Practical Python Programming for Life Scientists

Ryan M. Moore, PhD

2025-02-05

Table of contents

Introduction

Welcome to Practical Python Programming for Life Scientists! This book is designed for biology and life science students with little to no prior coding experience. Rather than aiming to make you Python experts, the goal is to help you develop fundamental programming concepts and data analysis skills using Python as a practical tool.

The content progresses from basic syntax through algorithms, functions, classes, error handling, data science applications, and testing methodologies. Each concept is presented with life science examples to show how programming principles can enhance your research capabilities.

This resource serves as an introduction to computational thinking in biological contexts, providing a solid foundation to approach scientific questions from a programming perspective and to effectively incorporate data analysis using Python into your research workflow.

Note: This book is a work in progress and will continue to evolve with student feedback. Some sections may still be in development, with the main chapters expected to be finished by May 2025.

1 Basics

Welcome to your first Python tutorial! In this lesson, we'll explore the fundamental building blocks of Python programming, including:

- Variables and how to use them
- Basic data types (numbers, text, and true/false values)
- Common operators for calculations and comparisons
- Essential built-in functions
- How to control program flow with conditional statements

This is a comprehensive tutorial that covers a lot of ground. Don't feel pressured to master everything at once – we'll be practicing these concepts throughout the course. Think of this as your first exposure to these ideas, and we'll build on them step by step in the coming weeks.

1.1 Introduction to Python

What is Python?

Python is a high-level, interpreted programming language known for its simplicity and readability. Created by Guido van Rossum in 1991, it has become one of the most popular languages in scientific computing and bioinformatics.

High level

Python is a high-level programming language, meaning it handles many complex computational details automatically. For example, rather than managing computer memory directly, Python does this for you. This allows biologists and researchers to focus on solving scientific problems rather than dealing with technical computing details.

Interpreted

Python is an interpreted language, which means you can write code and run it immediately without an extra compilation step. This makes it ideal for bioinformatics work where you often need to:

- Test different approaches to data analysis
- Quickly prototype analysis pipelines
- Interactively explore datasets

Readable syntax

Python's code is designed to be readable and clear, often reading almost like English. For example:

```
if dna_sequence.startswith(start_codon) and dna_sequence.endswith(stop_codon):
    potential_genes.append(dna_sequence)
```

Even if you're new to programming, you can probably guess that this code is looking for potential genes by checking a DNA sequence for a start and a stop codon, and if found, adding the sequence to a list of potential genes.

This readability is particularly valuable in research settings where code needs to be shared and reviewed by collaborators.

Use cases

Python is a versatile language that can be used for a wide range of applications, including:

- Artificial intelligence and machine learning (e.g., TensorFlow, PyTorch)
- Web development (Django, Flask)
- Desktop applications (PyQt, Tkinter)
- Game development (Pygame)
- Automation and scripting

And of course, bioinformatics and scientific computing:

- Sequence analysis and processing (Biopython, pysam)
- Phylogenetics (ETE Toolkit)
- Data visualization (matplotlib, seaborn)
- Pipeline automation (snakemake for reproducible workflows)
- Microbial ecology and microbiome analysis (QIIME)

Why Python for bioinformatics?

Python has become a widely used tool in bioinformatics for several key reasons:

- **Rich ecosystem**: Extensive libraries specifically for biological data analysis
- **Active scientific community**: Regular updates and support for bioinformatics tools
- **Integration capabilities**: Easily connects with other bioinformatics tools and databases
- **Data science support**: Strong support for data manipulation and statistical analysis
- **Reproducibility**: Excellent tools for creating reproducible research workflows

Whether you're analyzing sequencing data, building analysis pipelines, or developing new computational methods, Python provides the tools and community support needed for modern biological research.

1.2 Variables

Think of variables as labeled containers for storing data in your program. Just as you might label test tubes in a lab to keep track of different samples, variables let you give meaningful names to your data – whether they're numbers, text, true/false values, or more complex information.

For example, instead of working with raw values like this:

```
if 47 > 40:
    print("Temperature too high!")
```

```
Temperature too high!
```

You can use descriptive variables to make your code clearer:

```
temperature = 42.3
temperature_threshold = 40.0

if temperature > temperature_threshold:
    print("Temperature too high!")
```

```
Temperature too high!
```

In this section, we'll cover:

- Creating and using variables
- Understanding basic data types (numbers, text, true/false values)
- Following Python's naming conventions
- Converting between different data types
- Best practices for using variables in scientific code

By the end, you'll be able to use variables effectively to write clear, maintainable research code.

Creating variables

In Python, you create a variable by giving a name to a value using the `=` operator. Here's a basic example:

```
sequence_length = 1000
species_name = "Escherichia coli"
```

You can then use these variables anywhere in your code by referring to their names. Variables can be combined to create new variables:

```
# Combining text (string) variables
genus = "Escherichia"
species = "coli"
full_name = genus + " " + species
print(full_name)  # Prints: Escherichia coli

# Calculations with numeric variables
reads_forward = 1000000
reads_reverse = 950000
total_reads = reads_forward + reads_reverse
print(total_reads)  # Prints: 1950000
```

```
Escherichia coli
1950000
```

Notice how the `+` operator works differently depending on what type of data we're using:

- With text (strings), it joins them together
- With numbers, it performs addition

You can also use variables in more complex calculations:

```
gc_count = 2200
total_bases = 5000
gc_content = gc_count / total_bases
print(gc_content)  # Prints: 0.44
```

```
0.44
```

The ability to give meaningful names to values makes your code easier to understand and modify. Instead of trying to remember what the number 5000 represents, you can use a clear variable name like `total_bases`.

Reassigning variables

Python allows you to change what's stored in a variable after you create it. Let's see how this works:

```
read_depth = 100
print(f"Initial read depth: {read_depth}")

read_depth = 47
print(f"Updated read depth: {read_depth}")
```

```
Initial read depth: 100
Updated read depth: 47
```

This flexibility extends even further – Python lets you change not just the value, but also the type of data a variable holds:

```
quality_score = 30
quality_score = "High quality"
print(quality_score)
```

```
High quality
```

While this flexibility can be useful, it can also lead to unexpected behavior if you're not careful. Here's an example that could cause problems in a sequence analysis pipeline:

```
# Correctly calculates and prints the total number of sequences.
sequences_per_sample = 1000
sample_count = 5
total_sequences = sequences_per_sample * sample_count
print(f"total sequences: {total_sequences}")

# This one produces an unexpected result!
sequences_per_sample = "1000 sequences "
sample_count = 5
total_sequences = sequences_per_sample * sample_count
print(f"total sequences: {total_sequences}")
```

```
total sequences: 5000
total sequences: 1000 sequences 1000 sequences 1000 sequences 1000 sequences 1000 sequences
```

In the second case, instead of performing multiplication, Python repeats the string `"1000 sequences "` 5 times! This is probably not what you wanted in your genomics pipeline!

This kind of type changing can be a common source of bugs, especially when:

- Processing input from files or users
- Handling missing or invalid data
- Converting between different data formats

Best practice is to be consistent with your variable types throughout your code, and explicitly convert between types when necessary.

Augmented assignments

Let's look at a common pattern when working with variables. Here's one way to increment a counter:

```
read_count = 100
read_count = read_count + 50
print(f"Total reads: {read_count}")
```

```
Total reads: 150
```

Python provides a shorter way to write this using augmented assignment operators:

```
read_count = 100
read_count += 50
print(f"Total reads: {read_count}")
```

```
Total reads: 150
```

These augmented operators combine arithmetic with assignment. Common ones include:

- `+=`: augmented addition (increment)
- `-=`: augmented subtraction (decrement)
- `*=`: augmented multiplication
- `/=`: augmented division

These operators are particularly handy when updating running totals or counters, like when tracking how many sequences pass quality filters. We'll explore more uses in the next tutorial.

Named constants

Sometimes you'll want to define values that shouldn't change throughout your program.

```
GENETIC_CODE_SIZE = 64
print(f"There are {GENETIC_CODE_SIZE} codons in the standard genetic code")

DNA_BASES = ['A', 'T', 'C', 'G']
print(f"The DNA bases are: {DNA_BASES}")
```

```
There are 64 codons in the standard genetic code
The DNA bases are: ['A', 'T', 'C', 'G']
```

In Python, we use ALL_CAPS names as a convention to indicate these values shouldn't change. However, it's important to understand that Python doesn't actually prevent these values from being changed. For example:

```
MIN_QUALITY_SCORE = 30
print(f"Filtering sequences with quality scores below {MIN_QUALITY_SCORE}")

MIN_QUALITY_SCORE = 20  # We can change it, even though we shouldn't!
print(f"Filtering sequences with quality scores below {MIN_QUALITY_SCORE}")
```

```
Filtering sequences with quality scores below 30
Filtering sequences with quality scores below 20
```

Think of Python variables like labels on laboratory samples: you can always move a label from one test tube to another. When you write:

```
DNA_BASES = ['A', 'T', 'C', 'G']
DNA_BASES = ['A', 'U', 'C', 'G']  # Oops, switched to RNA bases!
print(f"These are now RNA bases: {DNA_BASES}")
```

```
These are now RNA bases: ['A', 'U', 'C', 'G']
```

You're not modifying the original list of DNA bases – instead, you're creating a new list and moving the `DNA_BASES` label to point to it. The original list isn't "protected" in any way. So, it's more of a convention that ALL_CAPS variables be treated as constants in your code, even though Python won't enforce this rule.

Dangerous assignments

Here's a common pitfall when naming variables in Python – accidentally overwriting built-in functions.

Python has several built-in functions that are always available, including one called `str` that converts values to strings. For example:

```
sequence = str()  # Creates an empty string
sequence
```

Note: if you convert this static code block to one that is runnable, and then actually run it, it would cause errors in the rest of the notebook in any place that uses the ***`str`*** *function. If you do this, you will need to restart the notebook kernel.*

However, Python will let you use these built-in names as variable names (though you shouldn't!):

```
str = "ATCGGCTAA"  # Don't do this!
```

Now if you try to use the `str` function later in your code:

```
quality_score = 35
sequence_info = str(quality_score)  # This will fail!
```

You'll get an error:

```
TypeError: 'str' object is not callable
```

This error occurs because we've "shadowed" the built-in `str` function with our own variable. Python now thinks we're trying to use the string "ATCGGCTAA" as a function, which doesn't work!

We'll discuss errors in more detail in a future lesson. For now, remember to avoid using Python's built-in names (like `str`, `list`, `dict`, `set`, `len`) as variable names. You can find a complete list of built-ins in the Python documentation.

Naming variables

Clear, descriptive variable names are crucial for writing maintainable code. When you revisit your analysis scripts months later, good variable names will help you remember what each part of your code does.

Valid names

Python variable names can include:

- Letters (A-Z, a-z)
- Numbers (0-9, but not as the first character)
- Underscores (_)

While Python allows Unicode characters (like Greek letters), it's usually better to stick with standard characters:

```
 = 3.14  # Possible, but not recommended
pi = 3.14  # Better!
```

Case Sensitivity

Python treats uppercase and lowercase letters as different characters:

```
sequence = "ATCG"
Sequence = "GCTA"
print(f"{sequence} != {Sequence}")
```

```
ATCG != GCTA
```

To avoid confusion, stick with lowercase for most variable names.

Naming Conventions

For multi-word variable names, Python programmers typically use snake_case (lowercase words separated by underscores):

```
# Good -- snake case
read_length = 150
sequence_count = 1000
is_high_quality = True

# Avoid - camelCase or PascalCase
readLength = 150
SequenceCount = 1000
```

Guidelines for Good Names

Here are some best practices for naming variables in your code:

Use descriptive names that explain the variable's purpose:

```
# Clear and descriptive
sequence_length = 1000
quality_threshold = 30

# Too vague
x = 1000
threshold = 30
```

Use nouns for variables that hold values:

```
read_count = 500
dna_sequence = "ATCG"
```

Boolean variables often start with `is_`, `has_`, or similar:

```
is_paired_end = True
has_adapter = False
```

Collections (which we'll cover later) often use plural names:

```
sequences = ["ATCG", "GCTA"]
quality_scores = [30, 35, 40]
```

Common exceptions where short names are okay:

- `i`, `j`, `k` for loop indices
- `x`, `y`, `z` for coordinates
- Standard abbreviations like `msg` for message, `num` for number

Keep names reasonably short while still being clear:

```
# Too long
number_of_sequences_passing_quality_filter = 100
# Better
passing_sequences = 100
```

Remember: your code will be read more often than it's written, both by others and by your future self. Clear variable names make your code easier to understand and maintain.

For more detailed naming guidelines, check Python's PEP 8 Style Guide.

1.3 Data Types

Python has many different types of data it can work with. Each data type has its own special properties and uses.

In this section, we'll cover the basic data types you'll use frequently in your code:

- Numbers
  - Integers (whole numbers, like sequence lengths or read counts)
  - Floating-point numbers (decimal numbers, like expression levels or ratios)
- Strings (text, like DNA sequences or gene names)
- Boolean values (True/False, like whether a sequence passed quality control)

We'll learn how to:

- Identify what type of data you're working with
- Convert between different types when needed

Understanding these fundamental data types is crucial for handling data correctly in your programs.

Checking the type of a value

Python is a dynamically typed language, meaning a variable's type can change during your program. While this flexibility is useful, it's important to keep track of your data types to avoid errors in your analysis.

You can check a variable's type using Python's built-in `type()` function. Here's how:

```
sequence_length = 150
print(type(sequence_length))  # <class 'int'>

sequence = "ATCGGCTAA"
print(type(sequence))  # <class 'str'>

is_valid = True
print(type(is_valid))  # <class 'bool'>
```

```
<class 'int'>
<class 'str'>
<class 'bool'>
```

As shown above, `type()` tells us exactly what kind of data we're working with. This can be particularly helpful when debugging calculations that aren't working as expected, or verifying data is in the correct format.

Don't worry too much about the `class` keyword in the output – we'll cover classes in detail later. For now, focus on recognizing the basic types: `int` for integers, `str` for strings (text), and `bool` for True/False values.

Numeric types (int, float)

Python has two main types for handling numbers:

- `int`: Integers (whole numbers) for counting things like:
  - Number of sequences

  - Read lengths
  - Gene counts

- `float`: Floating-point numbers (decimals) for measurements like:
  - Expression levels
  - P-values
  - GC content percentages

For readability with large numbers, you can use underscores: `1_000_000` reads is clearer than `1000000` reads.

Numeric operations

The operators `+`, `-`, `*`, `/` are used to perform the basic arithmetic operations.

```
forward_reads = 1000
reverse_reads = 800
print(forward_reads + reverse_reads)
print(forward_reads - reverse_reads)
print(forward_reads * 2)
print((forward_reads + reverse_reads) / 100)
```

```
1800
200
2000
18.0
```

Float division (`/`) always returns a float, whereas integer division (`//`) returns an int by performing floor division.

```
total_bases = 17
reads = 5
print(total_bases / reads)
print(total_bases // reads)
```

```
3.4
3
```

The operator `**` is used for exponentiation.

```
print(2 ** 8)
print(8 ** 2)
```

```
256
64
```

Parentheses `()` can be used to group expressions and control the order of operations.

```
# Order of operations
print(2 + 3 * 4)     # multiplication before addition
print( (2 + 3) * 4 ) # parentheses first
```

```
14
20
```

Modulo (`%`) gives remainder of division

```
position = 17
codon_position = position % 3  # Which position in codon (0, 1, or 2)
print(codon_position)
```

```
2
```

Be careful about combining negative numbers with floor division or modulo. Here are some interesting examples showing how negative numbers behave with floor division and modulo in Python:

```
# Floor division with negative numbers
print("Floor division with negative numbers:")
# Rounds down to nearest integer
print(17 // 5)
# Rounds down, not toward zero
print(-17 // 5)
print(17 // -5)
print(-17 // -5)

# Modulo with negative numbers
print("\nModulo with negative numbers:")
print(17 % 5)
```

```
# Result is positive with positive divisor
print(-17 % 5)
# Result has same sign as divisor
print(17 % -5)
print(-17 % -5)
```

```
Floor division with negative numbers:
3
-4
-4
3

Modulo with negative numbers:
2
3
-3
-2
```

Don't worry too much about the details of how negative numbers work with division and modulo operations. Just be aware that they can behave unexpectedly, and look up the specific rules if you need them.

Scientific notation

Scientific notation is essential when working with very large or small numbers:

```
# 3.2 billion bases
genome_size = 3.2e9

# 0.00000001 mutations per base
mutation_rate = 1e-8
```

Precision Considerations

Integers

Python can handle arbitrarily large integers, limited only by memory:

```
big_number = 125670495610435017239401723907559279347192756
print(big_number)
```

```
1256704956104350172394017239075592793471927 56
```

Floats

Floating-point numbers have limited precision (about 15-17 decimal digits). This can affect calculations:

```
x = 0.1
y = 0.2

# Might not be exactly 0.3
print(x + y)
```

```
0.30000000000000004
```

While these precision errors are usually small, they can accumulate in large-scale calculations.

Strings

Strings are how Python handles text data, like sequences or gene names.

```
# Strings can use single or double quotes
sequence = 'ATCG'
gene_name = "nrdA"
print(sequence)
print(gene_name)
```

```
ATCG
nrdA
```

Strings are immutable – once created, they cannot be modified. For example, you can't change individual bases in a sequence directly:

```
dna = "ATCG"
# This would raise an error:
# dna[0] = "G"
```

Try uncommenting that line and see what happens!

You can combine strings using the + operator:

```
# String concatenation
sequence_1 = "ATCG"
sequence_2 = "GCTA"
full_sequence = sequence_1 + sequence_2
print("the sequence is: " + full_sequence)
```

```
the sequence is: ATCGGCTA
```

Special characters can be included using escape sequences:

- `\n` for new line
- `\t` for tab
- `\\` for backslash

```
# Formatting sequence output
print("Sequence 1:\tATCG\nSequence 2:\tGCTA")
```

```
Sequence 1: ATCG
Sequence 2: GCTA
```

F-strings (format strings) are particularly useful for creating formatted output. They allow you to embed variables and expressions in strings using `{expression}`:

```
gene_id = "nrdJ"
position = 37_531

print(f"Gene {gene_id} is located at position {position}")
```

```
Gene nrdJ is located at position 37531
```

F-strings can also format numbers, which is useful for scientific notation and precision control:

```
# Two decimal places
gc_content = 0.42857142857
print(f"GC content: {gc_content:.2f}")

# Scientific notation
p_value = 0.000000342
print(f"P-value: {p_value:.2e}")
```

```
GC content: 0.43
P-value: 3.42e-07
```

Strings can contain Unicode characters:

```
# Unicode characters
print(" ")
print("   ")
```

While Python supports Unicode characters in variable names, it's better to use standard ASCII characters for code:

```
# Possible, but not recommended
  = 0.05
  = 0.20

# Better
alpha = 0.05
beta = 0.20
```

Common string operations

String operations are fundamental for processing and manipulating textual data, formatting output, and cleaning up input in your applications and analysis pipelines.

String concatenation with +

The + operator joins strings together:

```
# Joining DNA sequences
sequence1 = "ATCG"
sequence2 = "GCTA"
combined_sequence = sequence1 + sequence2
print(combined_sequence)

# Adding labels to sequences
gene_id = "nrdA"
labeled_sequence = gene_id + ": " + combined_sequence
print(labeled_sequence)
```

```
ATCGGCTA
nrdA: ATCGGCTA
```

String repetition with *

The * operator repeats a string a specified number of times:

```
# Repeating DNA motifs
motif = "AT"
repeat = motif * 3
print(repeat)

# Creating alignment gap markers
gap = "-" * 6
print(gap)
```

```
ATATAT
------
```

String indexing

Python uses zero-based indexing to access individual characters in a string. You can also use negative indices to count from the end:

```
# Indexing
s = "Hello, world!"
print(s[0])
print(s[7])
print(s[-1])
print(s[-8])
```

```
H
w
!
,
```

String slicing

Slicing lets you extract parts of a string using the format `[start:end]`. The `end` index is exclusive:

```
# Slicing
s = "Hello, World!"
print(s[0:5])
print(s[7:])
print(s[:5])
print(s[-6:])
print(s[-12:-8])
```

```
Hello
World!
Hello
World!
ello
```

String methods

Python strings have built-in methods for common operations. Here are a few common ones:

```
# Clean up sequence data with leading/trailing white space
raw_sequence = "  ATCG GCTA  "
clean_sequence = raw_sequence.strip()
print("|" + raw_sequence + "|")
print("|" + clean_sequence + "|")

# Convert between upper and lower case
mixed_sequence = "AtCg"
print(mixed_sequence.upper())
print(mixed_sequence.lower())

# Chaining methods
messy_sequence = "  AtCg  "
clean_upper = messy_sequence.strip().upper()
print("|" + clean_upper + "|")
```

```
|  ATCG GCTA  |
|ATCG GCTA|
ATCG
atcg
|ATCG|
```

Boolean values

Boolean values represent binary states (True/False) and are used to make decisions in code:

- `True` represents a condition being met
- `False` represents a condition not being met

(Note: These are capitalized keywords in Python!)

Boolean variables often use prefixes like `is_`, `has_`, or `contains_` to clearly indicate their purpose:

```
is_paired_end = True
has_adapter = False
contains_start_codon = True
```

Boolean values are used in control flow – they drive decision-making in your code:

```
is_high_quality = True
if is_high_quality:
    print("Sequence passes quality check!")

has_ambiguous_bases = False
if has_ambiguous_bases:
    # This won't execute because condition is False
    print("Warning: Sequence contains N's")
```

```
Sequence passes quality check!
```

Boolean values are created through comparisons, for example:

```
# Quality score checking
quality_score = 35
print(quality_score > 30)
print(quality_score < 20)
print(quality_score == 40)
print(quality_score != 35)
```

```
True
False
False
False
```

Logical operators (`and`, `or`, `not`) combine boolean values:

```
# Logical operations
print(True and False)
print(True or False)
print(not True)
print(not False)
```

```
False
True
False
True
```

For example, you could use logical operators to combine multiple logical statements:

```
is_long_enough and is_high_quality
```

```
is_exempt or exceeds_threshold
```

Comparison operators In Depth

Comparison operators are used to compare "compare" values. They return a boolean value (`True` or `False`) and are often used in conditional statements and loops to control program flow.

The basic comparison operators are:

- `==`: equal to
- `!=`: not equal to
- `<`: strictly less than
- `<=`: less than or equal to
- `>`: strictly greater than
- `>=`: greater than or equal to

Additional operators we'll cover later:

- `is`, `is not`: object identity
- `in`, `not in`: sequence membership

Here are a couple examples:

```python
# Basic boolean values
is_sunny = True
is_raining = False

print(f"Is it sunny? {is_sunny}")
print(f"Is it raining? {is_raining}")

# Comparison operations produce boolean results
temperature = 25
is_hot = temperature > 30
print(f"Is it hot? {is_hot}")

# Logical operations
is_good_weather = is_sunny and not is_raining
print(f"Is it good weather? {is_good_weather}")
```

```
Is it sunny? True
Is it raining? False
Is it hot? False
Is it good weather? True
```

```python
# Comparison operations
print(5 == 5)
print(5 != 5)
print(5 < 3)
print(5 <= 3)
print(5 <= 5)
print(5 > 3)
print(5 >= 3)
print(5 >= 5)
```

```
True
False
False
False
True
True
True
True
```

Chained Comparisons

Comparisons can be chained together, e.g. `1 < 2 < 3` is equivalent to `1 < 2 and 2 < 3`.

```
# Chained comparisons
print(1 < 2 < 3)
print(1 < 2 < 2)
print(1 < 2 <= 2)

# This one is a bit weird, but it's valid Python!
print(1 < 2 > 2)
```

```
True
False
True
False
```

The comparisons operators can also be used to compare the values of variables.

```
# Check if value is in valid range
coverage = 30
print(10 < coverage < 50)

quality_score = 35
print(20 < quality_score <= 40)

# Multiple range checks
temperature = 37.2
print(37.0 <= temperature <= 37.5)
```

```
True
True
True
```

Comparing Strings & Other Values

Python's comparison operators work beyond just numbers, allowing comparisons between various types of data. Be careful though – while some comparisons make intuitive sense, others might require careful consideration or custom implementation.

```
# Comparison of different types
print("Hello" == "Hello")
print("Hello" == "World")
print("Hello" == 5)
print("Hello" == True)

# Some non-numeric types also have a natural ordering.
print("a" < "b")
print("a" < "A")

# This is a bit weird, but it's valid Python!
print([1, 2, 3] <= [10, 20, 30])
```

```
True
False
False
False
True
False
True
```

Logical Operators In Depth

Think of logical operators as ways to combine or modify simple yes/no conditions in your code, much like how you might combine criteria when filtering data in Excel or selecting samples for an experiment.

For example, you can use logical operators to express conditions like:

- "If a DNA sequence is **both** longer than 250 bases **AND** has no ambiguous bases, include it in the analysis"
- "If a gene is **either** highly expressed **OR** shows significant differential expression, flag it for further study"
- "If a sample is **NOT** properly labeled, skip it and log a warning"

These operators (`and`, `or`, `not`) work similarly to the way we combine conditions in everyday language. Just as you might say "I'll go for a run if it's not raining AND the temperature is above 60°F," you can write code that makes decisions based on multiple criteria.

Here are a couple of examples:

```
# In a sequence quality filtering pipeline
#
# Both conditions must be true
if sequence_length >= 250 and quality_score >= 30:
    keep(sequence)

# In a variant calling pipeline
#
# Either condition being true is sufficient
if mutation_frequency > 0.01 or supporting_reads >= 100:
    report(variant)

# In a data validation step
#
# Triggers if the condition is false
if not sample_id.startswith('PROJ_'):
    warn_user(sample_id)
```

Think of these operators as the digital equivalent of the decision-making process you use in the lab: checking multiple criteria before proceeding with an experiment, or having alternative procedures based on different conditions.

Behavior of logical operators

Let's explore how Python's logical operators (`and`, `or`, `not`) work, using examples relevant to biological data analysis.

Think of these operators as ways to check multiple conditions, similar to how you might design experimental criteria:

- `and`: Like requiring ALL criteria to be met (e.g., both proper staining AND correct cell count)
- `or`: Like accepting ANY of several criteria (e.g., either elevated temperature OR positive test result)
- `not`: Like reversing a condition (e.g., NOT contaminated)

Here's a truth table showing all possible combinations.

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True

A	B	A and B	A or B	not A
False	False	False	False	True

Here are the rules:

- `and` only gives True if both conditions are True (like requiring all quality checks to pass)
- `or` gives True if at least one condition is True (like having multiple acceptable criteria)
- `not` flips True to False and vice versa (like converting "passed QC" to "failed QC")

Interestingly, Python can also evaluate non-boolean values (values that aren't strictly True or False) using these operators. We call values that Python treats as True "truthy" and values it treats as False "falsy". This becomes important when working with different types of data in your programs and analysis pipelines.

Understanding "Truthy" and "Falsy" Values

In Python, every value can be interpreted as either "true-like" (truthy) or "false-like" (falsy) when used in logical operations. This is similar to how in biology, we might categorize results as "positive" or "negative" even when the underlying data is more complex than a simple yes/no.

Think of "falsy" values as representing empty, zero, or null states – essentially, the absence of meaningful data. Python considers the following values as "falsy":

- `False`: The boolean False value
- `None`: Python's way of representing "nothing" or "no value" (like a blank entry in a spreadsheet)
- Any form of zero (like `0`, `0.0`)
- Empty containers:
  - Empty string (`""`)
  - Empty list (`[]`)
  - Empty set (`set()`)
  - Empty dictionary (`{}`)

Everything else is considered "truthy" - meaning it represents the presence of some meaningful value or data.

Let's look at some practical examples. We can use Python's `bool()` function to explicitly check whether Python considers a value truthy or falsy:

```
# Examples from sample processing:
sample_count = 0
# False (no samples)
print(bool(sample_count))

sample_ids = []
# False (empty list of IDs)
print(bool(sample_ids))

patient_data = {}
 # False (empty data table)
print(bool(patient_data))

# Compare with:
sample_count = 5
# True (we have samples)
print(bool(sample_count))

sample_ids = ["A1", "B2"]
# True (we have some IDs)
print(bool(sample_ids))

patient_data = {"age": 45}
# True (we have some data)
print(bool(patient_data))
```

```
False
False
False
True
True
True
```

Understanding truthy and falsy values becomes particularly useful when writing conditions in your code, like checking whether you have data before proceeding with analysis:

```
# Sort of like saying: if there are some samples IDs,
# then do something with them.
if sample_ids:
    process_samples(sample_ids)
else:
    print("No samples to process")
```

We'll see more examples of how this concept is useful in practice as we work through more advanced topics.

Even More Details About `and` and `or`

Note: This section is a bit low-level, so don't worry too much about it. It's just here for your reference.

One kind of neat thing about the logical operators is that you can directly use them as a type of control flow.

`and`

Given an expression `a and b`, the following steps are taken:

1. First, evaluate `a`.
2. If `a` is "falsy", then return the value of `a`.
3. Otherwise, evaluate `b` and return its value.

Check it out:

```
a = "apple"
b = "banana"
result = a and b
print(result)

name = "Maya"
age = 45
result = age >= 18 and f"{name} is an adult"
print(result)

name = "Amira"
age = 15
result = age >= 18 and f"{name} is an adult"
print(result)
```

```
banana
Maya is an adult
False
```

Were the values assigned to `result` what you expected?

`or`

Given an expression `a or b`, the following steps are taken:

1. First, evaluate `a`.
2. If `a` is "truthy", then return the value of `a`.
3. Otherwise, evaluate `b` and return its value.

Let's return to the previous example, but this time we will use `or` instead of `and`.

```
a = "apple"
b = "banana"
result = a or b
print(result)

name = "Maya"
age = 45
# Observe that this code isn't really doing what we want it to do.
# `result` will be True, rather than "Maya is an adult".
# That's because it should be using `and`
#   ...again, it's just for illustration.
result = age >= 18 or f"{name} is an adult"
print(result)

name = "Amira"
age = 15
# This code is a bit obscure, and you probably wouldn't
# write it like this in practice.  But it illustrates the
# point.
result = age >= 18 or f"{name} is not an adult"
print(result)
```

```
apple
True
Amira is not an adult
```

Were the values assigned to `result` what you expected?

1.4 Control Flow

Think of control flow as the decision-making logic in your code - like following a lab protocol, but for data analysis. Just as you make decisions in the lab ("if the pH is too high, add buffer"), your code needs to make decisions about how to handle different situations.

Control flow statements are the programming equivalent of those decision points in your protocols. They let your program take different paths depending on the conditions it encounters, much like how you might follow different steps in an experiment based on your observations.

In this section, we'll cover several ways to build these decision points into your code:

- Simple `if` statements (like "if the sequence quality is low, skip it")
- `if-else` statements (like "if the gene is expressed, mark it as active; otherwise, mark it as inactive")
- `if-elif-else` chains (for handling multiple possibilities, like different ranges of p-values)
- Nested conditions (for more complex decisions, like filtering sequences based on multiple quality metrics)

Control flow is essential for writing programs that can:

- Make decisions based on data
- Handle different scenarios
- Respond to user input
- Conditionally process data

Just as following the right branch points in a protocol is crucial for experimental success, proper control flow is key to writing programs that correctly handle your data.

Let's explore the main types of control flow in Python:

`if` Statements

Think of these as your basic yes/no checkpoints, like checking if a sample meets quality control:

```
quality_score = 35
if quality_score > 30:
    print("Sample passes QC")
```

```
Sample passes QC
```

`if-else` Statements

These handle two alternative outcomes, like categorizing genes as expressed or not expressed:

```
expression_level = 1.5
if expression_level > 1.0:
    print("Gene is upregulated")
else:
    print("Gene is not upregulated")
```

```
Gene is upregulated
```

`if-elif-else` Chains

Perfect for handling multiple possibilities, like categorizing p-values or expression levels:

```
p_value = 0.03
if p_value < 0.01:
    print("Highly significant")
elif p_value < 0.05:
    print("Significant")
else:
    print("Not significant")
```

```
Significant
```

Multiple Conditions

Sometimes you need to check multiple criteria, like filtering sequencing data:

```
read_length = 100
gc_content = 0.45
quality_score = 35

if read_length >= 100 and quality_score > 30 and 0.4 <= gc_content <= 0.6:
    print("Read passes all quality filters")
else:
    print("Read filtered out")
```

```
Read passes all quality filters
```

Key Points to Remember

- Conditions are checked in order from top to bottom
- Only the first matching condition's code block will execute
- Keep your conditions clear and logical, like a well-designed experimental workflow
- Try to avoid deeply nested conditions as they can become confusing

Think of control flow as building decision points into your data analysis pipeline. Just as you wouldn't proceed with a PCR if your DNA quality was poor, your code can automatically make similar decisions about data processing.

Nested Conditional Statements

Conditional statements can also be nested. Here is some code that is checking if someone can go to the beach. If they are not at work, and the weather is sunny, then they can go to the beach.

```
at_work = False
weather = "sunny"

if weather == "sunny" and not at_work:
    print("It's sunny and you are not at work, let's go to the beach!")
else:
    print("We can't go to the beach today for some reason.")

# Let's move the check for at_work nested inside the if statement that checks
# the weather.
#
# Note that this code isn't equivalent to the previous code, just an example
# of nesting.

if weather == "sunny":
    if at_work:
        print("You are at work and can't go to the beach.")
    else:
        print("It's sunny and you are not at work, let's go to the beach!")
else:
    print("It's not sunny, so we can't go to the beach regardless.")

# Just to be clear, let's "unnest" that conditional.
if weather == "sunny" and at_work:
    print("You are at work and can't go to the beach.")
```

```
elif weather == "sunny":
    print("It's sunny and you are not at work, let's go to the beach!")
else:
    print("It's not sunny, so we can't go to the beach regardless.")
```

```
It's sunny and you are not at work, let's go to the beach!
It's sunny and you are not at work, let's go to the beach!
It's sunny and you are not at work, let's go to the beach!
```

A Note on Keeping Things Simple

Just as you want to keep your experimental protocols clear and straightforward, the same principle applies to writing conditional statements in your code. Think of deeply nested if-statements like trying to follow a complicated diagnostic flowchart - the more branches and decision points you add, the easier it is to lose track of where you are.

For example, imagine designing a PCR troubleshooting guide where each problem leads to three more questions, each with their own set of follow-up questions. While technically complete, it would be challenging for anyone to follow correctly. The same goes for code – when we stack too many decisions inside other decisions, we're setting ourselves up for confusion.

Here's why keeping conditions simple matters:

- Each decision point is an opportunity for something to go wrong (like each step in a protocol)
- Complex nested conditions are harder to debug (like trying to figure out where a multi-step experiment went wrong)
- Simple, clear code is easier for colleagues to review and understand

When you find yourself writing deeply nested conditions, it's often a sign to step back and consider whether there's a clearer way to structure your code.

1.5 Basic Built-in Functions

Think of Python's built-in functions as your basic laboratory toolkit - they're always there when you need them, no special setup required. These functions will become your go-to tools for handling biological data, from DNA sequences to experimental measurements.

Here are some of the most useful built-in functions you'll use regularly:

- `print()`: Displays your data or results
- `len()`: Counts the length of something

- `abs()`: Gives you the absolute value
- `round()`: Tidies up decimal numbers
- `min()` and `max()`: Find the lowest and highest values
- `sum()`: Adds up a collection of numbers
- `type()`: Tells you what kind of data you're working with (helpful for debugging)

Let's look at some examples:

```
# Printing experimental results
print("Gene expression analysis complete!")

# Checking sequence length
dna_sequence = "ATCGATCGTAGCTAGCTAG"
length = len(dna_sequence)
print(f"This DNA sequence is {length} base pairs long.")

# Working with expression fold changes
fold_change = -2.5
absolute_change = abs(fold_change)
print(f"The absolute fold change is {absolute_change}x.")

# Cleaning up p-values
p_value = 0.0000234567
rounded_p = round(p_value, 6)
print(f"p-value = {rounded_p}")

# Analyzing multiple expression values
expression_levels = [10.2, 5.7, 8.9, 12.3, 6.8]
lowest = min(expression_levels)
highest = max(expression_levels)
print(f"Expression range: {lowest} to {highest}")

# Calculating average coverage
coverage_values = [15, 22, 18, 20, 17]
average_coverage = sum(coverage_values) / len(coverage_values)
print(f"Average sequencing coverage: {average_coverage}x")

# Checking data types
gene_name = "nrdA"
data_type = type(gene_name)
print(f"The variable gene_name is of type: {data_type}")
```

```
Gene expression analysis complete!
```

```
This DNA sequence is 19 base pairs long.
The absolute fold change is 2.5x.
p-value = 2.3e-05
Expression range: 5.7 to 12.3
Average sequencing coverage: 18.4x
The variable gene_name is of type: <class 'str'>
```

To use these functions, just type the function name followed by parentheses containing your data (the "arguments"). Some functions, like `min()` and `max()`, can handle multiple inputs, which is handy when comparing several values at once.

1.6 Wrap-Up

In this tutorial, we covered the fundamental building blocks of Python programming that you'll use throughout your bioinformatics work:

- **Variables** help you store and manage data with meaningful names
- **Data types** like numbers, strings, and booleans let you work with different kinds of biological data
- **Control flow** statements help your programs make decisions based on data
- **Built-in functions** provide essential tools for common programming tasks

Remember:

- Choose clear, descriptive variable names
- Be mindful of data types when performing operations
- Keep conditional logic as simple as possible
- Make use of Python's built-in functions for common tasks

These basics form the foundation for more advanced programming concepts we'll explore in future tutorials. Practice working with these fundamentals – they're the tools you'll use to build more complex bioinformatics applications.

Don't worry if everything hasn't clicked yet. Programming is a skill that develops with practice. Focus on understanding one concept at a time, and remember that you can always refer back to this tutorial as a reference.

Next up, we'll build on these basics to work with more complex data structures and write functions of our own!

2 Collections

In this tutorial, we'll explore Python's fundamental data structures and collections – the building blocks that help organize and analyze biological data effectively. From strings for handling DNA sequences to dictionaries for mapping genes to functions, you'll learn how to use these tools through practical examples. We'll cover when and why to use each type, giving you the foundation needed to tackle real bioinformatics problems.

2.1 Introduction to Python Collections

What are collections?

Collections in Python are containers that can hold multiple items, and provide convenient ways to store, access, and manipulate groups of related values.

Think of collections like different types of containers:

- A list is like a row of boxes where you can store items in order
- A tuple is similar but locked/sealed (immutable)
- A dictionary is like a filing cabinet with labeled folders (keys) containing items (values)
- A range represents a sequence of numbers stored in an efficient way

Collections let us:

- Group related data together
- Process multiple items efficiently
- Organize information in meaningful ways
- Access data using consistent patterns

Why we need different data structures

Python provides different collection types because different tasks require different tools. For example:

- If you need to store multiple DNA sequences in order and have fast access to them, use a List

- If you need to group various pieces of data together and ensure they don't change, use a Tuple
- If you need to look up protein functions by their names, use a Dictionary
- If you need to generate sample numbers efficiently, use a Range

Using the right data structure for the job optimizes both speed and code clarity. As we progress through this tutorial, you'll learn which data structures work best in different situations.

Common Python Data Structures at a Glance

We will break down the specifics of each type soon, but let's look first at a quick example of each type:

A list is a mutable, ordered collection of items:

```
nucleotides = ["A", "T", "C", "G"]
print(nucleotides)

# This is a for loop. We will talk more about them below.
for nucleotide in nucleotides:
    print(nucleotide)
```

```
['A', 'T', 'C', 'G']
A
T
C
G
```

A tuple is an immutable, ordered collection of items:

```
# (name, code, molecular_weight)
alanine = ("Alanine", "Ala", 89.1)
print(alanine)
```

```
('Alanine', 'Ala', 89.1)
```

A dictionary is a mapping from keys to values:

```python
# Dictionary -- key-value pairs (gene id -> function)
gene_functions = {
    "TP53": "tumor suppression",
    "BRCA1": "DNA repair",
    "INS": "insulin production"
}
print(gene_functions)

for gene, function in gene_functions.items():
    print(f"{gene} => {function}")
```

```
{'TP53': 'tumor suppression', 'BRCA1': 'DNA repair', 'INS': 'insulin production'}
TP53 => tumor suppression
BRCA1 => DNA repair
INS => insulin production
```

A range is a representation of a sequence of numbers:

```python
# 96 well plate positions
sample_ids = range(1, 96)
print(sample_ids)
```

```
range(1, 96)
```

Notice that each collection has a dedicated syntax for creating it. This makes it easy to create collections and gives you a visual cue for which collection you're working with.

- Lists are formed using square brackets (`[]`)
- Tuples are created with parentheses (`()`)
- Dictionaries use curly brackets (`{}`) and colons (`:`)
- Ranges are generated by the `range()` function

Being able to recognize these collection types and know when to use each is critical to both writing and reading code. Let's explore them further.

Note: Python contains other useful data structures, including sets, but we won't cover them in this tutorial.

2.2 Strings

In Python, strings are ordered collections of characters, meaning they are sequences that can be indexed, sliced, and iterated over just like other sequence types (such as lists and tuples), with each character being an individual element in the collection.

Though we covered strings in Tutorial 1, let's go over some basics again so that you have it here for easy reference.

String Literals

In Python, text data is handled with str objects, or strings. You can build strings with string literals:

```
# With single quotes
'a string'

# With double quotes
"another string"

# Triple quoted
"""Here is a string."""
'''And here is another.'''
```

```
'And here is another.'
```

If you need to embed quote marks within a string literal, you can do something like this:

```
# Double quote in single quoted string
'This course is "fun", right?'

# Single quote in double quoted string
"Of course! It's my favorite class!"
```

```
"Of course! It's my favorite class!"
```

There are also escape sequences for including different kinds of text inside a string literal. Tabs and newlines are some of the more common escape sequences:

```
# Tabs
print("name\tage")

# Newlines
print("gene 1\ngene 2")
```

```
name    age
gene 1
gene 2
```

String Methods

In addition to common operations like indexing, slicing, and concatenation, strings have a rich set of functionality provided by string methods.

A string method is essentially a function that is "attached" to a string. Some common string methods are:

- `upper`, `lower` – Case conversion
- `strip`, `lstrip`, `rstrip` – Remove whitespace
- `split` – Convert string to list based on delimiter
- `join` – Combine list elements into string
- `replace` – Replace substring
- `find`, `index` – Find substring position
- `startswith`, `endswith` – Check string prefixes/suffixes
- `count` – Count substring occurrences

Let's go through them now.

Case Conversion

The `upper` and `lower` methods convert strings to uppercase or lowercase. This is useful for standardizing text or making case-insensitive comparisons.

```
dna = "ATCGatcg"

print(dna.upper())
print(dna.lower())

fragment_1 = "ACTG"
fragment_2 = "actg"
```

```
# You can convert both sequences to lower case before
# comparing them for a case-insensitive comparison.
print(fragment_1.lower() == fragment_2.lower())
```

```
ATCGATCG
atcgatcg
True
```

Remove Whitespace

The `strip` method remove whitespace characters (spaces, tabs, newlines). `strip` removes from both ends, while `lstrip` and `rstrip` remove from left or right only. This is particularly useful when cleaning up input data.

```
dna_sequence = "  ATCG\n"
print(dna_sequence.strip())

gene_name = "nrdA    "
print(gene_name.rstrip())
```

```
ATCG
nrdA
```

Convert String To List

The `split` method divides a string into a list of substrings based on a delimiter. By default, it splits on whitespace. This is useful for parsing formatted data.

```
fasta_header = ">sp|P00452|RIR1_ECOLI Ribonucleoside-diphosphate reductase 1"
fields = fasta_header.split("|")
print(fields)
```

```
['>sp', 'P00452', 'RIR1_ECOLI Ribonucleoside-diphosphate reductase 1']
```

Check out this neat trick where Python will let us put the different fields directly into named variables.

```
_, uniprot_id, protein_info = fasta_header.split("|")

print(f"{uniprot_id} => {protein_info}")
```

```
P00452 => RIR1_ECOLI Ribonucleoside-diphosphate reductase 1
```

Pretty useful! (We will see more about this in the section on tuples.)

Combine List Into String

The `join` method combines a list of strings into one, using the string it's called on as a delimiter. This is useful for creating formatted output.

```
amino_acids = ["Met", "Gly", "Val"]
protein = "-".join(amino_acids)
print(protein)

fields = ["GeneName", "Length", "Count"]
tsv_line = "\t".join(fields)
print(tsv_line)
```

```
Met-Gly-Val
GeneName    Length  Count
```

Replace Substring

The `replace` method substitutes all occurrences of a substring with another. This is helpful for sequence modifications or text cleanup, like turning a DNA string into an RNA string.

```
dna = "ATCGTTA"
rna = dna.replace("T", "U")
print(rna)
```

```
AUCGUUA
```

Find Substring Position

The `find` and `index` methods locate the position of a substring. `find` returns `-1` if not found, while `index` raises an error. These are useful for sequence analysis.

```
sequence = "ATCGCTAGCT"
position = sequence.find("GCT")
print(position)

try:
    position = sequence.index("NNN")
    print(position)
except ValueError:
    print("not found!")
```

```
3
not found!
```

Don't worry too much now about this try/except construction for now – we will cover it in a later tutorial! Basically, it is a way to tell Python that we think an error may occur here, and if it does, what we should do to recover.

Check String Prefix/Suffix

The `startswith` and `endswith` methods check if a string begins or ends with a given substring. These are helpful for parsing user input, or validating sequence patterns and file names.

```
gene = "ATGCCGTAA"
print(gene.startswith("ATG"))
print(gene.endswith("TAA"))
```

```
True
True
```

Count Substring Occurrences

The `count` method counts how many times a substring appears in a string. This is useful for sequence analysis and pattern counting.

```
dna = "ATAGATAGATAG"
tag_count = dna.count("TAG")
print(tag_count)
```

```
3
```

String Summary

In Python, strings are immutable sequences of characters (including letters, numbers, symbols, and spaces) that are used to store and manipulate text data. They can be created using single quotes (`''`), double quotes (`""`), or triple quotes (`''' '''` or `""" """`) and support various built-in methods for operations like searching, replacing, splitting, and formatting text.

(For more info about string indexing, slicing, etc., see Tutorial 1.)

2.3 Lists

Lists are going to be one of your best friends in Python – they're flexible, easy to modify, and good for handling biological sequences and experimental data.

Creating Lists

You can create lists using square brackets `[]` and assign them to variables. As always, keep in mind best practices for naming variables!

```
# A DNA sequence
dna_sequence = ["A", "T", "G", "C", "T", "A", "G"]

# Gene names in a pathway
pathway_genes = ["TP53", "MDM2", "CDKN1A", "BAX"]

# Expression values
expression_levels = [0.0, 1.2, 3.4, 2.1, 0.8]

# Mixed data types (though it may be best to avoid mixing types like this)
sample_info = ["SAMPLE001", 37.5, "positive", True]

# Empty list to fill later
results = []
```

Creating an empty list might seem a bit weird, but is actually common practice in Python – create an empty list and then use a loop to store multiple things in it. We will see examples of this later in the tutorial.

List Indexing and Slicing

Remember that a list is like a row of boxes, each with something inside. The boxes are in a particular order and each has a number that you can use to access the data inside (the index).

You could imagine a list looking something like this:

```
 "A"   "T"   "G"   "A"   "C"    (values in the list)

  0     1     2     3     4     (indices of the values)
```

Which corresponds to the following Python code:

```
nucleotides = ["A", "T", "G", "A", "C"]
# index         0    1    2    3    4
```

Don't forget that Python starts counting with 0 rather than with 1.

*Note: For now, don't worry too much right now about **how** Python stores items in a list. Later in the tutorial, we will adjust our mental model for collections.*

Indexing

Similar to strings, you can get specific things out of a list with `list_name[]` syntax, which is sometimes called "indexing" the list. The most basic option is to grab items one at time:

```
# Get single elements
dna = "ATGC"
first_base = dna[0]
third_base = dna[2]
```

Just like with strings, you can also start indexing from the end of a list. Try to predict the outcome before uncommenting the `print()` statement.

```
mystery_base = dna[-1]
# print(mystery_base)
```

Slicing

If you want to get chunks of a list, you can use "slicing":

```
dna = "ACTGactgACTG"
first_four = dna[0:4]
middle_section = dna[4:8]

print(first_four)
print(middle_section)
```

```
ACTG
actg
```

You can leave off the beginning or the end of a slice as well:

```
dna = "ACTGactgGGGG"

# From index 4 to the end
print(dna[4:])

# From the beginning up to index 4, but *excluding* 4.
print(dna[:4])
```

```
actgGGGG
ACTG
```

Slices can get pretty fancy. Check this out:

```
dna = "AaTtCcGg"

# Get every other base, starting from the beginning.
every_second = dna[::2]
print(every_second)

# Get every other base starting from index 1
every_other_second = dna[1::2]
print(every_other_second)
```

```
ATCG
atcg
```

There are quite a few rules about slicing, which can get a bit complicated. For this reason, it's generally best to keep your slicing operations as simple as possible.

List Methods

Similar to strings, lists come with some methods that let you modify them or get information about them. Some of the most common are:

- `append`
- `insert`
- `pop`
- `sort`
- `count`

Let's take a look.

Adding Items to Lists

```
genes = ["TP53"]

# Adds to the end
genes.append("BRCA1")

# Adds at specific position
genes.insert(0, "MDM2")

# Adds multiple items
genes.extend(["ATM", "PTEN"])
```

Based on the information in the comments, what does our list look like now? Try to figure that out before running the next code block.

```
print(genes)
```

```
['MDM2', 'TP53', 'BRCA1', 'ATM', 'PTEN']
```

Removing Items from Lists

We know how to add items now, but what about removing them? There are several ways to do that as well:

```
genes = ["MDM2", "TP53", "BRCA1", "ATM", "PTEN"]

# Removes by value
genes.remove("BRCA1")
print(f"remaining genes: {genes}")

# Removes and returns last item
last_gene = genes.pop()
print(f"last_gene: {last_gene}, remaining genes: {genes}")

# Removes and returns item at index
specific_gene = genes.pop(0)
print(f"specific_gene: {specific_gene}, remaining genes: {genes}")
```

```
remaining genes: ['MDM2', 'TP53', 'ATM', 'PTEN']
last_gene: PTEN, remaining genes: ['MDM2', 'TP53', 'ATM']
specific_gene: MDM2, remaining genes: ['TP53', 'ATM']
```

Pay attention to `pop` in particular. While `remove` just takes a value out of our list, `pop` removes the item *and* returns it, which is what allows us to save it to a variable.

Other Useful List Methods

There are many other cool list methods. Here are a few more. Try to guess what the output will be before running the code block.

```
genes = ["MDM2", "TP53", "BRCA1", "ATM", "PTEN", "TP53"]

genes.sort()
print(genes)

genes.reverse()
print(genes)

print(genes.count("TP53"))
```

```
['ATM', 'BRCA1', 'MDM2', 'PTEN', 'TP53', 'TP53']
['TP53', 'TP53', 'PTEN', 'MDM2', 'BRCA1', 'ATM']
2
```

List Operations

We talked about operators in Tutorial 1. These operators can also be applied to lists in various ways. Let's check it out.

Similar to strings, you can concatenate lists into a single list using `+`:

```
forward_primers = ["ATCG", "GCTA"]
reverse_primers = ["TAGC", "CGAT"]
all_primers = forward_primers + reverse_primers
print(all_primers)
```

```
['ATCG', 'GCTA', 'TAGC', 'CGAT']
```

Take a small list a "multiply" its components to make a bigger list using `*`:

```
# Creates a poly-A sequence
poly_a = ["A"] * 20
print(poly_a)
```

```
['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A']
```

Check if something is in a list using `in`:

```
genes = ["MDM2", "TP53", "BRCA1", "ATM", "PTEN", "TP53"]

# Checking membership
if "TP53" in genes:
    print("TP53 present in our pathway")

if "POLA" in genes:
    print("POLA is not found!")
```

```
TP53 present in our pathway
```

And get the length of a list using `len`:

```
samples = ["Treatment_1", "Control_1", "Treatment_2", "Control_2"]
total_samples = len(samples)
print(total_samples)
```

```
4
```

Nested Lists

Lists can contain other lists, useful for representing things like matrices, graph connections, and simple hierarchical data.

```
# Matrix
sequences = [
    ["A", "T", "G", "C"],
    ["G", "C", "T", "A"],
    ["T", "A", "G", "C"]
]
print(sequences)

# Coordinates
coordinates =[
    [1, 2],
    [3, 4],
    [5, 6]
]
print(coordinates)

# Simple hierarchical data: Experimental data with replicates
expression_data = [
    ["Gene1", [1.1, 1.2, 1.0]],  # Gene name and replicate values
    ["Gene2", [2.1, 2.3, 1.9]],
    ["Gene3", [0.5, 0.4, 0.6]]
]
print(expression_data)
```

```
[['A', 'T', 'G', 'C'], ['G', 'C', 'T', 'A'], ['T', 'A', 'G', 'C']]
[[1, 2], [3, 4], [5, 6]]
[['Gene1', [1.1, 1.2, 1.0]], ['Gene2', [2.1, 2.3, 1.9]], ['Gene3', [0.5, 0.4, 0.6]]]
```

Many times, there will be a better solution to your problem than nesting lists in this way, but it's something that you should be aware of should the need arise.

Nested lists can be accessed just like regular lists, but there will be more "layers" to get through depending on what you want out of them.

```
# Accessing nested data

first_sequence = sequences[0]
print(first_sequence)

gene2_rep2 = expression_data[1][1][1]
print(gene2_rep2)
```

```
['A', 'T', 'G', 'C']
2.3
```

Lists are very flexible in Python, and so can be complicated. However, it will be good for you to get comfortable with Lists as they are one of the most commonly used data structures!

What Does Python Actually Store in the List?

When working with lists and other collections in Python, there's a crucial detail about how Python manages data that might seem counterintuitive at first. Let's explore this through a simple example using 2D points.

First, let's create some points and store them in a list:

```
# Represent points as [x, y] coordinates
point_a = [0, 3]
point_b = [1, 2]

# Store points in a list
points = [point_a, point_b]
print(points)
```

```
[[0, 3], [1, 2]]
```

We can access individual coordinates using nested indexing:

```
# Get the y-coordinate of the first point
print(points[0][1])
# Get the x-coordinate of the second point
print(points[1][0])
```

```
3
1
```

Now here's where things get interesting. Let's modify some values:

```
# Double the y-coordinate of the first point
points[0][1] *= 2
print(points)

# Now modify the original point_b
point_b[0] *= 10
print(point_b)

# What do you think our points list looks like now?
print(points)

# Also, we modified the first point via the list.
# What do you think `point_a` variable now contains?
print(point_a)
```

```
[[0, 6], [1, 2]]
[10, 2]
[[0, 6], [10, 2]]
[0, 6]
```

Did the last result surprise you? When we modified `point_b`, the change was reflected in our `points` list too! This happens because Python doesn't actually store the values directly in the list – instead, it stores references (think of them as pointers) to the data. It's like having a directory of addresses rather than copies of the actual data.

Understanding this behavior is important because it means changes to your data in one place can unexpectedly affect the same data being used elsewhere in your code.

With this in mind, we can now update our mental model and make it a bit more accurate. This time, the items in the lists are references that "point" to the actual items we care about.

```
"A"   "T"   "G"   "A"   "C"    (items "in" the list are objects)
 ↑     ↑     ↑     ↑     ↑

                          (values in the list are references)

 0     1     2     3     4     (indices of the references)
```

The diagram for the points example might look something like this:

```
  0     3       1     2      (items "in" the list are numbers)
  ↑     ↑       ↑     ↑

                        (each element in `points` is also a list)

  ↑     ↑       ↑     ↑


                           (the first level is the `points` list)
```

For now, don't get *too* hung up on the lower-level details – just be aware of the practical implications mentioned above.

2.4 Loops

So far, we've worked with two types of collections: lists and strings. But what if you want to work with each element in these collections one at a time? That's where loops come in!

Loops give you a way to automate repetitive tasks. Instead of copying and pasting the same code multiple times to process each item in a list (which would be both tedious and error-prone), loops let you write the instructions once and apply them to every item automatically.

For example, if you had a list of gene sequences and wanted to check each one for a particular pattern, you wouldn't want to write separate code for each sequence. A loop would let you perform this check systematically across your entire dataset.

Python offers several different types of loops, each suited for particular situations. In this section we will focus on for loops and while loops.

For Loops

A for loop processes each item in a sequence, one at a time. Think of it like going through a list and looking at each item one at a time:

```
for letter in ["D", "N", "A"]:
    print(letter)
```

```
D
N
A
```

Let's break that down:

1. `for` – tells Python we want to start a loop
2. `letter` – a variable that will hold each item
3. `in ["D", "N", "A"]` – tells Python to loop through the list `["D", "N", "A"]`
4. `:` – marks the beginning of the code block to be executed
5. The indented code (`print(letter)`) runs once for each item

Note that `for` and `in` are specifically required in for loop syntax. `letter` and `["D", "N", "A"]` will change depending on the context.

For example, this loop has the same behavior as the previous loop:

```
letters = ["D", "N", "A"]
for the_letter in letters:
    print(the_letter)
```

```
D
N
A
```

This time, we used a different variable name to store the items of the collection, and rather than putting the collection directly in the `for ... in ... :` part, we referred to the collection using a variable.

In addition to lists, for loops also work on strings:

```
nucleotides = "ATCG"
for nucleotide in nucleotides:
    print(f"The nucleotide was '{nucleotide}'")
```

```
The nucleotide was 'A'
The nucleotide was 'T'
The nucleotide was 'C'
The nucleotide was 'G'
```

You can actually use for loops on lots of different Python data structures: as long as it is iterable, then you can use a for loop with it.

Often you will want to take some action multiple times. For this, we can use `range`:

```
for number in range(5):
    print(number)
```

```
0
1
2
3
4
```

This should have printed 5 numbers: 0, 1, 2, 3, 4. Here is Python counting from zero again!

You can also tell range where to start and stop:

```
# Count from 1 to 5
for number in range(1, 6):
    print(number)
```

```
1
2
3
4
5
```

Here is a neat thing you can do with ranges. Before running the code, could you guess what it might do?

```
for i in range(2, 10, 2):
    print(i)
```

```
2
4
6
8
```

Let's break down what's happening with `range` here. While we've seen `range` create simple sequences of numbers before, it can actually take up to three arguments: `range(start, stop, step)`. The `step` tells Python how many numbers to count by each time.

It's like counting: normally we count "1, 2, 3, 4..." (step of 1), but sometimes we count "2, 4, 6, 8..." (step of 2). In this example, we're using a step of 2 to skip every other number.

The `start` and `step` arguments are optional – you can just use `range(stop)` if you want to count normally starting from zero. If you're curious about more advanced uses, like counting backwards or working with negative numbers, check out the Python range docs for more details.

Ranges are memory efficient – they don't store all the numbers in the range in memory. This is important when generating large batches of numbers.

(This code shouldn't be run. It's just here to illustrate the point.)

```
big_range = range(1, 1000000)  # Takes very little memory
big_list = list(big_range)     # Takes much more memory!
```

Nested For Loops

One feature of for loops is that you can put one inside another – something we call "nesting". Think of it like those Russian nesting dolls, where each doll contains a smaller one inside.

```
for i in range(2):
    for j in range(3):
        print(f"i: {i}; j: {j}")
```

```
i: 0; j: 0
i: 0; j: 1
i: 0; j: 2
i: 1; j: 0
i: 1; j: 1
i: 1; j: 2
```

Let's break down what's happening here. The outer loop (using `i`) runs two times (0, 1), and for *each* of those times, the inner loop (using `j`) runs three times (0, 1, 2). It's a bit like having a set of drawers where you check each drawer (outer loop), and within each drawer, you look at every item inside (inner loop).

When you run the above code, you'll see each combination of `i` and `j` printed out, showing how the loops work together. This pattern of nested loops is incredibly useful when you need to process data that has multiple levels or dimensions, for example, like comparing every gene in one dataset to every gene in another dataset.

Here is a schematic view:

```
i=0

  j=0           j=1           j=2

  print(...)    print(...)    print(...)



i=1

  j=0           j=1           j=2

  print(...)    print(...)    print(...)
```

You can have more than two levels of nesting. For example:

```
for i in range(2):
    for j in range(3):
        for k in range(4):
            print(f"i: {i}; j: {j}; k: {k}")
```

```
i: 0; j: 0; k: 0
i: 0; j: 0; k: 1
i: 0; j: 0; k: 2
i: 0; j: 0; k: 3
i: 0; j: 1; k: 0
```

```
i: 0; j: 1; k: 1
i: 0; j: 1; k: 2
i: 0; j: 1; k: 3
i: 0; j: 2; k: 0
i: 0; j: 2; k: 1
i: 0; j: 2; k: 2
i: 0; j: 2; k: 3
i: 1; j: 0; k: 0
i: 1; j: 0; k: 1
i: 1; j: 0; k: 2
i: 1; j: 0; k: 3
i: 1; j: 1; k: 0
i: 1; j: 1; k: 1
i: 1; j: 1; k: 2
i: 1; j: 1; k: 3
i: 1; j: 2; k: 0
i: 1; j: 2; k: 1
i: 1; j: 2; k: 2
i: 1; j: 2; k: 3
```

Though I bet you know what's going on with nested loops by now, let's break it down anyway. The innermost loop (`k`) completes all its iterations before the middle loop (`j`) counts up once, and the middle loop completes all its iterations before the outer loop (`i`) counts up once. In this example, for each value of `i`, we'll go through all values of `j`, and for each of those, we'll go through all values of `k`.

Remember that each additional level of nesting multiplies the number of iterations. In our example, we have $2 \times 3 \times 4 = 24$ total iterations. Keep this in mind when working with larger datasets.

Enumerated for Loops

Sometimes when you're working with a sequence, you need to know not just what each item is, but also where it appears. That's where Python's handy `enumerate` function comes in. It lets you track both the position (index) and the value of each item as you loop through them.

Here's a simple example:

```
for index, letter in enumerate("ABCDE"):
    print(f"index: {index}; letter: {letter}")
```

```
index: 0; letter: A
index: 1; letter: B
index: 2; letter: C
index: 3; letter: D
index: 4; letter: E
```

This will show you each letter along with its position in the sequence, starting from 0 (remember, Python always starts counting at 0!).

By the way, you can also use `enumerate` outside of loops. For instance, if you have a list of nucleotides:

```
nucleotides = ["A", "C", "T", "G"]
enumerated_nucleotides = enumerate(nucleotides)
print(list(enumerated_nucleotides))
```

```
[(0, 'A'), (1, 'C'), (2, 'T'), (3, 'G')]
```

This creates pairs of positions and values, which can be useful, say, when you need to track where certain elements appear in your sequence data.

While Loops

While loops keep repeating until the given condition is not true (or truthy). Let's look at a simple example that counts from 1 to 5:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

```
1
2
3
4
5
```

To understand what this loop does, imagine it following a simple set of instructions:

- Create a variable called `count` and set it to 1.
- Then, keep doing these steps as long as `count` is less than or equal to 5:

- Display the current value of `count`
- Add 1 to `count`.

The loop will keep running until `count` becomes 6, at which point the condition `count <= 5` becomes false, and the loop stops.

Just to make it super clear, let's write out the steps:

1. `count = 1`: is count <= 5? Yes! prints 1, then adds 1
2. `count = 2`: is count <= 5? Yes! prints 2, then adds 1
3. `count = 3`: is count <= 5? Yes! prints 3, then adds 1
4. `count = 4`: is count <= 5? Yes! prints 4, then adds 1
5. `count = 5`: is count <= 5? Yes! prints 5, then adds 1
6. `count = 6`: is count <= 5? No! stops because 6 is not <= 5

Infinite Loops and Other Problems

When working with while loops, it's crucial to ensure your loop has a way to end. Think of it like setting up an automated process – you need a clear stopping point, or the process will run forever!

There are two common pitfalls to watch out for:

1. If your condition is never true to begin with, the loop won't run at all
2. If your condition can never become false, the loop will run forever (called an *infinite loop*)

Here's an example of the 2nd problem. Can you figure out why this code would run forever?

```
# Infinite loop -- DO NOT RUN!!
count = 1
while count >= 0:
    print(count)
    count = count + 1
```

Let's think through what's happening:

- We start with `count = 1`
- The loop continues as long as `count` is greater than or equal to 0
- Each time through the loop, we're adding 1 to `count`
- So `count` keeps getting bigger: 1, 2, 3, 4, 5…
- But wait! A number that keeps getting bigger will always be greater than 0
- This means our condition (`count >= 0`) will always be true, and the loop will never end!

When writing your own loops, always be sure that your condition will eventually become false – you need a clear endpoint!

Modifying a List While Looping

One tricky aspect of using loops in Python occurs if you try to modify a collection while looping over it.

With a while loop and the `pop` method, it's not too weird – you run the while loop until the list is empty:

```
# Starting with a list of tasks
todo_list = ["task1", "task2", "task3"]

while todo_list:  # This is true as long as the list has items
    current_task = todo_list.pop()  # removes and returns last item
    print(f"Doing task: {current_task}")

print("All tasks complete!")
print(todo_list)
```

```
Doing task: task3
Doing task: task2
Doing task: task1
All tasks complete!
[]
```

However, things can get quite weird with for loops:

```
# This is probably not what you want!
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    numbers.remove(number)  # Don't do this!
print(numbers)
```

```
[2, 4]
```

Unfortunately, that did not remove all the items from `numbers` like you may have expected.

One way to address this issue is to use `[:]` to create a copy of `numbers` and iterate over *that* collection. Meanwhile, you remove items from the original `numbers`.

```python
numbers = [1, 2, 3, 4, 5]
for number in numbers[:]:  # The [:] creates a copy
    numbers.remove(number)
    print(f"Removed {number}. List is now: {numbers}")
print(f"at the end: {numbers}")
```

```
Removed 1. List is now: [2, 3, 4, 5]
Removed 2. List is now: [3, 4, 5]
Removed 3. List is now: [4, 5]
Removed 4. List is now: [5]
Removed 5. List is now: []
at the end: []
```

Really, this example is pretty artificial – you wouldn't be trying to delete every item in a list with a for loop anyway. Just be aware that if you modify a collection during a loop, special care must be taken to ensure that you don't mess things up.

Take note of this for miniproject 1 – you will "probably" have to remove some items from a list to complete it! But don't worry, you will see some more examples in the project description….

Comprehensions

While we are on the topic of loops, let's discuss one more thing: Comprehensions.

Comprehensions let you create new lists (and other collections) from existing lists (and other collections).

Let's say that you want to create a list of RNA bases and you've already made a list of DNA bases. One way to do this would be to take your existing list and convert any Thymines (T) to Uracils (U). We can do this with a traditional for loop:

```python
# Using traditional loop
dna = ["A", "T", "G", "C"]
rna = []
for base in dna:
    if base != "T":
        rna.append(base)
    else:
        rna.append("U")

print(rna)
```

```
['A', 'U', 'G', 'C']
```

Or with a comprehension:

```
dna = "ATGC"
rna = ["U" if base == "T" else base for base in dna]
print(rna)
```

```
['A', 'U', 'G', 'C']
```

The comprehension is much more concise! The list comprehension is doing everything that the traditional for loop is doing, but in a single line.

The basic structure of a comprehension can be broken down into these components:

```
new_list = [expression for item in iterable if condition]
```

Breaking it down:

- `new_list`: The resulting list
- `expression`: What you want to do with each item (like transform it)
- `for item in iterable`: The loop over some iterable object
- `if condition`: Optional filter (you can leave this out)

Note that in our original example, the `if` condition actually came before the `for` loop part – that's allowed!

Comprehensions are definitely weird at first! Let's look at some more examples.

Here is a basic example using `range` instead of an existing list:

```
squares = [x**2 for x in range(5)]
print(squares)

# Same as:
squares = []
for x in range(5):
    squares.append(x**2)

print(squares)
```

```
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
```

This example takes each number in the list produced by `range(5)`, squares it, and adds it to the new list `squares`. In this case:

- `squares` is the `new_list`
- `x**2` is the `expression`
- `x` is the `item` and `range(5)` is the `iterable`
- There is no `if condition`

Notice that you don't have to initialize an empty list for the comprehension to work – it makes the list itself, unlike with a for loop.

Let's look at an example with a condition:

```
# Using comprehension
expressions = [1.2, 0.5, 3.4, 0.1, 2.2]
high_expression = [x for x in expressions if x > 2.0]
print(high_expression)

# Using a for loop
expressions = [1.2, 0.5, 3.4, 0.1, 2.2]
high_expression = []
for x in expressions:
    if x > 2.0:
        high_expression.append(x)
print(high_expression)
```

```
[3.4, 2.2]
[3.4, 2.2]
```

In this example, we take an existing list, `expressions`, and make a new list, `high_expressions`, that contains only the expressions that are 2.0 or greater.

Notice that in this example, there is nothing done to the existing items in the list before adding them to the new one, which is why the comprehension starts with `x for x`.

Comprehensions can also be used to create dictionaries. Check this out:

```
squares = {x: x**2 for x in range(5)}
print(squares)

even_squares = {x: x**2 for x in range(5) if x % 2 == 0}
print(even_squares)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
{0: 0, 2: 4, 4: 16}
```

That is pretty neat right?

While comprehensions are compact, whether or not you think this conciseness leads to better code is a different story. As you gain more experience, you will get a better feel for such things. Whether you use them a lot or a little, you should be aware of them as they are quite common in Python codebases.

2.5 Tuples

Tuples are like lists that can't be changed – perfect for storing fixed information. We often use tuples when we want to ensure data integrity or represent relationships that shouldn't change.

Creating Tuples

Tuples have a dedicated syntax used for construction:

```
letters = ("a", "b", "c")

# Single item tuples still need a comma!
number = (1, )
```

The syntax for creating a tuple is not that different from creating a list, but you'll notice differences when trying to alter their components. For example, the following code would raise an error if we didn't put the `try/except` around it:

```
letters = ("a", "b", "c")

try:
    letters[0] = "d"
except TypeError:
    print("you can't assign to a tuple")
```

```
you can't assign to a tuple
```

If `letters` was a list, the above code would change the list to start with `d` instead of `b`. Instead, we got an error. This is because tuples are *immutable*.

Whether some data is mutable or immutable determines whether you can modify it or not. Here is a silly metaphor to illustrate what I mean:

- **Mutable** collections (like lists and dictionaries) are like erasable whiteboards – you can add, remove, or change items whenever you need to
- **Immutable** collections (like tuples) are more like carved stone tablets – once created, their contents are "set in stone"

Why does this matter? Here are two practical implications:

- **Data Safety**: Immutable collections help prevent accidental changes to important data
  - Remember how we could modify individual coordinates in our list earlier? If we had used tuples instead, Python would have prevented any accidental modifications
- **Technical Requirements**: Some Python features, like using values as dictionary keys (which we'll explore soon), only work with immutable data types

Tuples excel at representing fixed relationships between values that logically belong together. Think of them as a way to package related information that you know shouldn't change during your program's execution.

E.g., our coordinates example from above could be better written with a tuple:

```
# (x, y)
point = (1, 2)
print(point)
```

```
(1, 2)
```

Or, you could represent facts about a codon as a tuple:

```
methionine = ("Methionine", "Met", "M", "ATG")
print(methionine)
```

```
('Methionine', 'Met', 'M', 'ATG')
```

Or, you could represent related gene information:

```
gene_info = ("BRCA1",      # gene name
             "chr17",      # chromosome
             43044295,     # start position
             43125364,     # end position
             "plus")       # strand
print(gene_info)
```

```
('BRCA1', 'chr17', 43044295, 43125364, 'plus')
```

Tuple Packing and Unpacking

Let's look at two really useful Python features that make working with multiple values easier: tuple packing and unpacking.

Tuple packing is pretty straightforward – Python can automatically bundle multiple values into a tuple for you. Here's an example using a codon and its properties:

```
# Packing values into a tuple
codon = "AUG", "Methionine", "Start"
print(codon)
```

```
('AUG', 'Methionine', 'Start')
```

The opposite operation, tuple unpacking, lets you smoothly assign tuple elements to separate variables:

```
# Unpacking a tuple into individual variables
codon = ("AUG", "Methionine", "Start")
sequence, amino_acid, role = codon

print(f"Codon: {sequence}; Amino Acid: {amino_acid}; Role: {role}")
```

```
Codon: AUG; Amino Acid: Methionine; Role: Start
```

One of the coolest applications of packing and unpacking is swapping values between variables. Check this out:

```python
# Set initial values
x, y = 1, 2

# Print the original values
print(f"x: {x}; y: {y}")

# Swap values in one clean line
x, y = y, x

# Print the swapped values
print(f"x: {x}; y: {y}")
```

```
x: 1; y: 2
x: 2; y: 1
```

To appreciate how nice this is, here's how you'd typically swap values in many other programming languages:

```python
x = 1
y = 2

# Print the original values
print(f"x: {x}; y: {y}")

# The traditional way requires a temporary variable
tmp = y
y = x
x = tmp

# Print the swapped values
print(f"x: {x}; y: {y}")
```

```
x: 1; y: 2
x: 2; y: 1
```

Python's packing and unpacking syntax makes this common operation more intuitive and readable. Instead of juggling a temporary variable, you can swap values in a single, clear line of code. This is just one example of how Python's design choices can make your code both simpler to write and easier to understand.

Named Tuples

You may be thinking that it could get tricky to remember which field of a tuple is which. Named tuples provide a great way to address this. They're like regular tuples, but with the added benefit of letting you create them and access data using descriptive names instead of index numbers.

Let's see how they work:

```
# We need to import namedtuple from the collections module
from collections import namedtuple

# Create a Gene type with labeled fields
# (note the name is Gene and not gene)
Gene = namedtuple("Gene", "name chromosome start stop")

# Create a specific gene entry
#
# Using named arguments can keep you from mixing up the arguments!
tp53 = Gene(
    name="TP53",
    chromosome="chr17",
    start=7_571_720,
    stop=7_590_868,
)

# Access the data using meaningful names
print(tp53.name)
print(tp53.chromosome)

# You can still unpack it like a regular tuple if you want
name, chromosome, start, stop = tp53
print(name, chromosome, start, stop)
```

```
TP53
chr17
TP53 chr17 7571720 7590868
```

What makes named tuples great?

- They're clear and self-documenting – the labels tell you exactly what each value means
- They're less prone to errors – no more mixing up whether position 2 was start or stop

- They're efficient and unchangeable (immutable), just like regular tuples

For example, you can't change values after creation:

```
try:
    tp53.start = 1300  # This will raise an error
except AttributeError:
    print("you can't do this!")
```

```
you can't do this!
```

Named tuples are perfect for representing any kind of structured data. Here's another example using DNA sequences:

```
Sequence = namedtuple("Sequence", "id dna length gc_content")

# Create some sequence records
seq1 = Sequence("SEQ1", "GGCTAA", length=6, gc_content=0.5)
seq2 = Sequence("SEQ2", "GGTTAA", length=6, gc_content=0.33)

# Named tuples print out nicely too
print(seq1)  # Shows all fields with their values
print(seq2)
```

```
Sequence(id='SEQ1', dna='GGCTAA', length=6, gc_content=0.5)
Sequence(id='SEQ2', dna='GGTTAA', length=6, gc_content=0.33)
```

I have mentioned a few times now that tuples are immutable, and named tuples are as well. There is a way to get an modified copy of a named tuple however:

```
seq1 = Sequence("SEQ1", "GGCTAA", length=6, gc_content=0.5)

seq1_with_new_id = seq1._replace(id="sequence 1")

# The original seq1 is unchanged:
print(seq1)

# The new one has the same values as the original other than the id
print(seq1_with_new_id)
```

```
Sequence(id='SEQ1', dna='GGCTAA', length=6, gc_content=0.5)
Sequence(id='sequence 1', dna='GGCTAA', length=6, gc_content=0.5)
```

The bottom line: When you need to bundle related data together, named tuples are often a great choice. They're essentially as lightweight as regular tuples, but they make your code much easier to read and maintain. Think of them as regular tuples with the added bonus of built-in documentation!

When to Use Tuples vs. Lists

It may still be unclear when to choose tuples rather than lists. While you will get a feel for it over time, here are some guidelines that can help you choose:

Choose a Tuple When:

- Your data represents an inherent relationship that won't change (like a DNA sequence's start and end coordinates)
- You want to make sure your data stays protected from accidental modifications
- You need to use the data as a dictionary key (we'll explore this more soon)
- You're returning multiple related values from a function

Choose a List When:

- You'll need to add or remove items as your program runs
- Your data needs to be flexible and modifiable
- You're accumulating or building up data throughout your program

One way to think of it is: if you're working with data that should remain constant, reach for a tuple. If you need something more flexible that can grow or change (like collecting results), a list is your better choice.

Here is a nice section of the Python docs if you want to dive deeper: Why are there separate tuple and list data types?

2.6 Dictionaries

Dictionaries in Python are a bit like address books. Just as you can look up someone's phone number using their name, dictionaries let you pair up pieces of information so you can easily find one when you know the other. The first part (like the person's name) is called the *key*, and it leads you to the second part (like their phone number), which is called the *value*.

Let's say you want to keep track of gene names and their functions. Instead of scanning through a long list every time, a dictionary lets you jump straight to the function just by knowing the gene name. They are a great way to organize and retrieve your data quickly.

Creating Dictionaries

Dictionary Literals ({})

The most straightforward way to create dictionaries is using curly brackets `{}` with `key: value` pairs:

```
codon_table = {
    "AUG": "Met",
    "UAA": "Stop",
    "UAG": "Stop",
    "UGA": "Stop"
}

print(codon_table)
```

```
{'AUG': 'Met', 'UAA': 'Stop', 'UAG': 'Stop', 'UGA': 'Stop'}
```

`dict` Function

You can also create dictionaries using the `dict()` function, which is particularly nice when you have simple string keys:

```
gene = dict(gene="nrdA", product="ribonucleotide reductase")
print(gene)
```

```
{'gene': 'nrdA', 'product': 'ribonucleotide reductase'}
```

`dict` + `zip`

Here's a handy trick: if you have two separate lists that you want to pair up into a dictionary, you can use `zip` with `dict`:

```
genes = ["TP53", "BRCA1", "KRAS"]
functions = ["tumor suppressor", "DNA repair", "signal transduction"]

gene_functions = dict(zip(genes, functions))

print(gene_functions)
```

```
{'TP53': 'tumor suppressor', 'BRCA1': 'DNA repair', 'KRAS': 'signal transduction'}
```

The order matters when using `zip` – the first list provides the keys, and the second list provides the values:

```
# Switching the order gives us a different dictionary
mysterious_dictionary = dict(zip(functions, genes))
print(mysterious_dictionary)
```

```
{'tumor suppressor': 'TP53', 'DNA repair': 'BRCA1', 'signal transduction': 'KRAS'}
```

One Entry at a Time

You can also built up dictionaries one value at a time. Here's a common real-world scenario: you're reading data from a file and need to build a dictionary as you go.

For this example, imagine that `lines` came from parsing a file rather than being hardcoded.

```
# This could be data from a file
lines = [
    ["TP53", "tumor suppressor"],
    ["BRCA1", "DNA repair"],
    ["KRAS", "signal transduction"],
]

# Start with an empty dictionary
gene_functions = {}

# Add each item to the dictionary
for gene_name, function in lines:
    gene_functions[gene_name] = function

print(gene_functions)
```

```
{'TP53': 'tumor suppressor', 'BRCA1': 'DNA repair', 'KRAS': 'signal transduction'}
```

This pattern of building a dictionary piece by piece is something you'll use frequently when working with real data. It's especially useful when processing files or API responses where you don't know the contents ahead of time.

Duplicate Keys & Values

A few important things to know about dictionaries:

- Values can be repeated (the same value can appear multiple times)
- Keys must be unique (if you try to use the same key twice, only the last value will be kept)

Here's an example showing both of these properties:

```
# Values can be repeated
print(dict(a="apple", b="banana", c="apple"))

# Only the last value for a repeated key is kept
codons = {
    "AUG": "Met",
    "UAA": "Stop",
    "UAG": "Stop",
    "UGA": "Stop",
    "AUG": "Methionine",  # This will override the first AUG entry
}
print(codons)
```

```
{'a': 'apple', 'b': 'banana', 'c': 'apple'}
{'AUG': 'Methionine', 'UAA': 'Stop', 'UAG': 'Stop', 'UGA': 'Stop'}
```

Working with Dictionaries: Getting, Adding, and Removing Items

Let's see the basics of working with dictionaries in Python. We'll continue with our `gene_functions` dictionary from earlier:

```
genes = ["TP53", "BRCA1", "KRAS"]
functions = ["tumor suppressor", "DNA repair", "signal transduction"]
gene_functions = dict(zip(genes, functions))
print(gene_functions)
```

```
{'TP53': 'tumor suppressor', 'BRCA1': 'DNA repair', 'KRAS': 'signal transduction'}
```

Getting Items from a Dictionary

The most basic way to look up information in a dictionary is similar to how you'd look up a word in a real dictionary: you use the key to find the value. In Python, this means using square brackets:

```
# Looking up a value
p53_function = gene_functions["TP53"]
print(p53_function)
```

```
tumor suppressor
```

Trying to find a key that doesn't exist will cause an error. (Again, we wrap the code that will cause an error in a `try/except` block so that it doesn't break our notebook code.)

```
try:
    gene_functions["apple pie"]
except KeyError:
    print("there is no gene called 'apple pie'")
```

```
there is no gene called 'apple pie'
```

There is an alternative way to get info from a dictionary that will not raise an error if the key you're searching for is not found: `get`.

```
# This will return `None` rather than raise an error
# if the key is not found
result = gene_functions.get("BRCA2")
print(result)

# This will return the value "Unknown"
# if the key is not found
result = gene_functions.get("BRCA2", "Unknown")
print(result)
```

```
None
Unknown
```

Adding Items to a Dictionary

We mentioned that dictionaries are mutable. Let's see how to add items to our dictionary. You can either add items one at a time or several at once:

```
# Adding a single new entry
gene_functions["EGFR"] = "growth signaling"
print(gene_functions)

# Adding multiple entries at once
gene_functions.update({
    "MDM2": "p53 regulation",
    "BCL2": "apoptosis regulation"
})
print(gene_functions)
```

```
{'TP53': 'tumor suppressor', 'BRCA1': 'DNA repair', 'KRAS': 'signal transduction', 'EGFR': 'g
{'TP53': 'tumor suppressor', 'BRCA1': 'DNA repair', 'KRAS': 'signal transduction', 'EGFR': 'g
```

You can get a bit fancy with updating dictionaries if you want by using operators:

```
letters_and_numbers = dict(a=1, b=2) | dict(a=10, c=30)
print(letters_and_numbers)

letters_and_numbers |= dict(d=400, e=500)
print(letters_and_numbers)
```

```
{'a': 10, 'b': 2, 'c': 30}
{'a': 10, 'b': 2, 'c': 30, 'd': 400, 'e': 500}
```

When you're learning to code, it's best to stick with straightforward, easy-to-read solutions. While Python offers some fancy shortcuts (like complex operators), you'll usually want to write code that you and others can easily understand later. Simple and longer is often better than shorter and clever!

Here's an interesting feature of Python dictionaries that you might have noticed: when you print out a dictionary, the items appear in the exact order you added them. This wasn't always true in older versions of Python, but now dictionaries automatically keep track of the order of your entries.

One final thing to mention. You can't use every Python type as a dictionary key, only immutable types. E.g., you couldn't use a list as a key for a dictionary. The specific reason for that is beyond the scope of this tutorial, but you may be interested in reading more about it here: Why must dictionary keys be immutable?

Removing Items from a Dictionary

Need to remove something from your dictionary? Here are two options:

```
# Remove an entry with del.
#
# del will raise an error if the key is not present
try:
    del gene_functions["KRAS"]
except KeyError:
    print("KRAS was not present in the dictionary")
print(gene_functions)

# Remove and save the value with pop()
#
# We add the "Unknown" to the call to pop so that our program
# will still run if the key is not present.
removed_gene = gene_functions.pop("EGFR", "Unknown")
print(f"Removed function: {removed_gene}")
print(gene_functions)
```

```
{'TP53': 'tumor suppressor', 'BRCA1': 'DNA repair', 'EGFR': 'growth signaling', 'MDM2': 'p53
Removed function: growth signaling
{'TP53': 'tumor suppressor', 'BRCA1': 'DNA repair', 'MDM2': 'p53 regulation', 'BCL2': 'apopt
```

The `del` statement is probably the more common way to remove an item from a dictionary.

Note that if you run that code block more than one time, you will get different outputs. Can you think of why that would be?

By the way…before working with a key, it's often wise to first check if it exists:

```
if "TP53" in gene_functions:
    print("Found TP53's function!")
    function = gene_functions["TP53"]
else:
    print("TP53 not found in our dictionary")
```

```
Found TP53's function!
```

This same technique is a good idea before using `del` as well, since `del` will give you an error if you try to delete the value of a key that is not present in the dictionary.

```
if "TP53" in gene_functions:
    del gene_functions["TP53"]
    print(gene_functions)
else:
    print("TP53 not found in our dictionary")
```

```
{'BRCA1': 'DNA repair', 'MDM2': 'p53 regulation', 'BCL2': 'apoptosis regulation'}
```

Note the use of the `in` operator. It is for membership testing and also works with dictionaries.

Example: Creating the Reverse Complement of a DNA Sequence

Let's tackle a common task in DNA sequence analysis: generating a reverse complement. If you've worked with DNA before, you know that A pairs with T, and C pairs with G.

First, we'll create a dictionary that maps each nucleotide to its complement:

```
complement = {"A": "T", "T": "A", "G": "C", "C": "G"}
print(complement)
```

```
{'A': 'T', 'T': 'A', 'G': 'C', 'C': 'G'}
```

Then, we'll take a simple DNA sequence to demonstrate:

```
dna_sequence = "AACCTTGG"
```

Finally, we'll loop through the sequence backwards (that's what `reversed(...)` does) and look the complement of each nucleotide:

```
for nucleotide in reversed(dna_sequence):
    print(complement[nucleotide], end="")
```

```
CCAAGGTT
```

(The `end=""` parameter tells Python not to add newlines between letters, giving us one continuous sequence.)

Nested Dictionaries: Organizing Complex Data

While simple dictionaries work well for simple mappings like mapping the name of a gene to its function, biological data often has multiple layers of related information.

Let's look at one way we can organize this richer data using nested dictionaries – dictionaries that themselves contain other dictionaries or lists. (Remember how we could nest lists in other lists? This is similar!)

Here's an example showing how we might store information about the TP53 gene:

```
# Gene information database
#
# Imagine there are more genes in here too....
gene_database = {
    "TP53": {
        "full_name": "Tumor Protein P53",
        "chromosome": "17",
        "position": {"start": 7_571_720, "end": 7_590_868},
        "aliases": ["p53", "TRP53"],
    }
}
print(gene_database)
```

```
{'TP53': {'full_name': 'Tumor Protein P53', 'chromosome': '17', 'position': {'start': 757172(
```

Let's use the filing cabinet metaphor again: the main drawer is labeled "TP53", and inside that drawer are several folders containing different types of information. Some of these folders (like "position") contain their own sub-folders! (Alright, it's not the greatest metaphor…but hopefully you get the idea!)

Let's break down what we're storing:

- Basic information: The full name and chromosome location
- Position data: Both start and end coordinates on the chromosome
- Alternative names: A list of other common names for the gene

To access this information, we use square brackets to "drill down" through the layers. Each set of brackets takes us one level deeper:

```
# Get the full name
gene_name = gene_database["TP53"]["full_name"]
print(gene_name)

# Get the start position
start_position = gene_database["TP53"]["position"]["start"]
print(start_position)

# Get the first alias
first_alias = gene_database["TP53"]["aliases"][0]
print(first_alias)
```

```
Tumor Protein P53
7571720
p53
```

It's pretty similar to nested lists, right?

Handling Missing Data in Nested Dictionaries

With nested dictionaries, accessing missing data requires extra care to avoid errors. Let's see why:

```
# Trying to access data that doesn't exist
try:
    # Attempting to access methylation data that isn't stored
    methylation = gene_database["TP53"]["methylation"]["site"]
except KeyError as error:
    print(f"Oops! That data isn't available: {error}")
```

```
Oops! That data isn't available: 'methylation'
```

This code will raise a `KeyError` because we're trying to access a key ("methylation") that doesn't exist. When dealing with nested structures, it's particularly important to handle these cases because an error could occur at any level of nesting.

Here is what happens if we try and access a key that doesn't exist in the `position` map:

```
try:
    middle_position = gene_database["TP53"]["position"]["middle"]
except KeyError as error:
    print(f"Oops! That data isn't available: {error}")
```

```
Oops! That data isn't available: 'middle'
```

As you see, this approach will work for missing keys at different levels of nesting.

One thing to be aware of if you are mixing lists and dictionaries is that while "drilling down" into the data structure you could potentially get errors other than `KeyError`:

```
try:
    an_alias = gene_database["TP53"]["aliases"][10]
except IndexError as error:
    print(f"Oops! That data isn't available: {error}")
```

```
Oops! That data isn't available: list index out of range
```

In this case, we need to handle the `IndexError` because the data that the `aliases` key points to is a list, but that list doesn't have enough items to handle our request for the item at index `10`. Don't worry too much right now on handling specific errors. We will discuss error handling in greater depth in a future tutorial.

While there are quite a few other ways to handle missing data when "drilling down" through nested data structures in Python, for now, we will just use the `try/except` approach similar to the one shown above.

Default Dictionaries: A Nice Way to Handle Missing Keys

We mentioned earlier that you should check for key presence in a dictionary before doing something interesting with that key to avoid key errors. Default dictionaries solve this problem elegantly by automatically creating new entries with preset values when you access a key that doesn't exist yet.

A default dictionary is sort of like a self-initializing storage system. Instead of having to check if a key exists before using it, the dictionary takes care of that for you. It's particularly useful when you're counting occurrences or building categorized lists.

You can create default dictionaries with three common starting values:

- `int`: starts new entries at zero (perfect for counting)

- `list`: starts new entries with an empty list `[]` (great for categorizing or grouping)
- `str`: starts new entries with an empty string `""`

Here is an example showing how to initialize default dictionaries:

```
from collections import defaultdict

# For counting things (starts at 0)
nucleotide_counts = defaultdict(int)

# For grouping things (starts with empty list)
genes_chromosomes = defaultdict(list)
```

Let's look at some practical examples.

Counting Items with `defaultdict`

Say we want to count nucleotides in a DNA sequence. It is pretty straightforward with a default dictionary:

```
nucleotide_counts = defaultdict(int)
dna_sequence = "ATGCATTAG"

for base in dna_sequence:
    nucleotide_counts[base] += 1

for nucleotide, count in nucleotide_counts.items():
    print(f"{nucleotide} => {count}")
```

```
A => 3
T => 3
G => 2
C => 1
```

What's happening here? Each time we see a nucleotide:

- If we haven't seen it before, `defaultdict` automatically creates a counter starting at 0
- We add 1 to the counter

Without `defaultdict`, we'd need this more complicated code:

```
nucleotide_counts = {}
dna_sequence = "ATGCATTAG"

for base in dna_sequence:
    if base in nucleotide_counts:
        nucleotide_counts[base] += 1
    else:
        nucleotide_counts[base] = 1

for nucleotide, count in nucleotide_counts.items():
    print(f"{nucleotide} => {count}")
```

```
A => 3
T => 3
G => 2
C => 1
```

Yuck!

Grouping Items with `defaultdict`

Default dictionaries are also great for grouping related items. Let's organize some genes by their chromosomes:

```
chromosomes = defaultdict(list)

chromosomes["chr17"].append("TP53")
chromosomes["chr13"].append("BRCA2")
chromosomes["chr17"].append("BRCA1")

for chromosome, genes in chromosomes.items():
    for gene in genes:
        print(f"{chromosome}, {gene}")
```

```
chr17, TP53
chr17, BRCA1
chr13, BRCA2
```

Notice how we didn't need to create empty lists for each chromosome first? The `defaultdict` does it for us. Each time we reference a new chromosome, it automatically creates an empty list ready to store genes.

`defaultdict` Summary

The default dictionary approach is particularly useful when you're:

- Counting frequencies of any kind
- Grouping items by categories
- Building collections of related items

Default dictionaries combine the power of regular dictionaries with automatic handling of new keys, making your code both simpler and more robust.

Counters

Python has another type of dictionary called a counter. Counters provide a convenient way to tally hashable items.

Let's return to our example from above, but this time, we will use a `Counter`.

```
from collections import Counter

# This is all you need to tally the nucleotides!
nucleotide_counts = Counter("ATGCATTAG")

# You can loop through the Counter like a dictionary
for nucleotide, count in nucleotide_counts.items():
    print(f"{nucleotide} => {count}")
```

```
A => 3
T => 3
G => 2
C => 1
```

We can find the N most common items using most_common:

```
print(nucleotide_counts.most_common(2))
```

```
[('A', 3), ('T', 3)]
```

Very nice!

What if we wanted to calculate the ratio of nucleotides rather than the raw counts? A counter can help us here too:

```
nucleotide_counts = Counter("ATGCATTAG")

total = nucleotide_counts.total()

for nucleotide, count in nucleotide_counts.items():
    ratio = count / total
    print(f"{nucleotide} => {ratio:.3f}")
```

```
A => 0.333
T => 0.333
G => 0.222
C => 0.111
```

Pretty cool, right?

Counters have lots of other neat methods and operator support that you may want to check out and use in your own programs.

2.7 Control Flow with Collections

Now that we have covered some of Python's data structures and collections, and gone over the different type of loops, let's dive a little deeper into how you can combine collections, loops, and control flow into more realistic programs.

Overview

You have already seen how to loop over collections and sequences. But it never hurts to have a few more examples. Here is the for loop on a couple of different type of sequences:

```
phrase = "Hello, Python!"
for letter in phrase:
    print(letter)

foods = ["apple", "pie", "grape", "cookie"]
for food in foods:
    print(food)

for number in range(2, 10, 2):
    print(number)
```

```python
prices = {"book": 19.99, "pencil": 0.55}

# By default, we only get the keys of a dictionary
# in the for loop
for item in prices:
    print(item)

# Use .items() to get the key and value
for item, price in prices.items():
    print(f"{item} => ${price}")

# Use .values() to get just the values
for price in prices.values():
    print(price)
```

```
H
e
l
l
o
,

P
y
t
h
o
n
!
apple
pie
grape
cookie
2
4
6
8
book
pencil
book => $19.99
pencil => $0.55
19.99
```

```
0.55
```

As we mentioned earlier, you can use the for loop on anything that is iterable.

Recall that if you want to get the position of the item in the sequence over which you are looping, use `enumerate`.

```
phrase = "Hello, Python!"
for index, letter in enumerate(phrase):
    print(f"{index}: {letter}")

foods = ["apple", "pie", "grape", "cookie"]
for index, food in enumerate(foods):
    print(f"{index}: {food}")

for index, number in enumerate(range(2, 10, 2)):
    print(f"{index}: {number}")
```

```
0: H
1: e
2: l
3: l
4: o
5: ,
6:
7: P
8: y
9: t
10: h
11: o
12: n
13: !
0: apple
1: pie
2: grape
3: cookie
0: 2
1: 4
2: 6
3: 8
```

You can use `enumerate` with dictionaries as well, but it is a bit less common, as many times when you are using a dictionary you don't really care about the order anyway.

Controlling the Flow of Loops

When you're working with loops, sometimes you need more than just going through items one by one. You might want to skip certain items, stop the loop early, or take different actions based on what you find. Let's explore some techniques that will give you more control over how your loops behave.

Making Decisions in Loops

We can use boolean expressions and conditional statements to make decisions inside of loops. This allows us to take different actions depending on characteristics of the data.

```
for n in range(10):
    if n > 5:
        print(n)
```

```
6
7
8
9
```

Here, we are looping through the numbers from 0 to 9, and if the number is 6 or more, then we print it, otherwise, we just go on to the next number.

In this example, we want to keep DNA sequences that start with the start codon `ATG`:

```
start_codon = "ATG"
sequences = ['ATGCGC', 'AATTAA', 'GCGCGC', 'TATATA']

with_start_codons = []

for sequence in sequences:
    if sequence.startswith(start_codon):
        with_start_codons.append(sequence)

print(with_start_codons)
```

```
['ATGCGC']
```

This example is actually a decent one for a comprehension:

```
start_codon = "ATG"
sequences = ['ATGCGC', 'AATTAA', 'GCGCGC', 'TATATA']

with_start_codons = [
    sequence for sequence in sequences if sequence.startswith(start_codon)
]

print(with_start_codons)
```

```
['ATGCGC']
```

Comprehensions can be nice for simple filtering and transformations, like in this example. However, you should be cautious about making them too complex. As a rule of thumb:

Good for comprehensions:

- Simple filters (like checking if something starts with "ATG")
- Basic transformations (like converting strings to uppercase)
- When the logic fits naturally on one line

Avoid comprehensions when:

- The logic gets nested or complicated
- Multiple operations are involved
- The line becomes hard to read at a glance

In this case, the comprehension is kind of nice because it's doing a single, straightforward filter operation. But remember: code readability is more important than being clever. If you find yourself writing a complex comprehension, consider using a regular for loop instead.

`break`

Sometimes you find what you're looking for before going through the entire sequence. The `break` statement is like having an "early exit" button – it lets you stop the loop immediately when certain conditions are met. Sometimes this can make your code more efficient by preventing unnecessary iterations.

In this example, we are interested in seeing if a collection of DNA sequences contains at least one sequence with an ambiguous base (`N`), and if so, save that DNA fragment and stop looking:

```python
sequences = ['ATGCGC', 'AATTAGA', 'GCNGCGC', 'TCATATA']

for i, sequence in enumerate(sequences):
    print(f"checking sequence {i+1}")
    # Recall that we can use `in` to check if a
    # letter is in a word.
    if "N" in sequence:
        print(f"sequence {i+1} had an N!\n")
        sequence_with_n = sequence
        break

print(sequence_with_n)
```

```
checking sequence 1
checking sequence 2
checking sequence 3
sequence 3 had an N!

GCNGCGC
```

Notice how the loop stops after the 3rd sequence and doesn't continue all the way until the end. This is thanks to the `break` keyword.

`continue`

Think of `continue` as a "skip to the next item" command. When you hit a `continue` statement, the loop immediately jumps to the next iteration. This is perfect for when you want to skip over certain items without stopping the entire loop, like focusing only on the data points that meet your criteria.

In this example, we only want to process protein fragments that start with Methionine (`M`) and skip the others. While there are multiple ways to approach this, let's use `continue`:

```python
proteins = ["MVQIPQNPL", "ILVDGSSYLYR", "MAYHAFPPLTNSA", "GEPTGA"]

for protein in proteins:
    if not protein.startswith("M"):
        continue

    print(f"we will process {protein}")
```

```
we will process MVQIPQNPL
we will process MAYHAFPPLTNSA
```

This example is a little bit contrived. I actually think writing it *without* the `continue` is clearer:

```python
proteins = ["MVQIPQNPL", "ILVDGSSYLYR", "MAYHAFPPLTNSA", "GEPTGA"]

for protein in proteins:
    if protein.startswith("M"):
        print(f"we will process {protein}")
```

```
we will process MVQIPQNPL
we will process MAYHAFPPLTNSA
```

A Practical Example: Simulating Bacterial Growth

Let's look at something more interesting – simulating how bacteria might grow over time. We'll create a simple model where each bacterium can grow, shrink, or stay the same size each day.

Pay particular attention to this exmaple. It will be useful for Miniproject 1!

```python
import random

total_bacteria = 15

# Make 15 bacteria all starting with size 10
bacteria = [10] * total_bacteria

# Simple "growth" rules:
#
# - 50% chance to grow
# - 25% chance to shrink
# - 25% chance to stay the same

# The outer loop tracks days in the experiment
for day in range(20):

    # The inner loop tracks each individual bateria
    for i in range(total_bacteria):
```

```
        chance = random.random()

        # First we check if this bacterium will grow today
        if chance < 0.5:
            bacteria[i] += 1
        # If it will not grow, we need to check if it will shrink
        elif chance < 0.75:
            bacteria[i] -= 1

        # We don't need the `else` here because if the bacterium
        # won't grow AND it won't shrink, then no action is required.

# Finally, we print out the sizes of all the bacteria
# at the end of the experiment
for id, size in enumerate(bacteria):
    print(f"bacterium {id+1}, size: {size}")
```

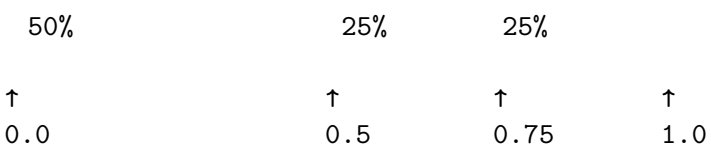
```
bacterium 1, size: 15
bacterium 2, size: 17
bacterium 3, size: 22
bacterium 4, size: 12
bacterium 5, size: 10
bacterium 6, size: 12
bacterium 7, size: 12
bacterium 8, size: 18
bacterium 9, size: 13
bacterium 10, size: 16
bacterium 11, size: 14
bacterium 12, size: 18
bacterium 13, size: 14
bacterium 14, size: 18
bacterium 15, size: 12
```

Here is what is happening:

- In the outer loop, we run the simulation for 20 days, with each iteration representing one day of bacterial growth.
- In the inner loop, we check each bacterium in our population and apply the growth rules using random chances.
- Then we loop through the bacteria sizes and print out the final size of each bacterium. (We treat the bacterium's location in the array (plus one) as its ID.)

How the Random Choices Work

The clever part here is how we use a single random number to make weighted choices. Think of it like a number line from 0 to 1, divided into three sections:



When we generate a random number between 0 and 1:

- If it falls in the first half (0.0-0.5), the bacterium grows
- If it falls in the next quarter (0.5-0.75), the bacterium shrinks
- If it falls in the last quarter (0.75-1.0), the bacterium stays the same size

This is one way to implement different probabilities for different outcomes. While this example uses bacterial growth, you could adapt this pattern for any situation where you need to simulate random events with different probabilities – like mutation rates, drug responses, or population changes.

If you are curious, Python has a method that simplifies this random choice logic. Check it out if you're curious! You might want to use it for your first miniproject….

2.8 Common Sequence Operations

You may have noticed that we can treat many of Python's collection types in a similar way.

One of Python's most helpful features is that many collection types (like lists, strings, and tuples) share the same basic operations. This means once you learn how to work with one type of sequence, you can apply that knowledge to others – you can find the length of any sequence using `len()`, check if something exists in a sequence using `in`, or grab a specific element using square bracket notation `[]`.

For instance, whether you're working with a DNA sequence as a string or a list of gene names, you can use the same syntax: `len("ATCG")` and `len(["nrdA", "nrdJ"])` both work the same way!

2.9 Choosing the Right Collection

When deciding which type of collection to use, consider these three key questions:

1. "How will I create or receive this data initially?"
2. "How will I need to access this data later?"
3. "How will I need to modify this data?"

Here's a practical guide to help you choose:

Use a list when...

- Your data has a meaningful order (e.g., lines from a file, time series)
- You need to access items by position (index) or slices
- You need ordered operations (iteration in sequence, sorting, reversing)
- You want efficient operations at the end of the collection (append/pop)
- You need to maintain duplicates
- You need to modify items in place

Use a dictionary when...

- Your data naturally comes as key-value pairs
- You need to look up values by a unique identifier (key)
- You need to efficiently find, add, or update specific items without linear searching
- You want to map one piece of data to another
- You need to combine data from multiple sources using a common key

Use a set when...

- You only care about uniqueness, not order or association
- You need automatic elimination of duplicates
- You're only concerned with presence/absence of items
- You need to perform set operations (unions, intersections, differences)
- You need fast membership testing

Examples

For instance, when processing a FASTA file, you'll encounter ID-sequence pairs. If you need to access sequences by their identifiers later, a dictionary is the natural choice. However, if you're only interested in the sequences themselves and won't need to reference them by ID, storing just the sequences in a list would be more appropriate.

As another example, consider analyzing homology search results where you need to organize multiple hits that correspond to each query sequence. If you'll need to retrieve all hits for a specific query using its identifier, a dictionary is ideal. You could structure it with query IDs as keys and lists of corresponding hits as values, allowing efficient lookup of results for any particular query of interest:

```
# Tuples of query-target-bitscore -- imagine these come directly from a BLAST
# output file or something similar.
homology_search_results = [
    ("query_1", "target_1", 95),
    ("query_1", "target_2", 32),
    ("query_2", "target_1", 112)
]

query_hits = {}

for query, target, bitscore in homology_search_results:
    hit_info = (target, bitscore)

    if query in query_hits:
        query_hits[query].append(hit_info)
    else:
        query_hits[query] = [hit_info]

print(query_hits["query_2"])
```

```
[('target_1', 112)]
```

Summary

To summarize, select the collection type that both enhances code readability and aligns with your specific patterns of data creation, access, and modification throughout your program's workflow.

2.10 Key Takeaways

We've covered a lot of material about some of Python's most commonly used data structures. Here are some key takeaways.

General Suggestions

- Generally keep data types consistent within collections
- Use clear, descriptive names
- Choose the simplest structure that works
- Use list comprehensions for simple transformations
- Handle missing dictionary keys with `get`
- Consider memory usage with large datasets

Watch Out For

- Modifying lists while iterating
- Forgetting tuple immutability
- Missing dictionary keys
- Infinite loops
- Using lists when dictionaries would be more appropriate

3 Algorithmic Thinking

3.1 Overview

An algorithm is a step-by-step procedure for solving a problem or accomplishing a task. It as a detailed set of instructions that takes some input, follows a clear sequence of steps, and produces a desired output.

Algorithms can vary greatly in their level of complexity, from simple operations like finding the larger of two numbers to complex tasks such as generating a phylogenetic tree from a sequence alignment. It's worth noting that the same problem might have multiple algorithmic solutions, each with its own advantages and trade-offs in terms of simplicity and efficiency.

Key Characteristics of Algorithms

(Adapted from The Art of Computer Programming by Donald E. Knuth)

All algorithms share some important properties:

- Defined inputs & outputs
  - Algorithms must have clearly defined inputs and outputs.
  - Example: PCR protocol
    - Input: Template DNA, primers, nucleotides, polymerase
    - Output: Amplified DNA fragments
- Definiteness
  - Steps must be clear and unambiguous
  - Each step must be understood exactly the same way by anyone following it
  - Examples:
    - Good example: "Heat the sample at 95°C for 5 minutes"
    - Bad example: "Heat the sample for a while"
- Finiteness
  - The algorithm must terminate after a finite number of steps
  - I.e., it cannot run indefinitely
  - Examples:

        * Good example: A PCR reaction has a specific number of cycles
        * Bad example: "Keep checking gel until bands appear" (no clear end point)

- Effectiveness
    - Each step must be basic enough to be executed
    - Must be doable with available resources (ideally by a person using a pen and paper, but not always practical)
    - Examples:
        * Good example: "Pipette 100 µL"
        * Bad example: "Separate molecules instantly"

There are a few more important properties of algorithms. Generally an algorithm should produce the same output given the same input. For example, if your algorithm is supposed to triple a number, an input of `5` should always produce an output of `15`. Additionally, an algorithm should ideally be general enough to solve similar problems in a category. Your tripling algorithm would be more useful (and general) if a user could supply both the number to be multiplied (multiplicand) and the multiplier. This way, the same algorithm could be used for doubling, tripling, quadrupling, etc.

3.2 Algorithmic Thinking Process

Being able to think algorithmically is essential for success in programming. Algorithmic thinking is the ability to break down problems into clear, logical steps that a computer can follow – like writing a very detailed recipe where nothing can be assumed or left to interpretation. This skill helps you break down complex problems and translate them into effective code solutions.

Let's go through the various aspects of algorithms.

Basic Components

Every algorithm consists of three fundamental parts:

- Input
    - The data or information that your algorithm needs to work with.
    - This could be numbers, text, DNA sequences, or any other form of data.
- Processing
    - The step-by-step instructions that transform your input into the desired result.
    - This is essentially your recipe or procedure.

- Output
  - The final result or solution that your algorithm produces.
  - This should match what you need to solve your problem.

Before you can write an algorithm, you need to understand what problem you're trying to solve:

- Defining the problem scope
  - Clearly state what your algorithm should and shouldn't do.
  - For example, "Find all prime numbers under 100" is clearer than "Find prime numbers."
- Understanding requirements
  - List everything your solution needs to handle.
  - What kinds of input should it work with?
  - What should it do with invalid input?

You can think of these as "behind-the-scenes" components. They are critical to algorithmic thinking and construction, but not always explicitly part of the algorithm itself.

Breaking Problems Into Steps

Once you have these components in mind, you can break large problems into smaller pieces, which are much more manageable:

- One big problem -> Mutliple sub-problems
  - Break your main problem into smaller, manageable tasks.
  - Instead of "Analyze DNA sequence," think about
    - Read sequence
    - Check validity
    - Find patterns
  - These steps can get as granular as necessary for you to solve the problem at hand.
- Determine the essential operations needed for each sub-problem.
- Create a logical sequence of operations
  - Arrange your sub-problems in a logical order.
  - What needs to happen first?
  - Which steps depend on other steps?

Pseudocode Development

Depending on the complexity of your problem, it can be helpful to sketch out your solution in plain language or pseudocode:

- Writing abstract steps
  - Write out your algorithm in everyday language.
  - Use simple statements like "For each number in the list" or "If sequence is valid then…"
- Planning program flow
  - Map out how your program will move from start to finish.
  - What decisions need to be made?
  - What steps might need to repeat?
- Outlining solution structure
  - Organize your steps into a clear structure, showing where loops and decisions occur.

Planning your code's structure and components before diving into actual programming makes the whole process much smoother. When you tackle problems this way, you can focus on one aspect at a time, first mapping out the logic and flow, then implementing the code itself. This approach prevents you from getting overwhelmed by trying to solve multiple challenges simultaneously.

While thorough planning is essential when you're learning, you'll likely develop a more streamlined approach as you gain experience. For simpler problems, you may find yourself able to start coding directly, having internalized the planning process. However, for complex projects, taking time to sketch out your approach first remains valuable regardless of your skill level.

Implementation

Now that you have a solid plan, it's time to translate it into working code.

- Convert each step from your pseudocode into actual Python code, one piece at a time
- Build your code following the structure you mapped out earlier
- Keep your code clean and maintainable by:
  - Using descriptive variable names that make sense
  - Adding helpful comments to explain what your code does
  - Following consistent formatting and organization

This stage is a bit like assembling the pieces of a puzzle where you already know what the final picture should look like. Take your time with each component – rushing through implementation often leads to mistakes that can be challenging to fix later.

Testing and Validation

After your implementation is complete, be sure to test that your algorithm works correctly:

- Testing with simple examples
  - Start with basic test cases where you know the correct answer.
  - Check that your algorithm produces correct results for all expected inputs.
- Iterative refinement
  - Improve your solution based on test results.
  - Fix errors and handle edge cases.

We will discuss more testing strategies in a later tutorial.

3.3 Real-World Algorithm Example – Making Coffee

Let's take an everyday activity, making coffee, and practice turning it into clear instructions that could work as an algorithm. We will start with some pretty bad instructions, identify problems with them, and then refine them.

(Apologies to all the tea lovers reading this!)

Take 1

Here is a silly set of instructions for making coffee:

> You'll want to put some liquid in there first, then put the paper thing in. Get the coffee ready – not too much, not too little. Make sure everything is closed tight before you get things going. Now, just give it a tap and wait a while. If all goes well, you should end up with something drinkable!

If you have ever made coffee before, you could probably figure out how to follow these instructions. However, it doesn't really work as an algorithm:

- "liquid": should I add water or milk?
- "in there": in where?
- "paper thing" instead of filter
- "coffee": should I add coffee beans, ground coffee, instant coffee?
- "give it a tap" instead of pressing the start button
- "not too much, not too little": no specific measurements or timing
- Uses subjective phrases like "nice and tight" and "you know how it goes"
- Uncertain outcome ("if all goes well")

Let's address some of these issues and try again.

Take 2

> Pour fresh water into the top part until it looks full enough. Insert a clean paper filter (any size should work) into the basket area. Measure some coffee grounds – about a few spoonfuls should do it, depending on how strong you like it. Make sure to close the top properly until you hear a click or something. Find the power button (it might be on the front or side) and push it down. After a few minutes when you stop hearing the machine make noise, your coffee should be done!

Though this version is definitely better than the last one, it still has a few issues:

- Imprecise measurements ("looks full enough", "a few spoonfuls")
- Ambiguous specifications ("any size should work")
- Subjective criteria ("how strong you like it")
- Uncertain timing ("a few minutes")
- Vague sensory cues ("hear a click or something", "stop hearing the machine")
- Optional or unclear elements ("or something", "might be on the front")

Again, if you have ever used a coffee machine, you could probably understand the instructions and adapt them to your taste to make a good cup of coffee. But to be a good algorithm, it still needs more precision, less ambiguity, and it shouldn't leave so much up to your own taste.

Let's address some more of those ambiguities.

Take 3

> Pour 8 cups of fresh water into the top reservoir, filling to the marked water line. Insert a #4 cone paper filter into the filter basket. Measure 2 level tablespoons of ground coffee per cup of water (16 tablespoons total for a full pot). Press down firmly on the lid until you hear a distinct click indicating it's fully closed. Locate the power switch on the front panel and press it to the "ON" position. The brewing process will take approximately 5-7 minutes, and is complete when the gurgling sound stops and the brewing indicator light turns off. Your coffee is now ready to serve.

Here are some specific improvements as compared to the last version:

- Using specific measurements (8 cups, #4 filter, 2 tablespoons per cup)
- Providing clear indicators (marked water line, distinct click)
- Giving a defined time range (5-7 minutes)
- Including concrete completion signals (gurgling stops, indicator light)
- Specifying exact locations (front panel)

Beyond the Basic Steps

Though we could keep refining these instructions, it's not a bad description of making coffee now!

If this were a "real algorithm" that we needed to program in Python, there are some more things we should think about. When writing instructions for a computer (or a coffee maker!), it's easy to focus on the happy path – the sequence of actions that work perfectly. However, robust algorithms must consider various other factors to effectively handle real-world situations where things can go wrong.

Here are some things to think about that are "beyond the basic steps".

Setup and Requirements

Before starting any process, we need to ensure everything is in place. This includes checking equipment, materials, and the system's readiness. (You might see the term "preconditions" if you are reading about algorithms online.)

- General Questions:
  - Have we verified all equipment is functional?
  - Are all necessary materials available?
  - Is the system in a ready state?
- Coffee Maker Example:
  - Is the coffee maker plugged in and working?
  - Is it clean/ready to use?
  - Do you have all needed supplies on hand?

Handling Problems

Things can go wrong. A good algorithm anticipates potential problems and provides solutions.

- General Questions:
  - How do we handle insufficient resources?
  - What happens if components fail?
  - How do we respond to interruptions?
  - What backup procedures are needed?
- Coffee Maker Example:
  - What if there's not enough water?

  - What if the filter is inserted incorrectly?
  - What if the machine doesn't turn on?
  - What if the brewing stops midway?

While you can't generally anticipate everything that may go wrong, it's a good idea to put some thought into it, and try to handle any likely errors.

Sequential Dependencies

Generally, certain steps will rely on others. We need to define the correct order of operations.

- General Questions:
  - Which steps must happen in a specific order?
  - What are the critical timing requirements?
  - Which steps block others from proceeding?
- Coffee Maker Example:
  - The filter must be inserted before adding coffee grounds
  - Water must be added before turning on the machine
  - The lid must be closed before powering on

Conditional Pathways

Algorithms often need to handle different scenarios based on input or conditions.

- General Questions:
  - How do varying inputs affect the process?
  - What alternative routes exist?
  - How do we handle different scenarios?
- Coffee Maker Example:
  - If making less than a full pot, adjust measurements accordingly
  - If using different grind sizes, adjust portions

Validation

Validation and verification of any post-conditions is essential to ensure each step is completed successfully and the final result is correct.

- General Questions:

  - How do we verify each step succeeded?
  - What indicates proper operation?
  - How do we confirm the final result?

- Coffee Maker Example:

  - How to check if the filter is seated properly
  - How to verify the water is actually flowing/brewing
  - How to confirm the coffee is properly brewed

Summary

While we don't need this level of detail for every example, it's valuable to understand how simple procedures evolve into robust algorithms through careful consideration of edge cases, error handling, and validation steps.

This methodology applies broadly to software development: start simple, then systematically address complexities and potential problems.

3.4 Building Blocks for Solving Programming Problems

When you're learning to program, it helps to recognize that many solutions are built from common, reusable patterns. These patterns are basic building blocks that you can combine and adapt to solve more complex problems.

While there are often many ways to solve a programming challenge, we'll focus on straightforward approaches that are easy to understand and implement. These might not always be the most efficient solutions, but they're good learning tools that will help you:

- Break big problems into manageable pieces
- Learn reliable approaches to common challenges
- Develop your problem-solving skills

As you gain experience, you'll learn more sophisticated methods, or ways that are built-in to the language itself, but mastering these fundamental patterns first will give you a solid foundation. Let's look at some practical examples that demonstrate these basic patterns in action.

In this tutorial, we will mostly focus on strategies that involve looking at one element at a time from a sequence. Often, this sequential processing will also involve tracking or accumulating values.

Character Processing

Tutorial 2 showed many examples of using for loops to process characters of a string one-by-one. We will repeat some of them here so that you can get more practice seeing the common patterns.

Printing Each Character

Printing each letter of a string:

```
word = "HELLO"
for character in word:
    # Do something interesting with each character...
    print(character)
```

```
H
E
L
L
O
```

Iterating with an Index

Accessing the index of the character during iteration:

```
# Print with position
word = "hello"
for index, letter in enumerate(word):
    print(f"Position {index}: {letter}")
```

```
Position 0: h
Position 1: e
Position 2: l
Position 3: l
Position 4: o
```

Iterating in Reverse

Processing a string in reverse order:

```
# Process in reverse
for letter in reversed(word):
    print(letter)
```

```
o
l
l
e
h
```

Frequencies

Counting the frequency of individual letters:

```
letter_counts = {}

for letter in word:
    current_count = letter_counts.get(letter, 0)
    letter_counts[letter] = current_count + 1

print(letter_counts)
```

```
{'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

Note: This is a good example of what I mentioned above regarding these solutions not always being the best way to do something. In Tutorial 2, we discussed a better way to approach this particular counting problem. Can you remember it?

Number Processing

Running Sum

Tracking a running sum:

```python
numbers = [2, 5, 3, 1]
total = 0

for number in numbers:
    total += number

print(f"Total: {total}")
```

```
Total: 11
```

While Python provides built-in functions like `sum()` for this specific case, understanding the basic pattern helps with more complex variations.

Summing Positive Numbers

Sum of positive numbers:

```python
numbers = [-1, 2, -5, 3, -8, 1]

positive_sum = sum(num for num in numbers if num > 0)

print(f"Sum of positive numbers: {positive_sum}")
```

```
Sum of positive numbers: 6
```

Averages

Calculating the average of a list of numbers:

```python
# Calculate average
numbers = [2, 5, 3, 8, 1]
average = sum(numbers) / len(numbers)
print(f"Average: {average}")
```

```
Average: 3.8
```

Finding Maximum/Minimum

Finding the largest number in a list without using Python's built-in `max` function:

```
numbers = [5, 3, 0, -1, 8]
largest_number = numbers[0]

for number in numbers[1:]:
    if number > largest_number:
        largest_number = number

print(largest_number)
```

```
8
```

Finding the shortest string in a list:

```
words = ["i", "like", "apple", "pie"]
shortest_word = words[0]

for word in words[1:]:
    if len(word) < len(shortest_word):
        shortest_word = word

print(shortest_word)
```

```
i
```

Simple Search/Validation

Another common task is finding an item in a collection or validating some condition of a collection.

Finding a Number in a List

Finding a specific number in a list:

```
target = 5
numbers = list(range(10))
is_found = False

for number in numbers:
    if number == target:
        is_found = True
        print("we found the number!")

if not is_found:
    print("we didn't find the number!")
```

```
we found the number!
```

Is a List Sorted?

Checking if a list is sorted. (Before reading the code, try and think of how the solution might look.)

```
numbers = [1, 2, 4, 3, 5]
previous_number = numbers[0]

is_sorted = True

for current_number in numbers[1:]:
    if current_number < previous_number:
        is_sorted = False
        break
    previous_number = current_number

if is_sorted:
    print("the list is sorted!")
else:
    print("the list is not sorted!")
```

```
the list is not sorted!
```

This example has a couple of interesting things to focus on:

- We start the iteration at index 1 in the array
- As soon as we see a number that is not sorted, we `break` since that is enough to say the array as a whole is unsorted.

Nested Loops

Problems often require nested loops, such as cases where for every item in one list, you need to process every item in another list. Note that these nested loop problems can often be solved in clever ways that help you avoid a having to look at every combination. There's a good chance you will see some of these clever solutions as you are exposed to more code in the future.

Distance Between Points

Calculating distances between points. Here we are using 1-dimensional points. The distance between two points in 1D (on a line) is the absolute value of their difference. E.g., if you have two points `x` and `x`, the distance between them is `|x - x|`.

```
points = [8, 3, 4]
distances = []

for x in points:
    for y in points:
        distance = abs(x - y)
        distances.append((x, y, distance))

for x, y, distance in distances:
    print(f"({x}, {y}) => {distance}")
```

```
(8, 8) => 0
(8, 3) => 5
(8, 4) => 4
(3, 8) => 5
(3, 3) => 0
(3, 4) => 1
(4, 8) => 4
(4, 3) => 1
(4, 4) => 0
```

You could imagine instead of distances of 1D points, this pattern could work for calculating all-vs-all homology scores from BLAST output, or comparing some aspect of each sample vs. every other sample.

Distance Between Samples

Here's a slightly different example. In this case, say we have ecological distances between all sampling locations stored in a dictionary. Here is one way that you might loop through them:

```
sample_distances = {
    "S1": {"S1": 0, "S2": 3, "S3": 5},
    "S2": {"S1": 2, "S2": 0, "S3": 1},
    "S3": {"S1": 6, "S2": 2, "S3": 0},
}

for sample_a, other_samples in sample_distances.items():
    for sample_b, distance in other_samples.items():
        print(f"{sample_a} -> {sample_b} => {distance}")
```

```
S1 -> S1 => 0
S1 -> S2 => 3
S1 -> S3 => 5
S2 -> S1 => 2
S2 -> S2 => 0
S2 -> S3 => 1
S3 -> S1 => 6
S3 -> S2 => 2
S3 -> S3 => 0
```

While there are often clever ways to avoid these type of all-vs-all comparisons, they still come up pretty frequently, so it's a good idea to get familiar with them!

3.5 Introduction to Algorithm Analysis

When we write code to solve a problem, there's usually more than one way to do it. It's a bit like how you may have different routes to get to work – some are faster, some are more efficient, and some are just easier to remember. The same applies to our code solutions.

When evaluating different approaches to solving a problem, we typically look at three main things:

1. Does it actually solve the problem correctly?
2. How efficiently does it run (in terms of time and computer memory)?
3. Is it clear and maintainable?

Time Complexity

Let's focus on efficiency in terms of time for a moment. Imagine you have a list of genes to search through. You could check each gene one by one (we call this linear time), or you might have a clever way to eliminate half the possibilities with each step (logarithmic time). As your gene list grows from hundreds to millions of entries, these different approaches can mean the difference between waiting seconds versus hours for your results.

Computer scientists use something called "Big O notation" to describe how an algorithm's performance changes as the input gets larger. Here are some common patterns you'll encounter.

- Constant time (`O(1)`): The operation always takes the same amount of time
- Linear time (`O(n)`): The time increases directly with the size of the input
- Quadratic time ($O(n^2)$): The time increases with the square of the input size

The key takeaway is that some solutions scale better than others when working with larger datasets. As you write code, keeping these basic patterns in mind will help you make better choices about how to approach problems.

Here are some simple examples to illustrate these three time complexities.

Constant Time – O(1)

Constant time operations like dictionary lookups:

```
gene_info = {"nrdA": "ribonucleotide reductase"}
result = gene_info["nrdA"]
print(result)
```

```
ribonucleotide reductase
```

Linear Time – O(n)

Linear time operations like checking each item in a list once:

```
# Counting mutations
dna_sequence = "ACTACTGTACTACTGTCACACTAGAGTAT"
t_count = 0

for base in dna_sequence:
    if base == "T":
        t_count += 1
```

```
print(t_count)
```

```
9
```

Quadratic Time – O(n²)

Quadratic time operations like comparing every item with every other item:

```
# Finding equivalent sequences
sequences = ["ACTG", "ATGAC", "ACTGGT", "ACTG"]
sequence_count = len(sequences)

for i in range(sequence_count):
    for j in range(sequence_count):
        if i != j and sequences[i] == sequences[j]:
            print(f"Match found: {sequences[i]}")
```

```
Match found: ACTG
Match found: ACTG
```

Space Complexity

In addition to thinking about how long our code takes to run, sometimes we also need to consider how much memory it uses. Space complexity describes how memory usage grows with input size. The two most common patterns you'll encounter are:

- O(1) space: Uses a fixed amount of extra memory regardless of input size
- O(n) space: Uses extra memory that grows with the input size

Here are some examples.

Constant Space – O(1)

In a constant space solution, the same few variables are used regardless of the input size:

```
g_count = 0

for base in dna_sequence:
    if base == "G":
        g_count += 1

print(g_count)
```

```
4
```

In this example, we're just counting, so we only need one variable no matter how long the DNA sequence is.

Linear Space – O(n)

In a linear space solution, the space needed to calculate the result grows linearly with the size of the input.

```
g_positions = []

for i in range(len(dna_sequence)):
    if dna_sequence[i] == "G":
        g_positions.append(i)

print(g_positions)
```

```
[6, 14, 23, 25]
```

In this example, we're storing positions, so we need more space for longer sequences.

3.6 Algorithmic Puzzles

Let's finish off this tutorial by looking at a common, beginner-level algorithmic puzzle: checking if a string is a palindrome.

> A word is a **palindrome** if it reads the same forward and backward.

Note: This problem description is adapted from LeetCode problem 125. Valid Palindrome.

Starting with the Problem

First, let's understand what we're trying to do in plain English:

- We need to check if a word reads the same forwards and backwards
- Examples
  - "racecar" → Yes!
  - "hello" → No!

(For this version of the problem, we can assume that the strings we need to check are all a single word with all lowercase letters.)

Solution 1: The Obvious Way

Let's start with the most obvious solution that uses Python string slicing to check the definition of a palindrome.

```
string = "racecar"
is_palindrome = string == string[::-1]
print(string, is_palindrome)

string = "apple"
is_palindrome = string == string[::-1]
print(string, is_palindrome)
```

```
racecar True
apple False
```

This is probably how most people would first think about it: "Just reverse it and compare!"

- It's perfectly valid
- It's easy to understand
- It uses built-in Python functions
- *But*...It requires creating a whole new reversed string in memory

Often, the "but" doesn't really matter in whatever work you are doing. Many times "clear and maintainable" are much more important that optimal efficiency. However, in the world of algorithmic puzzles, the "but" is definitely something you want to address!

Solution 2: Manual Comparison with a Loop

For this solution, we think, "Wait, do we really need to reverse the string? What a waste of time and space!"

Instead, we will look at "pairs" of letters. First, let's try a word that is a palindrome:

```
string = "racecar"
print("Checking {string}")
is_palindrome = True

for i in range(len(string)):
    j = len(string) - i - 1

    print(i, j, string[i], string[j])

    if string[i] != string[j]:
        is_palindrome = False
        break

print(string, is_palindrome)
```

```
Checking {string}
0 6 r r
1 5 a a
2 4 c c
3 3 e e
4 2 c c
5 1 a a
6 0 r r
racecar True
```

Next, try a word that is not a palindrome, just to see the difference:

```
string = "racethecar"
print("\n\nChecking {string}")
is_palindrome = True

for i in range(len(string)):
    j = len(string) - i - 1

    print(i, j, string[i], string[j])
```

```
    if string[i] != string[j]:
        is_palindrome = False
        break

print(string, is_palindrome)
```

```
Checking {string}
0 9 r r
1 8 a a
2 7 c c
3 6 e e
4 5 t h
racethecar False
```

Do you see how the non-palindrome stops before checking all the values?

Here is a breakdown of the above solution:

- We compare the first letter with last letter
- Then, the second letter with second-to-last letter
- And so on…
- *But*…We're doing each comparison twice!

Just so it is clear, let's explain that `j` index line. This diagram shows why we use that formula to calculate `j`:

```
len("racecar") == 7
```

```
 r   a   c   e   c   a   r

 0   1   2   3   4   5   6
 ↑   ↑   ↑   ↑   ↑   ↑   ↑
 i=0                  j = 7 - 0 - 1 = 6

     i=1            j = 7 - 1 - 1 = 5

         i=2     j = 7 - 2 - 1 = 4

             i=3, j = 7 - 3 - 1 = 3
```

```
and so on...
```

Solution 3: The Optimized Version

There was another "but" in Solution 2: Wouldn't it be better if we didn't have to do those duplicated checks? Let's give it a shot.

```
string = "racecar"
is_palindrome = True

i = 0
j = len(string) - 1

while i < j:
    print(i, j, string[i], string[j])

    if string[i] != string[j]:
        is_palindrome = False
        break

    i += 1
    j -= 1

print(string, is_palindrome)
```

```
0 6 r r
1 5 a a
2 4 c c
racecar True
```

And here again with a string that is not a palindrome:

```
string = "racethecar"
is_palindrome = True

i = 0
j = len(string) - 1

while i < j:
    print(i, j, string[i], string[j])
```

```
    if string[i] != string[j]:
        is_palindrome = False
        break

    i += 1
    j -= 1

print(string, is_palindrome)
```

```
0 9 r r
1 8 a a
2 7 c c
3 6 e e
4 5 t h
racethecar False
```

And the breakdown:

- We use two "pointers" moving toward each other
- We only check each pair once
- We stop at the middle

This is both time and space efficient, and doesn't do any more checks than we need to do. Cool, right?

General Approach

The above process of refinement suggests a general approach to these kinds of algorithmic puzzles.

- Start simple
  - Implement the first solution that comes to mind
  - Don't worry if it's not perfect
  - Make sure it works!
- Question your approach
  - Do I need all these steps?
  - Am I repeating work?
  - Is there a more direct way?
- Look for patterns

  - Notice we're comparing pairs of letters
  - Notice we're moving inward from both ends
  - These observations lead to better solutions

- Consider resource usage
  - Time: How many steps are we taking?
  - Space: How much extra memory do we need?
  - Can we reduce either?

Focus on making a basic version work before aiming for perfection. Begin with a simple solution, ensure it functions correctly, and then gradually improve it. This mirrors the process of algorithmic thinking described above of breaking down complex problems into manageable steps and refining your solution as needs evolve. This is similar to how programming works in real-world scenarios!

3.7 Summary & Connection to Bioinformatics

In this tutorial, we introduced the concept of algorithmic thinking and simple algorithms. We went over some common patterns for simple problems that can form the building blocks of more complex solutions. Then we covered the basics of algorithmic complexity analysis, and finally, went through the process of solving a common algorithmic puzzle.

You might be wondering how these basic programming concepts connect to the bioinformatics tools you'll use in your research. While we rely on sophisticated software for tasks like homology search and genome assembly, these powerful tools are built on the same fundamental programming principles we're learning now.

Algorithmic thinking, or understanding how to break down problems and translate them into logical steps, is a foundational skill for all coding work. Though we won't be building complex bioinformatics tools from scratch in this course, mastering the basics will give you a solid foundation for writing your own analysis scripts and understanding how the bioinformatics tools you use actually work.

3.8 Bibliography

Knuth, Donald E. 1997. The Art of Computer Programming: Volume 1: Fundamental Algorithms. 3rd ed. Boston, MA: Addison Wesley.

4 Functions

4.1 Introduction

Until now, we've mostly been using built-in Python's functions like `print()`, `len()`, and `range()` to get things done. While Python's standard library provides a ton of useful functionality, eventually you will need to create something more tailored to your specific needs.

This is where custom functions come in. Functions are like reusable blocks of code that you design to perform specific tasks. Creating your own functions allows you to extend Python's capabilities beyond what's available "out of the box." In this module, we'll explore functions from the ground up: their purpose, how to create them, and how to write them effectively.

Learning to work with functions is a core programming skill that offers several key benefits:

- It helps you tackle large problems by breaking them into smaller, manageable parts
- It makes your code more organized and easier to follow
- It makes your code easier for others to understand and use
- It eliminates the need to copy and paste code repeatedly
- It reduces mistakes by creating tested, reliable code blocks you can reuse

Whether you're analyzing data sets, automating repetitive tasks, or building scientific applications, functions will become an essential part of your programming toolkit. Let's dive in and learn how to create them!

What is a Function?

A function is a reusable block of code that performs a specific task, a sort of standard procedure for performing a particular computation or task in your code.

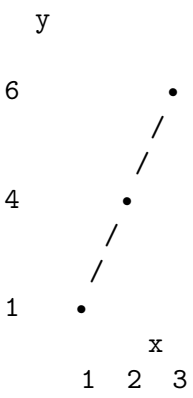
Functions help us organize our code by breaking down complex problems into smaller, more manageable pieces. For example, instead of writing one massive program that does everything, we can create separate functions for each logical step. This makes our code easier to write, understand, and fix when something goes wrong.

One important feature of functions is that we can give them descriptive names that explain what they do. When you see a function named `calculate_average` or `find_peak_values`,

you immediately have a good idea of its purpose. This makes your code more readable and easier to maintain.

You might hear functions called by different names like methods, procedures, or subroutines, depending on the programming language or context. Don't worry too much about these different terms; for now, we will simply call them functions. Regardless of their name, their main purpose remains the same: to package up a set of instructions that we can easily reuse.

Let's start with something familiar - mathematical functions. You might remember seeing expressions like $f(x) = 2x$ in math class. This simple function takes any number x and doubles it. Graphically, it creates a straight line where each input x corresponds to an output y that's twice as large:



We can create a function with similar behavior in Python:

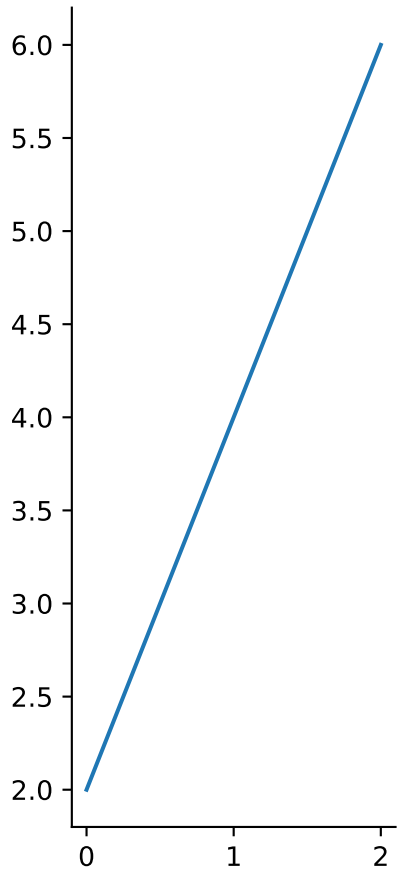
```
def times_2(x):
    return 2 * x
```

Here we see a function called `times_2` that takes a value `x` and returns a value that is twice as large (`2 * x`). Let's use the seaborn package to visualize some inputs and outputs of the `times_2` function:

```
# Import the seaborn package
import seaborn as sns


# Create some x values
xs = [1, 2, 3]

# The y values are results of running
# the function on all the x values
```

```
ys = [times_2(x) for x in xs]

# Print out the y values
print(ys)

# Finally, draw a plot of the results.
sns.relplot(ys, kind="line", aspect=0.5)
```

```
[2, 4, 6]
```



At its core, a function is like a machine that follows a specific set of instructions:

```
input              output

a function
```

You put something in (input), the function processes it according to its instructions, and you get something out (output).

As we will see, Python functions are more flexible than mathematical ones. They can:

- Take multiple inputs (or none at all)
- Return collections of values (or nothing at all)
- Perform actions beyond just calculations (like printing text or saving files)

Why Do We Need Functions?

You can think of Python functions as pre-packaged units of code that perform specific tasks. When you write a function, you're creating a reusable tool that you can call whenever you need it. Functions create a logical boundary around related operations, giving them a clear purpose and identity, which can make code more readable, reusable, and maintainable. Here's a simple example that shows how functions make code more readable:

```
gene_expression_data = read_csv("gene_expression.csv")

upregulated_genes = find_upregulated_genes(gene_expression_data)

plot_expression_data(upregulated_genes)
```

Even if you don't know exactly how these functions work internally, you can probably guess what this code does just by reading the function names. When you combine descriptive function names with clear variable names, your code becomes self-documenting, that is, it will be easier to understand without additional comments or explanations.

Here are some of the advantages and benefits of using functions in programming:

- Abstraction
  - Functions hide complex operations behind simple, easy-to-use interfaces
  - You can use a function without needing to understand all the details of how it works internally
- Organization
  - Functions help break down complex problems into smaller, manageable pieces
  - A well-organized series of function calls is much easier to understand than a long block of detailed code
- Reusability
  - Once you write a function, you can use it again in similar contexts

  - Functions can be shared across different projects
  - This saves time and reduces the chance of errors from rewriting the same code

- Testability
  - Functions make it easier to test your code in small, isolated pieces
  - You can verify each function works correctly on its own
  - If individual functions work properly, you can more confidently combine them into larger programs

Functions are the building blocks of your programs. Each block performs a specific task, and when you combine them thoughtfully, you can create complex programs that solve real-world problems. The key is to make each function clear and focused – if you can understand what each piece does on its own, it becomes much easier to work with them as a whole.

4.2 Function Syntax

Now that we understand the basic concept of functions, let's look at how to write them in Python. In a sense, you will need to learn the grammar of writing functions in Python.

Here's a simple example that checks if a DNA sequence ends with certain stop codons:

```
def ends_with_stop_codon(dna_sequence, stop_codons):
    """Check if the given DNA sequence ends with any of the given stop codons."""
    result = dna_sequence.endswith(stop_codons)
    return result


stop_codons = ("TAA", "TAG", "TGA")
result = ends_with_stop_codon("ATGAAACCACTGGTGGTTAA", stop_codons)
print(result)
```

```
True
```

Let's break down that down:

Function Definition

```
def ends_with_stop_codon(dna_sequence, stop_codons): ...
```

- The `def` keyword tells Python you're creating a new function
- `ends_with_stop_codon` is the name you're giving your function
- Inside the parentheses are your parameters (`dna_sequence` and `stop_codons`)
- The colon `:` marks where the function details begin

Remember to use lowercase letters with underscores for function names (like `read_csv`, not `readCSV`).

Key Function Components

- Docstring
  - The text between triple quotes `"""..."""`
  - Explains what the function does
- Function body
  - All the indented code below the function definition
- Return statement
  - `return result`
  - Specifies what the function sends back
- Function call
  - `ends_with_stop_codon("ATGAAACCACTGGTGGTTAA", stop_codons)`
  - How you actually use the function

Indentation Matters

Python uses indentation to know what code belongs to your function. Everything indented after the function definition is part of that function:

```
def hello(name, excitement):
    """Create a greeting message."""
    msg = f"Hello, {name}"

    if excitement == "intense":
        msg += "."
    elif excitement == "happy":
        msg += "!!!"
    else:
        msg += "!"
```

```
    return msg
```

4.3 Function Parameters

The most common way of getting data into functions is through function parameters, which hold the data your function needs to work with.

A quick note about terminology you'll encounter: You'll hear people use both "parameters" and "arguments" when talking about functions, and while they're often used interchangeably, in this tutorial we will make the following distinction:

- Parameters are the names you list when you're creating your function. They are the "placeholders" that tell the function what data to expect.
- Arguments are the actual values you send to the function when you use it.

For example, in this function, `a` and `b` are the parameters of the `add` function, and `1` and `2` are arguments to the function when it is called:

```
def add(a, b):
    return a + b


add(1, 2)
```

```
3
```

Don't worry too much about mixing up these terms. The important thing is understanding that functions need a way to receive data, and parameters/arguments are how we do that.

Function parameters and arguments are very flexible in Python. Let's see how we can define and use them.

Positional parameters

Let's look at our `add` function again:

```
def add(a, b):
    return a + b
```

In this function `a` and `b` are positional parameters. When we call this function, like `add(1, 2)`, Python matches each argument to its parameter based on position – that's why they're called "positional parameters": the first argument (`1`) goes with the first parameter (`a`), and the second argument (`2`) goes with the second parameter (`b`).

This is the most basic and common way to define parameters in Python functions.

Keyword Arguments

Check out this function that "clamps" a number between an upper and lower bound:

```
def clamp(x, min, max):
    "Clamp a number between a minimum and a maximum"
    if x < min:
        return min
    elif x > max:
        return max
    else:
        return x
```

(For simplicity, we'll assume users provide valid inputs where `min` is always less than `max`.)

While this function works, it has some potential usability issues. When calling this function, it's not immediately clear which argument goes where. For instance:

- Should we put `min` before `max`, or vice versa?
- Should the value we're clamping (`x`) go first, last, or in the middle?
- Looking at a call like `clamp(37, 45, 63)`, it's not obvious which number represents what without checking the function definition.

This is where Python's keyword arguments come to the rescue! Instead of relying on the order of arguments, we can explicitly label them:

```
print(clamp(10, min=25, max=75))
```

```
25
```

The format is straightforward: `keyword_arg=value`. The real power of keyword arguments is their flexibility. You can use keywords for some arguments but not others, and arrange the arguments in the order that makes the most sense for your use case.

Here are several valid ways to call the same function:

```
print(clamp(10, max=75, min=25))
print(clamp(max=75, min=25, x=10))
print(clamp(min=25, max=75, x=10))
```

```
25
25
25
```

All these calls do exactly the same thing, but the labeled arguments make it clear what each value represents.

Rules for Using Keyword Arguments

When using keyword arguments in Python, there are several important rules to follow. Let's look at some common mistakes and how to avoid them.

Don't forget the required arguments:

```
try:
    # Where is the number to clamp?
    clamp(min=25, max=75)
except TypeError as error:
    print(error)
```

```
clamp() missing 1 required positional argument: 'x'
```

Put the positional arguments before the keyword arguments:

```
# (If you try to run this, you will get a SyntaxError.)

# The 10 needs to go first!
clamp(min=25, max=75, 10)
```

Don't provide the same argument twice:

```
try:
    # Here, we're trying to set 'min' twice
    clamp(10, 75, min=25)
except TypeError as error:
    print(error)
```

```
clamp() got multiple values for argument 'min'
```

Only use keyword arguments that the function expects:

```
try:
    # 'number' isn't a parameter name in clamp()
    clamp(number=10, min=25, max=75)
except TypeError as error:
    print(error)
```

```
clamp() got an unexpected keyword argument 'number'
```

Essentially, these rules exist to prevent ambiguity so that Python knows which arguments go with which parameters.

Default Argument Values

When creating functions, you can set default values for their arguments. This means if you call your function without specifying all the arguments, Python will use these preset defaults instead. This essentially gives you a way to create optional functional parameters. Check it out:

```
def hello(who="World"):
    print(f"Hello, {who}!")


hello("Delaware")
hello()
```

```
Hello, Delaware!
Hello, World!
```

In this case, if no argument is provided when calling `hello`, Python automatically uses "World" as the default value.

You can also use variables to set these default values:

```
place = "World"


def hello(who=place):
    print(f"Hello, {who}!")


hello("Delaware")
hello()
```

```
Hello, Delaware!
Hello, World!
```

Python sets these default values when you first define the function, not when you call it. It will look at any variables available at that moment (what we call "in scope") to determine the default value. If you're using a variable as a default value, make sure it's defined before you create your function. We'll explore the concept of scope in more detail later in the tutorial.

A Common Pitfall: Mutable Default Arguments

Let's look at a subtle but important quirk in Python functions that can catch you off guard. Consider this seemingly straightforward function:

```
def sneaky_append(number, numbers=[]):
    numbers.append(number)
    return numbers
```

This function looks like it should do something simple: take a number, add it to a list, and if no list is provided, start with an empty one. Sounds reasonable, right?

Let's see what happens when we run it multiple times:

```
print(sneaky_append(3))
print(sneaky_append(0))
print(sneaky_append(2))
```

```
[3]
[3, 0]
[3, 0, 2]
```

Surprise! Instead of getting three separate lists, each containing one number, we get a single list that keeps growing. What's going on here?

The key is understanding when Python creates that empty list we used as a default argument. Python creates it once when the function is defined, not each time the function is called. This means all calls to the function are sharing the same list!

Here's a better way to write this function:

```
def cozy_append(number, numbers=None):
    if numbers is None:
        numbers = []

    numbers.append(number)
    return numbers


print(cozy_append(3))
print(cozy_append(0))
print(cozy_append(2))
```

```
[3]
[0]
[2]
```

In this improved version, we use `None` as our default argument and create a new empty list inside the function each time it's called. This gives us the behavior we actually wanted: a fresh, empty list for each function call when no list is provided.

The takeaway? Be *very* careful about using mutable objects (like lists, dictionaries, or sets) as default arguments. Instead, use `None` as your default and create your mutable object inside the function.

Combining Keyword Arguments & Default Arguments

Keyword arguments and default arguments look pretty similar. Let's take a look at them in the same example.

Keyword arguments can be combined with default arguments as well. Let's turn the `min` and `max` of our clamp function to default arguments.

```
def clamp(x, min=25, max=75):
    "Clamp a number between a minimum and a maximum"
    if x < min:
        return min
    elif x > max:
        return max
    else:
        return x


# You can override just one default argument.
print(clamp(10, min=5))  # Uses the default max=75

# All these calls do the same thing

# Using keyword arguments for the default parameters
print(clamp(10, max=75, min=25))

# Using all keyword arguments
print(clamp(max=75, min=25, x=10))

# Different order, same result
print(clamp(min=25, max=75, x=10))
```

```
10
25
25
25
```

There are a few important points to understand here:

- Default arguments (`min=25, max=75` in the function definition) provide fallback values when those arguments aren't specified in the function call.
- Keyword arguments (like `min=5` in the function call) let you specify which parameter you're passing a value to.
- Even required parameters (like `x`) can be passed using keyword syntax (`x=10`), though this isn't required.
- When using keyword arguments, the order doesn't matter as long as keyword arguments follow any positional arguments.

This flexibility can be useful when working with functions that have multiple parameters, as it can help make your code more readable and prevent mistakes from mixing up the order of arguments.

Functions That Can Take Any Number of Arguments

Sometimes you want to write a function that can handle different numbers of inputs. Python makes this possible with variable-length arguments. Let's explore how this works.

Taking Multiple Positional Arguments

To specify that a function takes a variable number of positional arguments, you use the `*args` syntax. The asterisk (`*`) tells Python to accept any number of arguments and pack them into a tuple. Sometimes you will see these called *variadic* arguments. Check it out:

```
def process_samples(*samples):
    for sample in samples:
        print(f"Processing sample: {sample}")


# Works with any number of arguments
process_samples("A1")  # One sample
process_samples("A1", "A2", "B1", "B2")  # Three samples
```

```
Processing sample: A1
Processing sample: A1
Processing sample: A2
Processing sample: B1
Processing sample: B2
```

You can combine the variadic arguments with positional arguments, as long as the formally specified positional arguments come first:

```
def print_stuff(separator, *words):
    msg = separator.join(words)
    print(msg)


print_stuff(", ", "apple", "banana", "cherry", "domino's pizza")
```

```
apple, banana, cherry, domino's pizza
```

The `*words` parameter tells Python to collect all remaining arguments into a tuple called `words`. Having to specify the separator first before all the words you want to print feels a bit awkward to me. We can improve this making `separator` a keyword argument that comes last:

```
def print_stuff(*words, separator):
    msg = separator.join(words)
    print(msg)


print_stuff("apple", "banana", "cherry", "domino's pizza", separator=", ")
```

```
apple, banana, cherry, domino's pizza
```

When you have an argument that follows the variadic arguments, it must be used like a keyword argument. For example, this doesn't work:

```
try:
    print_stuff("apple", "banana", "cherry", "domino's pizza", ", ")
except TypeError as error:
    print(error)
```

```
print_stuff() missing 1 required keyword-only argument: 'separator'
```

Without using a keyword argument, Python has no way of knowing that the final argument you passed into the function shouldn't be part of the `words` tuple.

Finally, it is very common in these situations that a default argument be provided to the argument that comes after a variadic argument list:

```
def print_stuff(*words, separator=", "):
    msg = separator.join(words)
    print(msg)


print_stuff("apple", "banana", "cherry", "domino's pizza")
```

```
apple, banana, cherry, domino's pizza
```

Collections & Variadic Arguments

Here's something that might surprise you:

```
foods = ["apple", "banana", "cherry", "domino's pizza"]

try:
    print_stuff(foods)
except TypeError as error:
    print(error)
```

```
sequence item 0: expected str instance, list found
```

When we pass a list directly, Python treats it as a single argument. To tell Python to treat each item in the list as a separate argument, we need to "unpack" the list using the * operator:

```
foods = ["apple", "banana", "cherry", "domino's pizza"]
print_stuff(*foods)  # This works!
```

```
apple, banana, cherry, domino's pizza
```

Let's look at another example that shows how this behavior can be tricky:

```
def greeter(*names):
    for name in names:
        print(f"Hello, {name}!")


# This passes three separate arguments
greeter("Ryan", "Pikachu", "Shaq")

# This passes one argument (a list)
print()
greeter(["Ryan", "Pikachu", "Shaq"])

# This "unpacks" the list again into multiple arguments
print()
greeter(*["Ryan", "Pikachu", "Shaq"])
```

```
Hello, Ryan!
Hello, Pikachu!
Hello, Shaq!

Hello, ['Ryan', 'Pikachu', 'Shaq']!

Hello, Ryan!
Hello, Pikachu!
Hello, Shaq!
```

Compare that to the version of `greeter` that takes a single collection argument rather than a variable number of arguments.

```python
def greeter(names):
    for name in names:
        print(f"Hello, {name}!")


# This passes three separate arguments
try:
    greeter("Ryan", "Pikachu", "Shaq")
except TypeError as error:
    print(error)

# This passes one argument (a list)
print()
greeter(["Ryan", "Pikachu", "Shaq"])

# This "unpacks" the list again into multiple arguments
print()

try:
    greeter(*["Ryan", "Pikachu", "Shaq"])
except TypeError as error:
    print(error)
```

```
greeter() takes 1 positional argument but 3 were given

Hello, Ryan!
Hello, Pikachu!
Hello, Shaq!
```

```
greeter() takes 1 positional argument but 3 were given
```

As you see, we had to wrap some of these in try/except blocks since they are using the function incorrectly.

Taking Multiple Keyword Arguments

To specify that a function takes a variable number of keyword arguments, you use the `**kwargs` syntax. The double asterisk (`**`) tells Python to accept any number of arguments and pack them into a dictionary.

Let's look at an example that creates a simple product description:

```
def keyword_example(price, **keywords):
    print(f"the price is ${price}")
    for keyword, value in keywords.items():
        print(f"{keyword:>8s} => {value}")


keyword_example(
    7.99,
    fruit="apple",
    dessert="tart",
    taste="yum!",
    coolness="very",
)
```

```
the price is $7.99
   fruit => apple
 dessert => tart
   taste => yum!
coolness => very
```

In this function, `price` is a regular parameter that must be provided, while `**keywords` captures any additional keyword arguments as a dictionary. The function then prints each keyword and its value.

When using `**kwargs`, it must be the last parameter in your function definition. If you try to put other parameters after it, Python will raise an error. (Try it yourself by swapping the order of `price` and `**keywords` to see what happens!) Contrast this with the variable length positional arguments, which *could* come before some keyword-only arguments.

You can also do the reverse: take a dictionary and "unpack" it into keyword arguments. Here's an example:

```
def greeter(**greetings):
    for greeting, people in greetings.items():
        for person in people:
            print(f"{greeting} {person}")


greeter(hello=["Ash", "Pikachu"], goodbye=["Gary", "Eevee"])

print()

greetings = {"hello": ["Ash", "Pikachu"], "goodbye": ["Gary", "Eevee"]}
greeter(**greetings)
```

```
hello Ash
hello Pikachu
goodbye Gary
goodbye Eevee

hello Ash
hello Pikachu
goodbye Gary
goodbye Eevee
```

Both calls to `greeter()` produce the same output, showing two different ways to pass keyword arguments to a function.

Combining Variable-Length Arguments

Python gives you the flexibility to combine regular parameters with variable-length arguments in the same function. Here's how it works:

```
def example(a, b, *arguments, **keyword_arguments):
    print("a:", a)
    print("b:", b)
    print("arguments:", arguments)
    print("keyword_arguments:", keyword_arguments)


example(1, 2, 3, 4, 5, hello="world", color="orange")
```

```
a: 1
b: 2
arguments: (3, 4, 5)
keyword_arguments: {'hello': 'world', 'color': 'orange'}
```

In this example:

- `a` and `b` are regular parameters that must be provided
- `*arguments` collects any extra positional arguments (here: `3, 4, 5`)
- `**keyword_arguments` collects any extra `name=value` pairs (here: `hello="world"`, `color="orange"`)

You'll often see these written in a shorter form as `*args` and `**kwargs` in Python code:

```
def example(a, b, *args, **kwargs): ...
```

This is just a common convention - `args` stands for "arguments" and `kwargs` for "keyword arguments". The important part is the `*` and `**` symbols, not the names themselves.

Check out this guide, Python args and kwargs: Demystified, if you want to dive deeper into this topic.

Controlling How Arguments Are Passed

When we call functions in Python, we can typically pass arguments in two ways: by their position in the parameter list or by explicitly naming them with keywords. However, Python also lets you set stricter rules about how arguments should be passed, requiring some to be positional-only or keyword-only. Here is how that looks:

```
def example(a, b, /, c, d, *, e, f): ...
```

This function's parameters are divided into three groups:

- Positional-only parameters (before the `/`):
  - `a` and `b` must be passed by position
- Standard parameters (between `/` and `*`):
  - `c` and `d` can be passed either by position or keyword
- Keyword-only parameters (after the `*`):
  - `e` and `f` must be passed by keyword

Here are some examples of valid and invalid ways to call the `example` function:

```
# Valid calls

example(1, 2, 3, 4, e=5, f=6)
example(1, 2, c=3, d=4, e=5, f=6)

# Invalid calls

try:
    example(a=1, b=2, c=3, d=4, e=5, f=6)
except TypeError as error:
    print(error)


try:
    example(1, 2, 3, 4, 5, 6)
except TypeError as error:
    print(error)
```

```
example() got some positional-only arguments passed as keyword arguments: 'a, b'
example() takes 4 positional arguments but 6 were given
```

Don't worry if this syntax seems tricky. While Python offers advanced parameter options, you won't need them too much when learning. Focus on mastering basic positional and keyword arguments first, as these cover most programming needs. For more details on advanced features, check the Python documentation on special parameters.

4.4 Return Values

Every function in Python can give back (or "return") a value when it finishes running. We use the `return` keyword to specify what value we want to get back:

```
import random

def random_number():
    return random.random()

print(random_number())
print(random_number())
```

Let's explore some interesting ways we can use `return` statements.

Returning Multiple Values

Sometimes we want a function to give back more than one piece of information. While Python doesn't technically return multiple values at once, we can package multiple values into a tuple and return that instead:

```
def gc_count(dna_sequence):
    g_count = dna_sequence.count("G")
    c_count = dna_sequence.count("C")

    return g_count, c_count


dna_sequence = "ACTGACTG"

# We can unpack the returned values into separate variables
g_count, c_count = gc_count(dna_sequence)
print(g_count, c_count, sep=", ")

# Behind the scenes, Python is creating a tuple
print(type(gc_count(dna_sequence)))
```

```
2, 2
<class 'tuple'>
```

This is a convenient way to get multiple values out of a function. Python makes it easy to unpack these values into separate variables that we can use later in our code.

Functions With No Return Value

What happens when a function doesn't explicitly return anything? Check out this little function that just prints a greeting:

```
def say_hello(name):
    msg = f"Hello, {name}!"
    print(msg)


say_hello("Luka")
```

```
Hello, Luka!
```

When we don't specify a return value, Python automatically returns a special value called `None`:

```
result = say_hello("LeBron")
print("the result of `say_hello` is:", result)
```

```
Hello, LeBron!
the result of `say_hello` is: None
```

`None` is Python's way of representing "nothing" or "no value." We can also explicitly return `None` when we want to indicate that a function couldn't produce a meaningful result. Here's an example:

```
def find_motif(dna_sequence, motif):
    """
    Find the position of a motif in a DNA sequence.

    Args:
        dna_sequence (str): DNA sequence to search
        motif (str): Motif to find

    Returns:
        position (int or None): Starting position of motif if found, None otherwise
    """
    position = dna_sequence.find(motif)

    if position == -1:
        return None
    else:
        return position


sequence = "ATCGTATAGCAT"
print(find_motif(sequence, "TATA"))
print(find_motif(sequence, "GGGC"))
```

```
4
None
```

In this example, if we can't find the motif, we return `None` to indicate that no result was found. This is a pretty common pattern. You'll see it in built-in methods too, like the dictionary `get()` method:

```python
d = {"a": 1, "b": 2}

# Tries to get a value that doesn't exist
result = d.get("c")

print(result)
```

```
None
```

`None` is particularly useful when you need to indicate the absence of a value or when a function couldn't complete its intended task successfully.

Aside: Early Returns & Conditional Expressions

Let's look at a few different ways to write the same function. We'll use our motif-finding code as an example:

```python
def find_motif(dna_sequence, motif):
    position = dna_sequence.find(motif)

    # Standard if/else
    if position == -1:
        return None
    else:
        return position
```

While this standard if/else structure works perfectly well, you might encounter two other common patterns when reading Python code:

Early Return Pattern

Here is the Early Return Pattern:

```python
def find_motif_version_2(dna_sequence, motif):
    position = dna_sequence.find(motif)

    if position == -1:
        # Exit the function early if no match is found
        return None
```

```
    # Otherwise return the position
    return position
```

In this function, using the early return pattern doesn't really make that much of a difference, because it is small. You could imagine a larger function that might have a few more checks leading to more nesting of conditional statements. In these cases, the early return pattern can really shine. Check out intentionally complicated example that takes some gene expression data and does some checks on it:

```
# Without early returns => complex nesting and logic
def process_gene_data(gene_id, sequence, expression_level):
    if gene_id is not None:
        if sequence is not None:
            if len(sequence) >= 50:
                if expression_level is not None:
                    if expression_level > 0:
                        # Actually process the data
                        processed_data = {
                            "id": gene_id,
                            "sequence": sequence,
                            "expression": expression_level,
                            "normalized": expression_level / 100,
                        }
                        return processed_data
                    else:
                        return {"error": "Expression level must be positive"}
                else:
                    return {"error": "Missing expression level"}
            else:
                return {"error": "Sequence too short"}
        else:
            return {"error": "Missing sequence"}
    else:
        return {"error": "Missing gene ID"}
```

Let's untangle that mess using the early return pattern:

```
def process_gene_data(gene_id, sequence, expression_level):
    if gene_id is None:
        return {"error": "Missing gene ID"}
```

```
    if sequence is None:
        return {"error": "Missing sequence"}

    if len(sequence) >= 50:
        return {"error": "Sequence too short"}

    if expression_level is None:
        return {"error": "Missing expression level"}

    if expression_level <= 0:
        return {"error": "Expression level must be positive"}

    # At this point, we know the data is valid, so we can process it.
    processed_data = {
        "id": gene_id,
        "sequence": sequence,
        "expression": expression_level,
        "normalized": expression_level / 100,
    }

    return processed_data
```

Again, this is an intentionally complex example to prove a point about early returns, but the point still stands: the second version is *much* less complex. It shows how early returns can:

- Make input validation clearer and more maintainable
- Avoid deeply nested conditional statements
- Separate validation logic from the main processing
- Make it easier to add new validation steps later

Conditional Expressions

The other pattern is using a conditional expressions (also called a ternary operator) for the return value:

```
def find_motif_version_2(dna_sequence, motif):
    position = dna_sequence.find(motif)

    # Conditional expression
    return None if position == -1 else position
```

You can also flip the condition around if it makes more sense for your specific case:

```python
def find_motif_version_3(dna_sequence, motif):
    position = dna_sequence.find(motif)

    # Conditional expression with the check reversed
    return position if position != -1 else None
```

Recap

All these versions do exactly the same thing, but they express it in slightly different ways:

- The standard if/else is the most verbose but also the most explicit
- The early return pattern can make code more readable when you have multiple conditions to check
- The conditional expression is more compact but might take some getting used to

There's no "best" way – they're all valid Python. However, it's good practice to be consistent within a single script or project. In situations like these, it can often make sense to pick the style that makes the most sense to you and your colleagues and stick with it.

Check out PEP 308 for the full story behind Python's conditional expressions.

4.5 Scope & Name Resolution: Where Can You Use Your Variables?

Scope is a set of rules that determine where you can use different names in your code (like variables, function names, etc.). Scope is essentially the "visibility" of variables in different parts of your code. Let's look at a simple example to understand this better:

```python
def some_genes():
    # Create a list of genes inside the function
    genes = ["nrdA", "nrdJ"]

    # We can use `genes` here because we're inside the function
    # where it was created
    print(genes)


# Let's try to access `genes` outside the function
try:
    print(genes)
```

```
except NameError as error:
    print(error)
```

```
name 'genes' is not defined
```

In this code, `genes` only exists inside the `some_genes` function. When we're working inside that function, `genes` is "in scope", meaning we can use it. However, as soon as we step outside the function, `genes` goes "out of scope", and we can't access it anymore.

When does a name come into scope? In Python, this happens when you first create it, which can happen in several ways:

- Assigning a value to a variable (`x = 1`)
- Importing a module (`import random`)
- Defining a function (`def add(): ...`)
- Creating function parameters (`def add(x, y): ...`)

Next, let's look at how Python decides whether a name is in scope or not.

Understanding Scope: How Python Finds Variables

When you write code, Python needs to know where to look for the variables and functions you're trying to use. It follows a specific search pattern called the LEGB Rule, which defines four levels of scope (think of scope as the "visibility" of variables in different parts of your code).

Let's break down these levels from smallest to largest:

- Local Scope (L)
  - This is the most immediate scope, created whenever you define a function
  - Variables created inside a function only exist inside that function
  - Each time you call the function, Python creates a fresh local scope
- Enclosing Scope (E)
  - This comes into play when you have a function inside another function
  - The outer function's scope becomes the "enclosing" scope for the inner function
  - The inner function can "see" and use variables from the outer function
  - We'll explore this more when we discuss nested functions
- Global Scope (G)
  - These are variables defined in the main body of your script
  - They're accessible throughout your program

- Built-in Scope (B)
  - This is Python's pre-loaded toolkit
  - Includes all of Python's built-in functions like `print()`, `len()`, and `max()`
  - These tools are always available, no matter where you are in your code

When you use a variable name, Python searches through these scopes in order (L → E → G → B) until it finds a match. If it can't find the name anywhere, you'll get a `NameError`.

Let's go through each of these in more detail.

Local Scope: What Happens in a Function, Stays in a Function

When we write functions, we create what's called a "local scope", which is sort of like the function's private workspace. Any variables we create inside a function only exist inside that function. Once the function finishes running, these variables disappear, and we can't access them from outside the function.

Important Note about Quarto and Jupyter Notebooks: In our notebook environment, code cells can see variables that were created in previous cells. For our examples to make sense, we need to start fresh. We'll use a special notebook command called "reset" to clear out any existing variables.

```
# This is a special notebook command that clears all our previous variables.
# You won't use this in regular Python programs, it's just for notebooks.
%reset -f


def example(x):
    # Here, we can use 'x' because it's a parameter passed to our function
    # We can also create new variables like 'y'
    y = 10
    print(x * y)


# This works fine
example(2)

# But watch what happens when we try to use these variables outside the
# function.

try:
    # This will be an error because 'x' doesn't exist out here
    print(x)
```

```
except NameError as error:
    print(error)

try:
    # Same problem with 'y': it only existed inside the function
    print(y)
except NameError as error:
    print(error)
```

```
20
name 'x' is not defined
name 'y' is not defined
```

This behavior is intentional! It helps keep our functions self-contained and prevents them from accidentally interfering with code outside the function. By default, functions can read variables from the global scope (outside the function), but they can't modify them or create new global variables from inside the function.

Enclosing Scope: Functions Within Functions

Sometimes we write functions inside other functions (nested functions). When we do this, the inner function has access to variables defined in the outer function. This relationship creates what we call an "enclosing scope."

Let's look at an example:

```
%reset -f


def outer():
    # These variables are accessible throughout outer() and inner()
    x = 1
    y = 2

    def inner():
        # This creates a new 'y' that's different from the outer y
        y = 10
        print(f"from inner -- x: {x}, y: {y}")
        return x + y

    # Calculate z using inner()
```

```
    z = inner()

    print(f"from outer -- x: {x}, y: {y}, z: {z}")


outer()

# Once outer() finishes running, we can't access any of its variables
# or the inner() function anymore:

try:
    x
except NameError as error:
    print(f"trying to access 'x' in global scope -- {error}")

try:
    y
except NameError as error:
    print(f"trying to access 'y' in global scope -- {error}")

try:
    z
except NameError as error:
    print(f"trying to access 'z' in global scope -- {error}")

try:
    inner()
except NameError as error:
    print(f"trying to access 'inner' in global scope -- {error}")
```

```
from inner -- x: 1, y: 10
from outer -- x: 1, y: 2, z: 11
trying to access 'x' in global scope -- name 'x' is not defined
trying to access 'y' in global scope -- name 'y' is not defined
trying to access 'z' in global scope -- name 'z' is not defined
trying to access 'inner' in global scope -- name 'inner' is not defined
```

In this example, `inner()` can "see" and use variables defined in `outer()` because `outer()` is its enclosing scope. Think of it like `inner()` being contained within `outer()`'s environment.

A few key points:

- Variables defined in `outer()` are accessible inside `inner()`
- If `inner()` creates a variable with the same name as one in `outer()`, the inner version takes precedence inside `inner()` (like `y` in our example)
- Once `outer()` finishes running, none of its variables or the `inner()` function are accessible anymore

This concept of enclosing scope is useful when you want to create a function that needs access to variables from its surrounding context while keeping those variables private from the rest of your program.

Global Scope: The World Is Your Variable

Next up is global scope, which refers to variables that are defined at the top level of your Python script. While we'll dive deeper into this when we cover modules, here's what you need to know for now:

```
%reset -f

name = "Pikachu"


def say_hi():
    print(f"Hi, {name}!")


say_hi()
```

```
Hi, Pikachu!
```

In this example, `name` is a global variable because it's defined outside of any function. Functions can "see" and use global variables, which is why our `say_hi()` function can access `name`.

However, there's an important limitation: functions can't modify global variables directly. Here's what happens when we try:

```
%reset -f

best_player = "Messi"


def update_best_player():
```

```
    best_player = "Ronaldo"


print(best_player)
update_best_player()
print(best_player)
```

```
Messi
Messi
```

Notice that `best_player` stays as "Messi" even after we run the function. When we create `best_player = "Ronaldo"` inside the function, we're actually creating a new, local variable that only exists inside the function. It doesn't affect the global `best_player`.

There is a way to modify global variables from within a function using the `global` keyword, though it's generally not recommended:

```
%reset -f

best_player = "Messi"


def update_best_player():
    # This tells Python we want to modify the global variable
    global best_player
    best_player = "Ronaldo"


print(best_player)
update_best_player()
print(best_player)
```

```
Messi
Ronaldo
```

While using `global` works, it's usually better to avoid it. Good programming practice encourage passing values as parameters and returning results from functions instead. This makes your code easier to understand and maintain.

(There's also a similar feature called `nonlocal` for nested functions, but that's a topic for another day!)

Changing Mutable Values from Within a Function

Let's look at how functions can modify certain types of data (like lists) that exist outside the function. This is a common pattern in Python, so it's important to understand what's happening.

```
numbers = [1, 2]


def append():
    numbers.append(3)


append()
print(numbers)
append()
print(numbers)
```

```
[1, 2, 3]
[1, 2, 3, 3]
```

This works because lists in Python are mutable (changeable). Even though the list `numbers` is defined outside our function, we can still modify it from inside the function because Python is just following a reference to where that list lives in memory.

Here's another example that makes this concept more explicit:

```
def append_to(numbers, number):
    print(f"inside append_to, list id is {id(numbers)}")
    numbers.append(number)


some_numbers = [10, 11]
print(f"list {id(some_numbers)}: {some_numbers}")

append_to(some_numbers, 12)
print(f"list {id(some_numbers)}: {some_numbers}")

append_to(some_numbers, 13)
print(f"list {id(some_numbers)}: {some_numbers}")
```

```
list 4461305344: [10, 11]
inside append_to, list id is 4461305344
list 4461305344: [10, 11, 12]
inside append_to, list id is 4461305344
list 4461305344: [10, 11, 12, 13]
```

In this version, we're being more explicit by passing the list as an argument to our function. The `id()` function shows us the memory address where our list lives. Notice how this ID stays the same throughout the program – that's because we're always working with the same list, just referring to it by different names (`some_numbers` outside the function and `numbers` inside it).

This behavior highlights an important aspect of how Python handles variables and mutable objects: variables are like name tags that point to data, rather than containers that hold data. When we pass a list to a function, we're giving that function another name tag that points to the same data. We'll revisit this topic again when we cover classes later in the course.

Built-In Scope: Python's Ready-to-Use Tools

Think of Python's built-in scope as your basic toolkit that's always available when you start Python. These are the fundamental tools that Python automatically loads for you, and they're the last place Python looks when trying to find where a name is defined (remember our LEGB rule).

You've already been using built-in functions throughout this course. For example:

```
total = sum(range(6))
print(total)
```

```
15
```

These functions like `sum()` and `print()` aren't magic, they actually live in a specific place in Python called the `builtins` module, which contains essential tools like built-in functions, constants, and error types that you can use anywhere in your code.

You can see what's available using Python's special `__builtins__` name:

```
# Let's look at the first 10 built-in tools
for thing in dir(__builtins__)[0:10]:
    print(thing)
```

```
ArithmeticError
AssertionError
AttributeError
BaseException
BaseExceptionGroup
BlockingIOError
BrokenPipeError
BufferError
BytesWarning
ChildProcessError
```

While you normally don't need to, you can also access these tools directly through the `builtins` module:

```
import builtins

total = builtins.sum(builtins.range(6))
builtins.print(total)

%reset -f
```

```
15
```

This explicit way of accessing built-ins isn't common practice. It's just helpful to understand where these tools actually live in Python's structure.

LEGB Recap

```
 BUILT-IN SCOPE
(print, len, str, etc.)
  GLOBAL SCOPE
  global_var = 100
    ENCLOSING SCOPE
    def outer_function():
        outer_var = 200
      LOCAL SCOPE
      def inner_function():
          local_var = 300
          print(local_var)
          print(outer_var)
          print(global_var)
```

- **Built-in Scope**: Contains Python's pre-built functions and tools that are always available
- **Global Scope**: Where your main program variables live
- **Enclosing Scope**: Contains variables from outer functions
- **Local Scope**: Contains variables defined within the current function

When your code needs to use a variable, Python looks for it starting from the innermost scope (local) and works its way outward until it finds it. This is why you can use global variables inside functions, but you can't use local variables outside their functions.

Special Cases in Python Scope

While we've covered the main rules of scope, Python has a few special situations that work a bit differently. Let's look at three interesting cases that might surprise you.

List Comprehensions Keep Their Variables Private

When you use a list comprehension (that shorthand way to create lists), Python handles its variables differently than regular loops. Here's what I mean:

```
%reset -f

# This list comprehension doubles each number from 0 to 4
numbers = [x * 2 for x in range(5)]
print(numbers)

# Try to access 'x' - it won't work!
try:
    x
except NameError as error:
    print(error)
```

```
[0, 2, 4, 6, 8]
name 'x' is not defined
```

Compare this to a regular for loop, where the loop variable sticks around:

```
%reset -f

# Regular for loop doing the same thing
numbers = []
for x in range(5):
    numbers.append(x * 2)
print(numbers)

# Here we can still use 'x' - it exists!
print(x)
```

```
[0, 2, 4, 6, 8]
4
```

Error Variables Stay in Their Block

When handling errors (which we'll cover more in a later tutorial), variables created in `except` blocks are only available within that block:

```
try:
    1 / 0
except ZeroDivisionError as error:
    print(error)  # Works here

# Try to use 'error' later...
try:
    print(error)  # Won't work!
except NameError:
    print("can't get the original 'error' here")
```

```
division by zero
can't get the original 'error' here
```

These special cases are part of Python's design to prevent variables from accidentally leaking into parts of your code where they weren't intended to be used. While they might seem like quirks at first, they actually help keep your code cleaner and more predictable.

4.6 Best Practices for Writing Functions

When you're learning to write functions, you'll find lots of advice online, some of it contradictory! However, most guidelines share a common goal: making your code easier to understand and maintain. Here are some practical tips to help you write better functions from the start.

Use Clear, Meaningful Names

Function names are like labels on laboratory equipment – they should clearly indicate what they do:

```
# Not very helpful
def calc(x, y):
    return x * y


# Much clearer
def area(length, width):
    return length * width
```

Stay consistent with your naming style. Pick one approach and stick with it:

```
# Inconsistent naming
def area(length, width): ...
def getPerimeter(l, w): ...
def calcvolume(x, y, z): ...


# Better -- consistent and clear
def area(length, width): ...
def perimeter(length, width): ...
def volume(length, width, height): ...
```

Keep Functions Focused and Concise

Each function should do one specific task well. If you find your function doing multiple different things, it's usually better to split it into smaller, more focused functions.

Here's an example analyzing a DNA sequence:

```
# Too many tasks in one function
def process_dna_sequence(sequence):
    has_start = sequence.startswith("ATG")
    has_stop = sequence.endswith(("TAA", "TAG", "TGA"))
    gc_content = (sequence.count("G") + sequence.count("C")) / len(sequence)
    return has_start, has_stop, nucleotide_counts, gc_content


# Better: Split into focused functions
def has_start_codon(sequence):
    return sequence.startswith("ATG")


def has_stop_codon(sequence):
    return sequence.endswith(("TAA", "TAG", "TGA"))


def calculate_gc_percentage(sequence):
    gc_count = sequence.count("G") + sequence.count("C")
    return gc_count / len(sequence) * 100
```

Make Function Behavior Clear and Predictable

When writing functions, make it obvious what data the function needs to do its job. Pass required data directly to the function rather than having it rely on information defined elsewhere in your code.

```
# Less clear -- relies on external data
measurements = []


def add_measurement(new_measurement):
    measurements.append(new_measurement)
    return sum(measurements) / len(measurements)


# Better -- everything the function needs is passed directly
def add_measurement(measurements, new_measurement):
    measurements.append(new_measurement)
    return sum(measurements) / len(measurements)
```

When Should You Write a Function?

Consider writing a function when:

- You have a specific task that you can clearly define
- You find yourself copying and pasting similar code multiple times
- You want to make complex operations easier to understand
- You plan to reuse the same code in different parts of your project

Summary

These guidelines will help you write functions that are easier to understand, test, and maintain. As you gain experience, you'll develop an intuition for when and how to apply these practices. Remember, the goal is to write code that both you and others can easily work with, even months or years after it was written.

4.7 Function Documentation: Making Your Code Clear and Useful

You could think of documentation as leaving helpful instructions for yourself and others about how your code works. While it might seem tedious at first, good documentation is like a roadmap that helps people understand and use your code correctly.

Let's look at an example of a function without documentation:

```
def math_monster_addition(a, b):
    if a >= b:
        return a + b
    else:
        return a - b
```

Looking at this code, you can figure out what it does, but not why it exists or when you should use it. Let's improve it by adding proper documentation:

```
def add_or_subtract(a, b):
    """
    Performs addition or subtraction based on the relative values of two numbers.

    This function models the Math Monster's arithmetic behavior.  When the
    first number is bigger (or equal), the monster is happy and adds the
    numbers.  When the first number is smaller, the monster gets grumpy and
    subtracts them instead.
```

```
    Args:
        a (numeric): First number
        b (numeric): Second number

    Returns:
        numeric: The sum of a and b if a is greater than or equal to b,
          otherwise returns the difference (a - b).

    Examples:
        >>> add_or_subtract(5, 3)  #=>  8
        >>> add_or_subtract(2, 7)  #=> -5
        >>> add_or_subtract(1, 1)  #=>  2
    """
    if a >= b:
        return a + b
    else:
        return a - b
```

Key Parts of Good Documentation

- **Summary Line**: A brief, clear statement of what the function does.
- **Description**: More detailed explanation of the function's purpose and behavior.
- **Args**: List of parameters with their types and descriptions.
- **Returns**: What the function gives back and under what conditions.
- **Examples**: Real-world usage examples showing inputs and outputs.

Documentation Style

While there are several ways to format documentation (like Google's style shown above), what matters most is consistency. Pick one style and stick with it throughout your project. You can explore different styles in Real Python's documentation guide.

Why Documentation Matters

Good documentation:

- Makes your code more accessible to others
- Helps you remember how your own code works months later
- Can reveal problems in your code design
- Makes your code more maintainable

- Enables automatic documentation generation

A Note on Writing Documentation

If you find it difficult to write clear documentation for a function, it might be a sign that the function is too complex or trying to do too many things. Use documentation as an opportunity to review and potentially simplify your code.

Remember: The goal isn't to document every single line of code, but to provide enough information so that someone (including future you) can understand and use your code effectively.

4.8 Advanced Topics

Let's explore a few more interesting features of Python functions that you might encounter in your programming journey.

Function Names and Aliases

Think of function names as labels. You can create multiple labels (aliases) that point to the same function, similar to how you might have different names for the same thing:

```
def add1(x):
    return x + 1


# Create another name for the same function
add_one = add1

print(add1(2))  # Prints 3
print(add_one(2))  # Also prints 3
```

```
3
3
```

This can be useful when you want to use a more context-appropriate name:

```
def calculate_tax(amount):
    return amount * 0.2


# Using a more specific name for sales contexts
sales_tax = calculate_tax

price = 100
tax = sales_tax(price)  # Clearer what this represents
```

Even when two functions do exactly the same thing, Python treats them as distinct objects if they're defined separately:

```
def add1(x):
    return x + 1


def add_one(x):
    return x + 1


print("add1 points to", add1)
print("add_one points to", add_one)
print("add1 == add_one is", add1 == add_one)

print(f"add1(2) == {add1(2)}")
print(f"add_one(2) == {add_one(2)}")
```

```
add1 points to <function add1 at 0x109e5a340>
add_one points to <function add_one at 0x109e58540>
add1 == add_one is False
add1(2) == 3
add_one(2) == 3
```

Functions as Objects

In Python, functions are objects that you can work with just like numbers or strings. Here's a practical example using Python's `sorted` function:

```
# Default sorting (alphabetical)
words = ["apple", "pie", "is", "good"]
sorted_words = sorted(words)
print(sorted_words)

# Sorting by length instead.  Here, `len` is the built-in length function.
sorted_words = sorted(words, key=len)
print(sorted_words)
```

```
['apple', 'good', 'is', 'pie']
['is', 'pie', 'good', 'apple']
```

We can also write our own functions to customize sorting. Here's an example with student grades:

```
def get_grade(student):
    return student[1]


student_grades = [("Pikachu", 97), ("Charmander", 91), ("Bulbasaur", 86)]
sorted_student_grades = sorted(student_grades, key=get_grade)
print(sorted_student_grades)
```

```
[('Bulbasaur', 86), ('Charmander', 91), ('Pikachu', 97)]
```

Lambda Expressions

Sometimes writing a full function is overkill for a simple operation. That's where lambda expressions come in – they're tiny, unnamed functions:

```
student_grades = [("Pikachu", 97), ("Charmander", 91), ("Bulbasaur", 86)]
sorted_student_grades = sorted(student_grades, key=lambda student: student[1])
print(sorted_student_grades)
```

While you can store lambda functions in variables, it's usually better to write a regular function if you plan to reuse the code.

Type Hints

You might see functions written with extra information about their inputs and outputs:

```
def greet(name: str) -> str:
    return f"Hello, {name}!"


def add(a: int, b: int) -> int:
    return a + b
```

These are called type hints. They help document:

- What kind of data a function expects (like `name: str` meaning "name should be a string")
- What kind of data it returns (like `-> str` meaning "returns a string")

Key points about type hints:

- They're optional
- Python doesn't enforce them automatically
- They're fairly common in larger projects
- You can ignore them while learning Python

Type hints are helpful for documentation and code maintenance, but your code will work perfectly fine without them!

4.9 Wrap-Up

Functions are one of the most fundamental and powerful concepts in Python programming. In this tutorial, we've covered everything from basic function syntax to advanced topics like variable-length arguments and scope rules. You've learned how to write clear, reusable code by packaging logic into well-named functions, how to work with different types of parameters, and how to use return values effectively. We've also explored important concepts like variable scope, documentation best practices, and some advanced features like lambda expressions and type hints. With this foundation in functions, you're now equipped to write more organized, maintainable, and self-documenting Python code that can be easily shared and reused across your bioinformatics projects.

4.10 Suggested Readings

You might enjoy checking out some of these resources:

- Python Docs: 4.8. Defining Functions
- Google Python Style Guide: Functions and Methods
- Real Python: Python Scope & the LEGB Rule: Resolving Names in Your Code
- Type hints cheat sheet

5 Introduction to Object-Oriented Programming

Think about the natural world around you for a moment. You can imagine birds, trees, cells, and countless other entities. Each of these things has specific characteristics and behaviors that help you recognize them. One way to turn these entities into code is by using Classes and Objects.

In programming terms, an *object* is sort of like a specific entity that exists in the real world:

- A particular robin building a nest outside your window
- An individual *E. coli* cell growing in your Petri dish
- A specific patient in a hospital

Just like real-world things, objects have two key aspects:

1. **Characteristic Properties**: what the object *is* or *has* (identity, color, size, weight)
2. **Behaviors**: what the object can *do* (fly, eat, divide)

For example, that robin outside your window has a particular color (characteristic) and can sing a specific song (behavior). The *E. coli* cell in your Petri dish has a particular size (characteristic) and can metabolize lactose (behavior).

If objects are the specific things (like a particular robin or *E. coli* cell) in your domain, then *classes* are the blueprints or templates that define the characteristics and behaviors of those entities.

For instance, the `Robin` *class* might specify that all robins have characteristics like species, wingspan, feather color, beak shape, plus behaviors such as flying, singing, and nest-building. But each individual robin *instance* would have its own specific values for these characteristics and behaviors that depend on those specific characteristics.

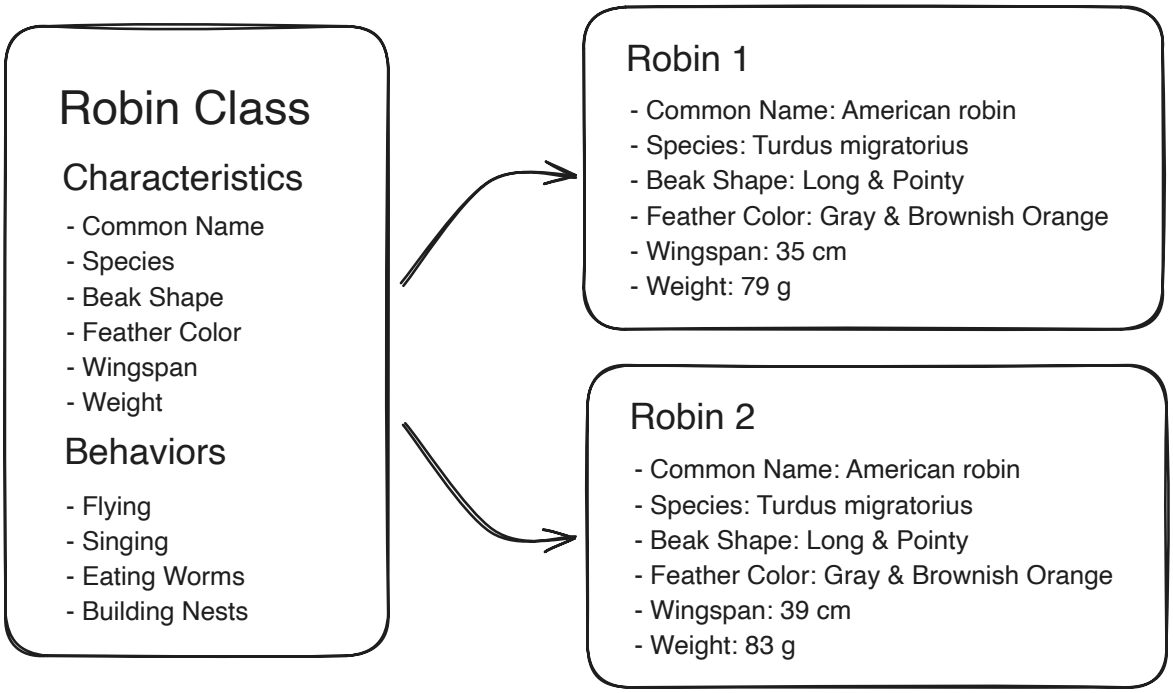


Figure 5.1: An example robin class with two instances

Classes enable us to create our own custom data types that model real-world entities. Instead of just having numbers and strings, we can have Birds, Bacteria, Patients, Proteins, and Molecules, each with their own specialized characteristics and behaviors.

This connection between our programs and the real-world domain we are modeling is what makes object-oriented programming powerful. It allows us to represent and manipulate complex biological entities in a way that feels natural and intuitive, mirroring how we already think about these systems in our research.

For the rest of this tutorial, we'll explore how to implement Object-Oriented Programming (OOP) in Python, giving you the tools to represent the biological systems you work with in a natural way.

5.1 Four Pillars of OOP

When learning about object-oriented programming (OOP), you'll often hear about four fundamental concepts (pillars) that form its foundation. These concepts help us organize code in a way that mirrors how we think about real-world objects and their relationships:

- **Encapsulation**: bundling data and methods that work on that data into a single unit (a class)
  - Restricts direct access to some components
  - Protects internal state of an object
  - Hides data
- **Abstraction**: showing only essential features while hiding complicated implementation details
- **Polymorphism**: objects of different classes responding to the same method in their own ways
- **Inheritance**: creating new classes that receive attributes and methods from existing classes

We'll introduce some of these concepts at a practical level in this tutorial and continue to explore them throughout the course.

5.2 Classes in Python: Syntax and Structure

Now that we have some conceptual framework for object-oriented programming, let's look at how to write classes in Python. To define a class, we use the `class` keyword, followed by the name of the class. In the indented section that follows, we can define variables and functions that will be associated with the class and with instances of the class:

```
# Define a class named "Robin".
#
# (Class names look LikeThis rather than like_this.)
class Robin:
    # Class Attributes
    common_name = "American Robin"
    species = "Turdus migratorius"
    beak_shape = "pointy"

    # The special method to initialize instances of this class
    def __init__(self, wingspan, weight, color):
        self.wingspan = wingspan
        self.weight = weight
        self.color = color

    # Define some instance methods that describe the behavior that Robins
    # will have.
```

```
    # Each of these methods will use `id(self)` so that you can see the identity
    # of the object that it is being called on.

    def fly(self, to):
        print(f"robin {id(self)} is flying to {to}!")

    def sing(self):
        print(f"robin {id(self)} is singing!")

    def eat(self, what):
        print(f"robin {id(self)} is eating {what}!")

    def build_nest(self, where):
        print(f"robin {id(self)} is building a nest {where}!")
```

Creating an instance of a class looks a lot like calling a function:

```
robin = Robin(wingspan=35, weight=80, color="gray & reddish brown")
```

You can access the instance attributes using the "dot" syntax:

```
print(
    f"You see a {robin.color} robin "
    f"that weighs {robin.weight} grams "
    f"with a wingspan of {robin.wingspan} centimeters!"
    "\nThat's a nice find!!\n"
)
```

```
You see a gray & reddish brown robin that weighs 80 grams with a wingspan of 35 centimeters!
That's a nice find!!
```

Even though we specified what each of the objects attributes should be when we created it, that doesn't mean we can't change them later if we need to. Let's say our robin eats a worm, and then it gains a little weight afterwards:

```
# Show the robin's current weight
print(f"before eating the worm, the robin weighs {robin.weight} grams")

# The robin eats the worm
robin.eat("a delicious worm")
```

```
# Then it gains 2 grams of weight
robin.weight += 2

# Show the robin's weight again
print(f"after eating the worm, the robin weighs {robin.weight} grams")
```

```
before eating the worm, the robin weighs 80 grams
robin 4360143600 is eating a delicious worm!
after eating the worm, the robin weighs 82 grams
```

We can access the object's behavior by "calling" its methods:

```
robin.sing()
robin.fly(to="Mexico")
robin.eat(what="a worm")
robin.build_nest(where="in a tree")
```

```
robin 4360143600 is singing!
robin 4360143600 is flying to Mexico!
robin 4360143600 is eating a worm!
robin 4360143600 is building a nest in a tree!
```

Multiple distinct instances of the `Robin` class can be created. Check out how each of them has a different ID:

```
robin_1 = Robin(wingspan=35, weight=80, color="gray & reddish brown")
robin_2 = Robin(wingspan=32, weight=78, color="gray & brownish orange")
robin_3 = Robin(wingspan=36, weight=79, color="gray & reddish brown")

print(robin_1)
print(robin_2)
print(robin_3)

robin_1.sing()
robin_2.sing()
robin_3.sing()
```

```
<__main__.Robin object at 0x107857ed0>
<__main__.Robin object at 0x107857d90>
```

```
<__main__.Robin object at 0x103d7eb10>
robin 4421156560 is singing!
robin 4421156240 is singing!
robin 4359449360 is singing!
```

Even though all three of the `Robin` objects were created from the same class, they are distinct entities in our program. If we change something about one of them, it won't change the others:

```
print(
    "before changing weight and color of robin 1,",
    "the weight and color of robin 2 are:",
)
print(robin_2.weight)
print(robin_2.color)


robin_1.weight += 1
robin_1.color = "mostly gray, with some reddish brown"

print(
    "\nafter changing weight and color of robin 1,",
    "the weight and color of robin 2 are:",
)
print(robin_2.weight)
print(robin_2.color)
```

```
before changing weight and color of robin 1, the weight and color of robin 2 are:
78
gray & brownish orange

after changing weight and color of robin 1, the weight and color of robin 2 are:
78
gray & brownish orange
```

Finally, we can even give objects completely new attributes if we want to:

```
robin_1.favorite_food = "french fries"

print(robin_1.favorite_food)
```

```
french fries
```

Be careful with this though. That attribute will *not* be available on all your objects. It will only exist on the specific object where you explicitly added it:

```
try:
    print(robin_2.favorite_food)
except AttributeError as error:
    print(error)
```

```
'Robin' object has no attribute 'favorite_food'
```

Next, let's dig into some more of the details of creating and using classes in Python.

5.3 Initializing Objects

When you instantiate an object from a class in Python, it doesn't simply create an empty shell. Instead, it invokes a special initialization method called `__init__` (if defined). This method serves as the object's constructor, handling the necessary setup to ensure the newly created object is fully functional.

Let's make a new class called `BlueJay` to illustrate some of the details:

```
class BlueJay:
    def __init__(self):
        self.wingspan = 38  # centimeters
        self.color = "blue"
```

In this class, the **init** method takes only one parameter, self, which refers to the newly created object. It assigns the instance attribute color a default value of "blue". Take a look:

```
blue_jay = BlueJay()
print(blue_jay)
print(blue_jay.wingspan)
print(blue_jay.color)

# We can change the value of an instance attribute after the object is created.
blue_jay.wingspan = 35
blue_jay.color = "shockingly blue"

print(blue_jay.wingspan)
print(blue_jay.color)
```

```
<__main__.BlueJay object at 0x103e281a0>
38
blue
35
shockingly blue
```

Creating an object then immediately updating its attributes is such a common operation that Python lets you do it all in one step. This is done by adding additional parameters to the `__init__` function:

```
class BlueJay:
    def __init__(self, wingspan, color):
        self.wingspan = wingspan
        self.color = color


blue_jay = BlueJay(wingspan=36, color="bright blue")
print(blue_jay)
print(blue_jay.wingspan)
print(blue_jay.color)
```

```
<__main__.BlueJay object at 0x10781b230>
36
bright blue
```

Now, we need to call `BlueJay` and provide the `color` and `wingspan` arguments. Failing to do so will result in an error:

```
try:
    BlueJay()
except TypeError as error:
    print(error)
```

```
BlueJay.__init__() missing 2 required positional arguments: 'wingspan' and 'color'
```

The error hints at Python's inner workings. It mentions that two required arguments are missing: `color` and `wingspan`. But hang on, doesn't `__init__` have three parameters?

It does! This reveals how Python handles class instantiation: it first *creates* the object (via `__new__` behind the scenes), then *initializes* it with `__init__`.

When initializing, Python automatically passes the new object as the first argument (typically called `self`, but you could name it anything). You only need to provide the remaining arguments – hence the error about missing two arguments, not three.

Remember that `__init__` works like any other Python function. You can use all the parameter options we covered earlier (see Section 4.3), such as default values for parameters like `color`:

```
class BlueJay:
    def __init__(self, wingspan, color="blue"):
        self.wingspan = wingspan
        self.color = color

blue_jay = BlueJay(wingspan=35)

print(blue_jay.color)
```

```
blue
```

Validating Inputs

The `__init__` method plays a crucial role in validating data when creating class instances. For example, if we need to ensure birds can't have negative wingspans or empty color strings, we can build these checks directly into initialization. When someone creates a bird with invalid data, instead of failing, the code can substitute sensible defaults. This approach guarantees that all instances meet our basic requirements, protecting against bad input:

```
class BlueJay:
    def __init__(self, wingspan, color="blue"):
        if wingspan < 0:
            self.wingspan = 0
        else:
            self.wingspan = wingspan

        if color == "":
            self.color = "blue"
        else:
            self.color = color


blue_jay = BlueJay(-234, "")
print(blue_jay.wingspan)
print(blue_jay.color)
```

```
0
blue
```

If you can't identify a reasonable default value, the most straightforward approach is to simply raise an error. This strategy helps to prevent failures later on.

```
class Bird:
    def __init__(self, species):
        if species == "":
            raise ValueError("species name cannot be blank")


try:
    Bird("")
except ValueError as error:
    print(error)
```

```
species name cannot be blank
```

5.4 Attributes

Attributes are the data that is associated with the class and with instances of that class.

Instance Attributes

When initializing a BlueJay object, the `__init__` function sets up two attributes: `wingspan` and `color`. The `self` parameter refers to the actual instance being created, so `self.wingspan = 1234` creates a wingspan attribute on your new `BlueJay` object.

Instance attributes are not shared between different instances of the same class:

```
tiny_blue_jay = BlueJay(wingspan=28)
big_blue_jay = BlueJay(wingspan=40)

print(tiny_blue_jay.wingspan)
print(big_blue_jay.wingspan)
```

```
28
40
```

If we change one after creation, it will not affect the other:

```
tiny_blue_jay.wingspan += 1

print(tiny_blue_jay.wingspan)
print(big_blue_jay.wingspan)
```

```
29
40
```

Nice! Each object independently manages its state, completely separate from other objects. Just be careful with mutable values in instance attributes though. For example, if you store a bird's colors as a list (since birds can have multiple colors), you might run into some unexpected behavior:

```
colors = ["blue", "white", "black"]

# Create two BlueJay instances with the same color list.
tiny_blue_jay = BlueJay(wingspan=28, color=colors)
big_blue_jay = BlueJay(wingspan=40, color=colors)
print(tiny_blue_jay.color)
print(big_blue_jay.color)

# Can you guess what will happen if we change one of the colors?
tiny_blue_jay.color[0] = "electric blue"
print("\nafter changing color[0] of tiny_blue_jay")
print(tiny_blue_jay.color)
print(big_blue_jay.color)

# Or add a color to one of them?
print("\nafter appending a new color to big_blue_jay")
big_blue_jay.color.append("light blue")
print(tiny_blue_jay.color)
print(big_blue_jay.color)
```

```
['blue', 'white', 'black']
['blue', 'white', 'black']

after changing color[0] of tiny_blue_jay
['electric blue', 'white', 'black']
['electric blue', 'white', 'black']
```

```
after appending a new color to big_blue_jay
['electric blue', 'white', 'black', 'light blue']
['electric blue', 'white', 'black', 'light blue']
```

I know I just explained that instance attributes are independent between objects, which might seem contradictory here. But remember our discussion about mutable parameters back in Section 4.5? Python variables are actually references to objects, not the objects themselves. In the example, both `BlueJay` instances ended up referencing the identical list object. Keep this behavior in mind: it's a common source of subtle bugs.

As mentioned earlier, you can add more instance attributes to an object after creation:

```
blue_jay = BlueJay(wingspan=35)

blue_jay.sneakiness = "very sneaky"

print(f"this blue jay is {blue_jay.sneakiness}!")
```

```
this blue jay is very sneaky!
```

Class Attributes

In our earlier look at the `Robin` class in Section 5.2, we used class attributes for data shared across all instances. This approach is ideal for information common to all robins, things like common name, species, and beak shape. Class attributes make sense when the data belongs to the entire group rather than to specific individuals.

Class attributes are defined directly within the class, but outside any methods:

```
class BlueJay:
    common_name = "Blue Jay"
    species = "Cyanocitta cristata"
    beak_shape = "medium-length, conical"


blue_jay = BlueJay()

print(blue_jay.species)
print(blue_jay.beak_shape)
```

```
Cyanocitta cristata
medium-length, conical
```

You can also access class attributes directly on the class object itself:

```
print(BlueJay.species)
print(BlueJay.beak_shape)
```

```
Cyanocitta cristata
medium-length, conical
```

Aside: that might seem a bit weird, but in Python, classes themselves are also objects that have properties and methods:

```
print(BlueJay.__class__)
```

```
<class 'type'>
```

We will talk more about this later in the course.

Let's update the `BlueJay` class to have both class and instance attributes:

```
class BlueJay:
    # Set class attributes
    common_name = "Blue Jay"
    species = "Cyanocitta cristata"
    beak_shape = "medium-length, conical"

    def __init__(self, wingspan=38, color="blue"):
        # Set instance attributes
        self.wingspan = wingspan
        self.color = color


tiny_blue_jay = BlueJay(wingspan=28)
big_blue_jay = BlueJay(wingspan=40)

# All blue jays will have the same values for the class attributes, but likely
# have different values for the instance attributes.
print(tiny_blue_jay.wingspan, tiny_blue_jay.species)
print(big_blue_jay.wingspan, big_blue_jay.species)
```

```
28 Cyanocitta cristata
40 Cyanocitta cristata
```

Modifying Class Attributes

Class attributes don't have to be constant, unchanging things. Let's look at an example where we change the value of a class attribute from within an instance method to create sequential IDs for instances of that class. Check it out:

```
class Amoeba:
    # This is a class attribute
    counter = 0

    def __init__(self, name=None):
        # We increment the value stored in the counter class attribute by 1.
        Amoeba.counter += 1

        # Then, we set that value to the value of this amoeba instance's `id`
        # attribute.
        self.id = Amoeba.counter

        # If the user doesn't specify a name, then we create a default name
        # that includes the ID.
        if name is None:
            self.name = f"Amoeba_{self.id}"
        else:
            self.name = name


amoeba_1 = Amoeba()
amoeba_2 = Amoeba(name="Bob the Amoeba")
amoeba_3 = Amoeba()

print(amoeba_1.name)
print(amoeba_2.name)
print(amoeba_3.name)
```

```
Amoeba_1
Bob the Amoeba
Amoeba_3
```

Pretty neat! We will go into more fancy details like this in a future tutorial.

A Tricky Example

In our last example, we contained mutations within class methods—a safer approach than external state modification, which often causes bugs. Let's flip this and see what happens when we modify class variables from outside. Warning: it gets a bit confusing!

```
print("tiny_blue_jay species:", tiny_blue_jay.species, id(tiny_blue_jay.species))
print("big_blue_jay species:", big_blue_jay.species, id(big_blue_jay.species))
print()

tiny_blue_jay.species = "i don't know!"

print("tiny_blue_jay species:", tiny_blue_jay.species, id(tiny_blue_jay.species))
print("big_blue_jay species:", big_blue_jay.species, id(big_blue_jay.species))
print()

BlueJay.species = "something else"

print("tiny_blue_jay species:", tiny_blue_jay.species, id(tiny_blue_jay.species))
print("big_blue_jay species", big_blue_jay.species, id(big_blue_jay.species))
print()

another_blue_jay = BlueJay()
print(
    "another_blue_jay species:",
    another_blue_jay.species,
    id(another_blue_jay.species),
)
```

```
tiny_blue_jay species: Cyanocitta cristata 4360567728
big_blue_jay species: Cyanocitta cristata 4360567728

tiny_blue_jay species: i don't know! 4360382448
big_blue_jay species: Cyanocitta cristata 4360567728

tiny_blue_jay species: i don't know! 4360382448
big_blue_jay species something else 4360386224

another_blue_jay species: something else 4360386224
```

Let's break this down:

1. Initially, all `BlueJay` instances share the class attribute `species` with value `"Cyanocitta cristata"` (note the matching IDs in the first two lines).
2. When we set `tiny_blue_jay.species = "i don't know!"`, we're not changing the class attribute, we're creating a new instance attribute that shadows it. The ID changes for `tiny_blue_jay` but stays the same for `big_blue_jay`.
3. With `BlueJay.species = "something else"`, we modify the actual class attribute. This affects all instances that don't have their own shadowing attribute—big_blue_jay sees the new value, but tiny_blue_jay still shows its instance-specific value.
4. Any new instance (like `another_blue_jay`) gets the updated class attribute value.

The apparent complexity stems from Python's attribute lookup sequence:

1. Check the instance namespace first
2. Then check the class namespace
3. Finally, check parent classes

This enables both shared values and individual customization with the same name. This is very flexible, but potentially confusing if you don't understand the lookup mechanism.

5.5 Methods

Methods are functions inside classes that usually work with instances of that class. Class methods exist too (like class attributes), but we'll skip those for now.

Our previous examples were pretty basic. Let's look at something a bit more interesting to showcase why classes and objects are actually useful.

```
class Amoeba:
    """
    Represent an amoeba with position and energy tracking.

    This class models an amoeba that can move around in a 2D space and manage
    its energy levels. Each amoeba has a unique ID, position coordinates, and
    energy value.

    Attributes:
        counter (int): Class variable that keeps track of how many amoeba instances
            have been created.
        id (str): Unique identifier for each amoeba instance.
        position (list): A list of two integers representing [x, y] coordinates.
        energy (int): Current energy level of the amoeba.
    """
```

```
    counter = 0

    def __init__(self, position=None, energy=5):
        Amoeba.counter += 1
        self.id = f"Amoeba #{Amoeba.counter}"

        if position is None:
            self.position = [0, 0]
        else:
            self.position = position

        self.energy = energy

    # This method controls how Amoebas will be printed
    def __str__(self):
        return f"{self.id} -- Position: {self.position}, Energy: {self.energy}"

    def move(self, direction):
        """Move the amoeba, consuming energy."""

        if self.energy <= 0:
            print(f"{self.id} is too weak to move!")
            return

        if direction == "right":
            print(f"{self.id} moves right!")
            # Moving to the right means adding one to the x position
            self.position[0] += 1
        elif direction == "left":
            print(f"{self.id} moves left!")
            # Moving to the left means subtracting one from the x position
            self.position[0] -= 1
        elif direction == "up":
            print(f"{self.id} moves up!")
            # Moving up means adding one to the y position
            self.position[1] += 1
        elif direction == "down":
            print(f"{self.id} moves down!")
            # Moving down means subtracting one from the y position
            self.position[1] -= 1
        else:
            raise ValueError("direction must one of up, down, left, or right")
```

```
        self.energy -= 1

    def eat(self):
        """The amoeba eats, increasing its energy."""
        print(f"{self.id} eats")
        self.energy += 2
```

There are a lot of things to break down about the `Amoeba` class. Let's look at a few key points:

- **Class attributes vs. Instance attributes**:
  - `counter` is a class attribute shared across all instances
  - `id`, `position`, and `energy` are instance attributes unique to each object
- **Constructor Implementation**:
  - Uses `__init__` to set up each new amoeba with its initial state
  - Automatically increments the counter to assign unique IDs
  - Handles default parameters (`position=None, energy=5`)
  - Sets a default position if none is provided
- **String Representation**:
  - Implements `__str__` to provide a human-readable representation
  - Returns formatted string containing the amoeba's ID, position and energy
- **'move() Method Implementation**:
  - `move()` validates input parameters
  - Checks current state before performing actions (`if self.energy <= 0`)
  - Demonstrates internal state modification (changing position and reducing energy)
  - Shows error handling with a descriptive error message (`ValueError`) for bad inputs
- **2D Movement Representation**:
  - Uses a list `[x, y]` to represent position in 2D space
  - Adjusts coordinates based on movement direction
- **State Management**:
  - Class methods track and update the amoeba's internal state (position, energy)
  - Behavior depends on internal state (the amoeba can't move with 0 energy)
- **Code Organization**:
  - Uses docstrings for class and method documentation

- Follows consistent indentation and naming conventions (methods names are short verbs)

Now, let's try it out!

```
import random

# These are the four directions that an amoeba can move
four_directions = ["up", "down", "left", "right"]

# Seed the random generator so that we get the same result each time we run the code
random.seed(37424)

# Generate a random "walk"
directions = random.choices(four_directions, k=10)

# Create a new amoeba instance
amoeba = Amoeba()

# Go through each of the moves one by one
for direction in directions:
    print(amoeba)

    # Each turn, the amoeba has a 1/5 chance in eating some lunch
    if random.random() < 0.2:
        amoeba.eat()

    # Then the amoeba tries to move
    amoeba.move(direction)
    print()
```

```
Amoeba #1 -- Position: [0, 0], Energy: 5
Amoeba #1 eats
Amoeba #1 moves down!

Amoeba #1 -- Position: [0, -1], Energy: 6
Amoeba #1 moves down!

Amoeba #1 -- Position: [0, -2], Energy: 5
Amoeba #1 moves left!

Amoeba #1 -- Position: [-1, -2], Energy: 4
```

```
Amoeba #1 eats
Amoeba #1 moves down!

Amoeba #1 -- Position: [-1, -3], Energy: 5
Amoeba #1 moves up!

Amoeba #1 -- Position: [-1, -2], Energy: 4
Amoeba #1 moves down!

Amoeba #1 -- Position: [-1, -3], Energy: 3
Amoeba #1 moves up!

Amoeba #1 -- Position: [-1, -2], Energy: 2
Amoeba #1 moves left!

Amoeba #1 -- Position: [-2, -2], Energy: 1
Amoeba #1 moves up!

Amoeba #1 -- Position: [-2, -1], Energy: 0
Amoeba #1 is too weak to move!
```

Aside: Refactoring the Amoeba Class

After building our Amoeba class, we can see that some functionality, specifically position tracking and movement, isn't necessarily Amoeba-specific. We're currently using a two-element list for position and updating it within the Amoeba's `move` method. While this works for our small class, let's extract this common behavior into a dedicated `Position` class.

```python
class Position:
    """
    Represents a position in 2D space.

    This class handles tracking and updating a position in a 2D grid system,
    with methods for moving in cardinal directions.

    Attributes:
        x (int): The x-coordinate
        y (int): The y-coordinate
    """

    def __init__(self, x=0, y=0):
        self.x = x
```

```
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"

    def move_right(self):
        """Move one unit to the right (increase x)."""
        self.x += 1

    def move_left(self):
        """Move one unit to the left (decrease x)."""
        self.x -= 1

    def move_up(self):
        """Move one unit up (increase y)."""
        self.y += 1

    def move_down(self):
        """Move one unit down (decrease y)."""
        self.y -= 1
```

Now that we have the `Position` class, we can use it in the `Amoeba` class instead of the original two-element list. In this way, the `Amoeba` class delegates the behavior of position and movement to the `Position` class rather than manage that itself.

```
class Amoeba:
    """
    Represent an amoeba with position and energy tracking.

    This class models an amoeba that can move around in a 2D space and manage
    its energy levels. Each amoeba has a unique ID, position coordinates, and
    energy value.

    Attributes:
        counter (int): Class variable that keeps track of how many amoeba instances
            have been created.
        id (str): Unique identifier for each amoeba instance.
        position (Position): A Position object representing the amoeba's location.
        energy (int): Current energy level of the amoeba.
    """

    counter = 0
```

```
    def __init__(self, position=None, energy=5):
        Amoeba.counter += 1
        self.id = f"Amoeba #{Amoeba.counter}"

        if position is None:
            self.position = Position()
        else:
            self.position = position

        self.energy = energy

    def __str__(self):
        return f"{self.id} -- Position: {self.position}, Energy: {self.energy}"

    def move(self, direction):
        """Move the amoeba, consuming energy."""

        if self.energy <= 0:
            print(f"{self.id} is too weak to move!")
            return

        if direction == "right":
            print(f"{self.id} moves right!")
            self.position.move_right()
        elif direction == "left":
            print(f"{self.id} moves left!")
            self.position.move_left()
        elif direction == "up":
            print(f"{self.id} moves up!")
            self.position.move_up()
        elif direction == "down":
            print(f"{self.id} moves down!")
            self.position.move_down()
        else:
            raise ValueError("direction must one of up, down, left, or right")

        self.energy -= 1

    def eat(self):
        """The amoeba eats, increasing its energy."""
        self.energy += 1
```

There's a new syntax element to note: `self.position.move_right()`. This expression chains multiple "dots" to connect attributes and method calls, something you'll see frequently in Python. You can read it left to right like this:

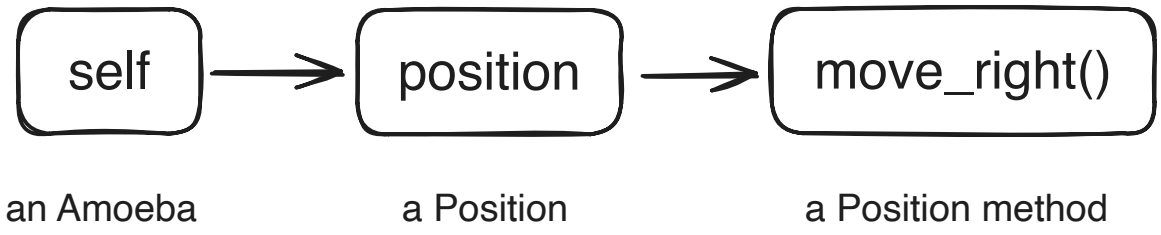


Figure 5.2: A schematic of chained method calls

Extracting the `Position` class from the `Amoeba` class brings a couple of nice benefits:

- Separation of concerns: Each class now has a single responsibility
- Reusability
  - The `Position` class could now be used for other entities that need position tracking.
  - If we need to change how positions work, we only need to change one place rather than in every class that need position tracking.
- Encapsulation: Position management details are hidden inside the `Position` class

This process is called a refactoring, and in this case, we have shown how to spot shared functionality and extract it into a separate class. While this is a good skill to have in your toolkit, don't feel that you have to separate everything. Sometimes it's overkill, especially for one-off scripts or when code reuse is unlikely. Use your judgment!

Special Methods

Special methods (like `__init__` and `__str__`) let your custom classes work with Python's built-in operations. These "magic methods" act as interfaces between your code and core Python functionality. Though they have a cool name, these methods aren't mysterious, rather, they're like standardized hooks that let your classes interact more easily with Python's built-in functions.

For example, when Python performs certain operations, it looks for certain methods:

- When using the `+` operator, Python looks for the `__add__` method
- When using the `len()` function, Python looks for the `__len__` method
- When initializing an object, Python looks for the `__init__` method
- When printing an object, Python looks for the `__str__` method

By implementing these methods, your custom objects can behave like Python's native types. For example, a DNA sequence class with `__add__` could allow sequence concatenation using the simple `+` operator, just like with strings.

In the `Amoeba` and `Position` classes, we implemented `__str__`. The `__str__` method is called whenever Python needs a human-readable string representation of your object, such as when you use the `print()` function.

The `__str__` method should return a concise, informative string that gives the user meaningful information about the object. If you don't define a `__str__` method for your class, Python will use a default representation. Compare the output when printing an `Amoeba` instance versus when printing a `BlueJay` instance:

```
print(Amoeba())
print(BlueJay())
```

```
Amoeba #1 -- Position: (0, 0), Energy: 5
<__main__.BlueJay object at 0x107b92fd0>
```

Since we didn't explicitly define a `__str__` method for the `BlueJay` class, Python prints a default representation of the object, rather than a nice, informative message.

5.6 Duck Typing

The duck test goes something like this:

> If it looks like a duck, swims like a duck, and quacks like a duck, then it probably *is* a duck!

Many programming languages, including Python, use this idea for something we call duck typing. In this context, Python is essentially saying, "Well, I don't know what class this object is, but if it can `swim` or `fly` or `quack`, then I will treat it like a `Duck`." In other words, it only cares whether the object can perform the needed actions. If it can, then Python will happily work with it regardless of what it actually is. This practical approach represents one way Python implements the polymorphism pillar of OOP mentioned earlier.

Let's see an example of this in action:

```
class Duck:
    def quack(self):
        print("quack, QUACK!!")

    def fly(self):
        print("the duck is flying!")

    def swim(self):
        print("the duck is swimming")


class Gull:
    def squawk(self):
        print("squawk!!!")

    def fly(self):
        print("the crow flies!")

    def swim(self):
        print("the gull swims near the beach")


class Whale:
    def swim(self):
        print("the whale is swims underwater")
```

All three of these classes have a method called `swim`. If we have a collection of those objects, we can call the `swim` method on each of them without having to know what type the object is:

```
animals = [Duck(), Duck(), Gull(), Whale()]

for animal in animals:
    animal.swim()
```

```
the duck is swimming
the duck is swimming
the gull swims near the beach
the whale is swims underwater
```

While convenient, you might not always know if your objects have the method you need. There are two common ways to deal with this:

1. Simply call the method and catch any exceptions that occur. (the Python docs call this EAFP style programming – "Easier to ask for forgiveness than permission")
2. Check if the object has the method before attempting to call it. (the Python docs call this LBYL style programming – "look before you leap")

Both approaches have their place, depending on your specific situation and coding style preferences. Let's look at examples of both approaches.

Asking for Forgiveness

In this example, we call the method `fly` on each `animal` and catch any exceptions that occur using `try/except`.

```
animals = [Duck(), Duck(), Gull(), Whale()]

for animal in animals:
    try:
        animal.fly()
    except AttributeError as error:
        print(error)
```

```
the duck is flying!
the duck is flying!
the crow flies!
'Whale' object has no attribute 'fly'
```

Looking Before You Leap

In this example, we use `hasattr` (short for "has attribute") to check if each `animal` has the `fly` attribute before attempting to call the method.

```
animals = [Duck(), Duck(), Gull(), Whale()]

for animal in animals:
    if hasattr(animal, "fly"):
        animal.fly()
    else:
        print(f"{animal} can't fly!")
```

```
the duck is flying!
the duck is flying!
the crow flies!
<__main__.Whale object at 0x103e7c910> can't fly!
```

Special Methods & Duck Typing

Let's return once more to Python's special "magic methods". These are a great example of how duck typing can be useful. If we implement special methods like `__str__` or `__len__` for our custom classes, then Python can treat them like built-ins. Here's an example of a `Gene` class:

```python
class Gene:
    def __init__(self, name, sequence):
        self.name = name
        self.sequence = sequence

    # Special method for string representation
    def __str__(self):
        return f">{self.name}\n{self.sequence}"

    # Special method for length
    def __len__(self):
        return len(self.sequence)


gene = Gene("awesome_gene", "ATGATATCCATCGCTACTAGACTACTACGCGCGGCTCT")

print(len(gene))
print(gene)
```

```
38
>awesome_gene
ATGATATCCATCGCTACTAGACTACTACGCGCGGCTCT
```

In this example, our `__str__` method formats the gene to match FASTA file conventions, while the length method returns the sequence length. These additions help our `Gene` class behave more like native Python objects, seamlessly integrating with the language's expectations.

5.7 Standard Python Classes

In Chapter 2, we introduced built-in Python data structures like lists, dictionaries, and sets. Now we can recognize these as classes that follow the principles we've just learned: they combine data and behavior into objects with methods that help us work with them safely and effectively.

Let's look at a few examples of how built-in Python classes provide methods that operate on their internal data:

```
# List methods
my_list = [1, 2, 3, 4]

# Adds an element to the end
my_list.append(5)

# Reverses the list in-place
my_list.reverse()

print(my_list)

# String methods
text = "  hello world  "

# Strip whitespace
print(text.strip())

# Strip whitespace, then uppercase
print(text.strip().upper())

# Strip whitespace, then uppercase, then replace capital L with lowercase l
print(text.strip().upper().replace("L", "l"))


gene_info = {"name": "some_gene", "chromosome": 10}

# Print the keys
print(gene_info.keys())

# "Merge" the dictionaries
gene_info.update({"function": "tumor suppressor"})

print(gene_info)
```

```
[5, 4, 3, 2, 1]
hello world
HELLO WORLD
HEllO WORlD
dict_keys(['name', 'chromosome'])
{'name': 'some_gene', 'chromosome': 10, 'function': 'tumor suppressor'}
```

Other commonly used built-in classes include:

- `set` – for handling collections of unique items
- `file` objects – returned when opening files with the `open()` function
- `datetime` – for working with dates and times

Each of these classes encapsulates both data (like the items in a list) and behavior (methods like `append` or `sort`), just like the custom classes we wrote in this tutorial.

5.8 Object-Oriented Thinking

Now that you have seen the mechanics of building and using classes in Python, let's get a little meta. When thinking in an object-oriented way, you see your system in terms interacting entities – an approach that fits naturally with how we like to think about biological systems. Here's a quick guide to developing an object-oriented mindset:

- Identify key entities
  - What are your main working pieces?
  - Physical things like cells, proteins, genes, patients?
  - Abstract concepts like Files, Queues, Nodes, Graphs?
- Determine their properties
  - What characteristics define them?
  - Example: cells have size, type, metabolic rate
- Identify their actions
  - What can they do?
  - Cells divide, proteins fold, birds fly, files can be read
- Group related functionality: cell properties and behaviors belong in a Cell class
- Split classes when they become unwieldy or overly complex

Start with these steps as your foundation. As you gain experience, you'll naturally develop a feel for modeling systems that works best for your specific projects.

Balancing Approaches

Python's flexibility lets you mix programming approaches within a single project or script.

Good Python code often combines different paradigms:

- **Object-oriented** for modeling biological entities where data and behavior are linked
- **Functional** for data transformations and analysis
- **Procedural** for simple sequential operations

Let the problem you're tyring to solve guide your programming approach. Sometimes a simple module with functions works better than a class, while other times a well-designed class hierarchy models your domain in the clearest way.

In general, classes work well when you're working with entities that have both data (attributes) and behaviors (methods) that naturally belong together. Consider using classes when:

- You need multiple similar entities with shared structure and behavior (like 1000 cells in a simulation)
- You repeatedly apply the same operations to specific data types
- Your data needs validation (ensuring DNA sequences only contain valid nucleotides)
- Data and actions are tightly coupled
- You need to track state across operations
- Your project needs additional structure
- Your data has natural hierarchies (DNA and Proteins are both Molecules)

Note: We didn't get into class hierarchies in this tutorial. That's an advanced topic for a later time.

Consider simpler alternatives when:

- Writing straightforward data processing scripts
- Dealing with mostly static data
- Using functions that transform data without maintaining state
- Working with data where built-in Python structures are sufficient
- Your project is small and classes would add unnecessary complexity

Many popular data science libraries like pandas and seaborn emphasize data transformations and function chaining rather than object-oriented approaches. When using these libraries, follow their patterns instead of forcing objects where they don't make sense.

The goal is clear, maintainable code that you and colleagues understand. Start simple, adding complexity only when needed. With experience, you'll develop intuition for when classes are the right tool.

General Tips and Pitfalls

Let's wrap up with some broad best practices and common pitfalls for working with classes. We'll refine these as you tackle more complex programs throughout the course.

General tips:

- Follow naming conventions
  - PascalCase for classes (`BacterialCell`), snake_case for methods/attributes (`growth_rate`)
- Keep classes focused
  - One clear purpose per class
  - Avoid mixing responsibilities
- Design intuitive interfaces
  - Consider how others will interact with your classes
  - Make correct usage easy, incorrect usage difficult
- Document with docstrings
- Validate inputs in `__init__`
- Implement magic methods when appropriate

General pitfalls:

- Modeling excessive behavior
  - Include only what's relevant to your specific application
  - A bird has many attributes, but you only need what serves your research question
- Overloaded classes
  - A `Cell` class shouldn't also process sequencing data and generate plots
- Underutilized classes
  - Data-only classes might work better as dictionaries
  - Method-only classes might work better as function modules
- Poor naming choices
  - `class Gene` communicates purpose better than `class X`

5.9 Wrap-Up

Python classes create custom data types that bundle related data and functions together, helping organize code by grouping attributes and methods. In scientific programming, they're great for representing complex entities with intertwined data and behavior. Use classes when they make your code more organized and readable – with experience, you'll develop instincts for when to use them versus simpler approaches. Keep exploring, stay curious, and experiment with different solutions to your scientific problems!

5.10 Suggested Readings

You might enjoy checking out some of these resources:

- Python Docs: Classes
- Real Python
  - Object-Oriented Programming (OOP) in Python
  - Python Class Constructors: Control Your Object Instantiation
  - Class and Instance Attributes
  - Duck Typing in Python: Writing Flexible and Decoupled Code
- Wikipedia's Object-oriented programming