditions

When projects grow, mixing those layers becomes painful.

A good habit is:

1. simulate into raw arrays or files
2. convert results into tidy summaries
3. build figures from those tidy summaries
4. keep figure logic reproducible and separate from model logic

That is one of the most transferable lessons in this chapter.

13.17 A computational biologist's checklist for spatial circuit models

When you move from simple gene expression models to spatial models, ask the following questions.

13.17.1 What is the spatial coordinate?

Is it:

- radial distance from an edge?
- absolute x-position in a channel?
- a 2D grid location?
- a graph or lineage coordinate?

Good models often begin by choosing the smallest coordinate that still captures the phenomenon.

13.17.2 What couples space to circuit dynamics?

Possibilities include:

- dilution through growth
- nutrient gradients
- inducer gradients
- diffusion of signals

- mechanics and crowding
- variable plasmid burden

In this chapter the coupling variable was growth.

13.17.3 What output representation makes the phenomenon interpretable?

Possibilities include:

- line plots at fixed positions
- snapshots in space
- kymographs
- phase diagrams
- tidy parameter scans

Do not wait until the end to think about this. The right output format can shape the whole analysis pipeline.

13.17.4 Can you build a reduced model before the full model?

You usually should.

Reduced models help with:

- intuition
- debugging
- parameter interpretation
- teaching
- identifying which mechanisms actually matter

13.18 A small extension: exporting tidy simulation summaries

A good chapter example should leave behind something reusable. Let us export our scan table.

```
output_path = "outputs/ch10_spatial_oscillator_scan.csv"
scan_df.to_csv(output_path, index=False)
output_path
```

```
'outputs/ch10_spatial_oscillator_scan.csv'
```

This is a tiny example of a big habit.

Advanced computational biology benefits from saving derived summaries explicitly. Do not rely only on one notebook state or one figure. Save the tables that support your interpretation.

13.19 What this chapter does *not* capture

It is important to be honest about model scope.

Our toy model does **not** include:

- the full repressilator ODE system
- explicit mRNA and protein states for three repressors
- individual-based mechanics
- realistic colony geometry updates
- stochastic cell-level variability
- explicit microfluidic boundary conditions

That is okay.

The value of this chapter is that it teaches the computational logic behind the paper without forcing the reader to begin with a large simulation framework.

In research, both levels matter.

- the **detailed model** is needed for realism and testing
- the **reduced model** is needed for explanation and design intuition

13.20 Exercises

1. Change `r0` in the growth profile and see how quickly growth decays away from the colony edge. What happens to the spatial phase profile?
2. Modify `spatial_phase_profile()` so that `alpha` has a stronger or weaker effect. How does that change the apparent wavelength?
3. Add a second signal and phase-shift it by `2π/3` to mimic another node in the repressilator. Plot the three signals at one spatial position.
4. Replace the exponential growth profile with a linear or logistic profile. Which features of the kymographs are robust, and which depend strongly on the chosen profile?
5. Build a channel version of this toy model by using one spatial axis `x` rather than distance from a colony edge.
6. Save the tidy kymograph table for one condition and compute the mean expression at each position over time.

13.21 Recap

This chapter introduced a different style of computational biology.

We moved from single-variable circuit models to a multi-scale way of thinking in which:

- space matters
- growth matters
- mechanics matter
- visual summaries matter
- reduced models and detailed models support each other

The key biological lesson from the paper is that a simple genetic oscillator can generate rich multicellular patterns when coupled to a spatial growth profile.

The key computational lesson is even broader.

Advanced computational biology is often about choosing the right abstraction at the right level:

- a physical profile simple enough to analyze
- a reduced model simple enough to understand
- a simulation rich enough to test the idea
- an analysis representation simple enough to interpret

That is exactly the mindset synthetic biologists need when they begin designing systems that live not only in time, but also in space.

14. Myers Lab Case Studies

14.1 Draft focus

Use current lab tools to show standards-driven, integrative synbio software.

14.2 Candidate tools

- SynBioSuite
- SeqImprove
- BuildCompiler
- PUDU as a bridge from prior work

14.3 Draft note

This chapter can emphasize software ecosystems, interfaces, and how toolchains evolve inside research groups.

15. Personal Projects

15.1 Draft focus

This chapter can bridge biography, technical growth, and software identity.

15.2 Candidate threads

- LOICA contributions and notebook-based modeling
- Flapjack API related examples
- automation-oriented projects
- educational mini-libraries spun out of the book itself

15.3 Draft note

Keep this chapter less autobiographical than methodological. The point is to show how projects evolve from learning, collaboration, and research needs.

V

Part V - The Future Lab

16 The DRAGGON Lab Vision 169
16.1 Draft focus 169
16.2 Themes 169
16.3 Draft note 169

17 Capstone Projects 171
17.1 Draft focus 171
17.2 Candidate capstones 171
17.3 Draft note 171

18 References 173

18 Bibliography 175

16. The DRAGGON Lab Vision

16.1 Draft focus

Articulate the future lab as a coherent research and engineering program.

16.2 Themes

- computational-experimental integration
- automated and modular DBTL workflows
- AI and hybrid modeling for biological engineering
- software ecosystems for engineering biology across scales
- interoperable tools, standards, and simulation engines

16.3 Draft note

This chapter should not read like marketing copy. It should read like a research and software agenda that future trainees can build from.

17. Capstone Projects

17.1 Draft focus

End the book with buildable project ideas.

17.2 Candidate capstones

- sequence analysis mini-package
- promoter or part library explorer
- simple DBTL tracker
- regulatory network simulator
- standards-aware design conversion utility
- educational package developed alongside the book

17.3 Draft note

Each capstone should have:

- learning objectives
- suggested extensions
- pointers back to earlier chapters

18. References

A bibliography file can be added later to support formal citations, software references, and recommended reading.

For now, this chapter can collect:

- software repositories
- papers
- standards documentation
- textbooks and open educational resources

Bibliography