



CONTENTS

1	Concrete, Metals, and Fertilizer	5
1.1	Fertilizer	5
1.1.1	The Nitrogen Cycle	5
1.1.2	The Haber-Bosch Process	6
1.1.3	Other nutrients	7
1.2	Concrete	7
1.2.1	Steel reinforced concrete	7
1.2.2	Recycling concrete	8
1.3	Metals	8
1.3.1	Steel	8
1.3.2	What metal for what task?	10
2	Advanced Python Syntax	11
2.1	Where did we leave off?	11
2.2	Functions	11
2.2.1	Defining and Calling a Function	11
2.2.2	Parameters and Arguments	12
2.2.3	Return Values	12
2.2.4	Scope	14
2.2.5	Summary	14
2.3	Classes and More Basic Object-Oriented Programming	15
2.3.1	Defining a Class	15
2.3.2	Creating Objects	16
2.3.3	The <code>__init__</code> Method	16

2.3.4	Accessing Object Attributes	16
2.3.5	Methods	17
2.3.6	Printing Objects	17
2.3.7	Why Use Classes?	18
2.3.8	Summary	19
2.4	Errors	20
2.4.1	Syntax Errors	20
2.4.2	Exceptions	21
2.4.3	Try-Except	22
2.5	Recursive Functions	23
2.5.1	Binary Search and Divide-and-Conquer Algorithms	24
3	Making Plots with matplotlib	27
3.1	Libraries	27
3.2	Installing and Importing Matplotlib	27
3.3	A Simple Line Plot	28
3.4	Bar Plots	30
3.5	Several Lines on One Plot	30
3.6	Scatter Plots	31
3.6.1	Plotting Data from a File	32
3.7	Histograms	33
3.8	The Object-Oriented Approach	35
3.9	Simulating a Falling Hammer	37
3.9.1	Graphing the Simulation	39
4	Sorting and Big-O Notation	43
4.1	Why Sorting Matters	43
4.2	Selection Sort	43
4.3	Insertion Sort	44
4.4	Merge Sort	44
4.5	Big-O Notation	45
4.6	Comparing the Sorts	45
5	Testing and Debugging	49
5.1	Why Test?	49
5.2	Print Debugging	49
5.3	Assertions	49
5.4	Unit Tests	50
5.5	Reading a Failing Test	50

	3
5.6 A Debugging Routine	51
6 Version Control with Git	53
6.1 Why Use Version Control?	53
6.2 Creating a Repository	53
6.3 Saving a Snapshot: Commits	53
6.4 Branches	54
6.5 A Routine for Each Session	54
A Answers to Exercises	57
Index	63

Concrete, Metals, and Fertilizer

1.1 Fertilizer

How do you feed 8 billion people? In 1950, there were 2.5 billion people on Earth, and hunger was common. Today there are more than three times as many of us, and a much smaller share go hungry, although malnutrition has not disappeared. How did farms grow so much more food? There were several factors: better crop varieties, reliable irrigation, increased mechanization, and affordable fertilizers.

When a plant grows, it takes molecules out of the soil and uses them to build proteins. It primarily needs the elements nitrogen (N), phosphorus (P), and potassium (K).

When you buy a bag of fertilizer at the store, it typically has three numbers on the front. For example, you might buy a bag of “24-22-4”. This means that 24% of the mass of the bag is nitrogen. The other two numbers give phosphate (P_2O_5) and potash (K_2O), which are the forms that fertilizer labels use. The phosphorus and potassium in the bag are smaller amounts than these numbers suggest.

Potassium comes as potassium carbonate (K_2CO_3), potassium chloride (KCl), potassium sulfate (K_2SO_4), and potassium nitrate (KNO_3). Any blend of these chemicals is known as “potash”. Potash is dug up out of mines.

Phosphorus is also mined, but is refined into phosphoric acid (H_3PO_4) before it is put into fertilizer.

Nitrogen is an especially interesting case for 2 reasons:

- Worldwide, farmers apply more nitrogen to their soil than potassium or phosphorous combined.
- 78% of the air we breathe is nitrogen in the form of N_2 , but neither plants nor animals can utilize nitrogen in that form.

1.1.1 The Nitrogen Cycle

Converting the N_2 in the air into a form that a plant can use is known as *nitrogen fixation*. For billions of years, there were only two ways that nitrogen fixation occurred on earth:

- **Lightning.** The energy of a lightning bolt splits N_2 and O_2 in the air, which recombine into nitrogen oxides. Rain washes these into the soil as nitrate (NO_3^-). This is roughly 10% of natural nitrogen fixation.
- **Bacteria.** Certain bacteria do the rest, converting N_2 into ammonia (NH_3).

Let us say that you are eating soybeans. Bacteria called *rhizobia* have a symbiotic relationship with soybean plants. Rhizobia fixes nitrogen for the soybean plant. The soybean plant performs photosynthesis, then gives sugars to the rhizobia.

The proteins in the soybeans contain nitrogen from the rhizobia. When you eat them, you use some of the nitrogen to build new proteins. You probably do not use all the nitrogen, so your cells release ammonia into your blood.

Ammonia likes to react with things, so your liver combines the ammonia with carbon dioxide to make urea ($\text{CO}(\text{NH}_2)_2$). Your kidneys take the urea out of your blood and mix it with a bunch of water and salts in your bladder. When you urinate, the urea leaves your body.

If you urinate on the ground, the nearby plants can take the nitrogen out of the urea.

When you die, the nitrogen in your proteins will return to the soil as ammonia and nitrate.

For centuries, farms got their nitrogen from urine, feces, and rotting organic material. There were two challenges with this:

- Human pathogens had to be kept away from human food.
- There was simply not enough of it to feed billions of people.

This meant we had to figure out how to do nitrogen fixation at an industrial level.

1.1.2 The Haber-Bosch Process

During World War I, two German scientists, Fritz Haber and Carl Bosch, figured out how to make ammonia from N_2 and H_2 using high temperatures and pressures. This is how nearly all nitrogen fertilizer is created today.

Where do we get the H_2 ? From methane (CH_4) in natural gas. Today, 3-5% of the world's natural gas production is consumed in the Haber-Bosch process.

The ammonia is converted into ammonium nitrate (NH_4NO_3) or urea before it is shipped to farms.

1.1.3 Other nutrients

Healthy plants require several other elements that are sometimes applied as fertilizer: calcium, magnesium, and sulfur.

Finally, tiny amounts of copper, iron, manganese, molybdenum, zinc, and boron are sometimes needed.

1.2 Concrete

To make concrete, you mix cement with water and an aggregate (sand or rock). The cement is usually only about 10 to 15 percent of the mixture. The cement reacts with the water, and the resulting solid binds the aggregate together. In 2019, the world consumed 4.5 billion tons of cement.

Concrete is hard and durable. The Romans built with it 2,000 years ago, and the concrete dome of the Pantheon in Rome is still standing. Today, we use concrete to build many structures including buildings, bridges, airport runways, and dams. Mortar and grout are close cousins of concrete, used to glue bricks together and to fill gaps.

There are many kinds of cement, but the most common is Portland cement. It is made by heating limestone (calcium carbonate) with clay (for silicon) in a kiln. Two things come out of the kiln: Carbon dioxide and a hard substance called “clinker”. The clinker is ground up with some gypsum before it is sent to market.

Breaking limestone apart takes a lot of heat (the reaction is endothermic), and the carbon dioxide goes into the atmosphere. Cement manufacturing is responsible for about 8% of the world’s CO₂ emissions; it is a major contributor to climate change.

Especially hard concrete, like that used in a nuclear power plant, can support very large loads without being crushed. However, if you pull on two ends of a piece of concrete, it comes apart relatively easily. We say that concrete can handle a lot of *compressive stress*, but not much *tensile stress*.

1.2.1 Steel reinforced concrete

Many places where we use concrete (like in a bridge), we need both compressive and tensile stress. Often, the top of a beam is undergoing compression and the bottom of the beam is undergoing tension.

Steel has tremendous tensile strength, but it is far more expensive than concrete. So we get the best of both by burying steel bars or cables inside the concrete: the concrete carries the

squeezing, and the steel carries the pulling. This is known as *steel-reinforced concrete*. The concrete generally does a very good job protecting the steel, which keeps it from rusting.

You may have heard of *rebar* before. That is just short for “reinforcing bar”. Typically, rebar has bumps and ridges that keep the bar and the concrete from moving independently.

1.2.2 Recycling concrete

Many concrete structures only last about 100 years. When they are demolished, the concrete can be reused as aggregate in other projects. Often, the concrete bits are mixed with cement and made into concrete once more.

If the concrete to be reused is reinforced with steel, the steel has to be removed and recycled separately. The concrete is then crushed into small pieces.

1.3 Metals

Elements that transmit electricity well, even at low temperatures, are called *metals*. Many metals are likely familiar to you, such as aluminum, iron, copper, tin, gold, silver, and platinum. Aluminum and iron are particularly common; together they make up about 14% of the earth’s crust.

An *alloy* is a mixture of elements that includes at least one metal. Brass, for example, is an alloy of copper and zinc. Bronze is an alloy of copper and tin.

1.3.1 Steel

One of the most common alloys is steel, which is an alloy of iron and carbon. In pure iron, layers of atoms slip past each other easily, so pure iron is relatively soft and easily deformed. The carbon in steel prevents that slipping, which is why steel is much, much harder than iron.

How much carbon does steel have? If you have less than 0.002% by weight, you end up with something very much like pure iron. As you increase the carbon, it gets harder and harder. Once it gets above about 2%, the result is very brittle.

If you add about 11% chromium to steel, you get *stainless steel*, which resists rusting.

Worked example: stress in a steel cable. A steel cable with a cross section of 2 mm^2

carries a load of 1400 N. The stress is the force divided by the area:

$$\sigma = \frac{F}{A} = \frac{1400 \text{ N}}{2 \times 10^{-6} \text{ m}^2} = 7.0 \times 10^8 \text{ Pa} = 700 \text{ MPa}$$

This is the tensile strength of the steel listed below, so the cable is at its limit.

Exercise 1 Tensile Strength

Working Space

The tensile strength of steel is usually between 400 MPa and 1200 MPa. A Mega Pascal (MPa) is the strength necessary to hold 1,000,000 newtons of force with a cable that has a 1 square meter cross section. Or, equivalently, to hold 1 newton of force with a cable that has a 1 square millimeter cross section.

You are buying a round cable that has a tensile strength of 700 MPa, and it must hold a 100 kg person aloft. What is the diameter of the smallest cable you can use?

Answer on Page 57

Exercise 2 Holding Capacity

Working Space

A steel rod with a cross section of 4 mm² is loaded to its tensile strength of 700 MPa. What is the largest force it can carry?

Answer on Page 57

Here are some approximate tensile strengths of other materials:

Material	Tensile strength (MPa)
Cast iron	200
Concrete	4
Rubber	16
Glass	33
Wood	40
Nylon	100
Human hair	200
Aluminum	300
Steel	700
Spider webs	1000
Carbon fiber	4000

1.3.2 What metal for what task?

Copper is often used for electrical wires in your house and appliances. This is because it is very efficient at moving electricity (very little power is lost as heat). It is also very good at transmitting heat, so you will often see copper pots and pans.

Aluminum is less dense than copper, and is still a relatively good conductor of electricity. This combination of lighter weight and conductivity is why the overhead wires in a power system are often made of aluminum.

Aluminum is not as strong as steel, but considerably lighter. It is often used where weight matters, such as in airplanes, cars, boats, and soda cans.

Titanium is about as strong as steel, but it weighs about half as much. Titanium is very difficult to work with, so it is used in places where weight and strength are very important and cost is not, such as in airplanes and bicycles.

(Carbon fiber, which is light, strong, and very easy to work with, is replacing aluminum and titanium in many applications. 20 years ago, many expensive bicycles were made of titanium. These days the vast majority are made with carbon fiber.)

Zinc and tin are very resistant to corrosion, so they are often used as a coating to prevent steel from rusting. They are also used in many alloys for the same reason. In the United States, the penny is 97.5% zinc and only 2.5% copper.

Advanced Python Syntax

2.1 Where did we leave off?

In the Introduction to Python chapter, you installed Python and VS Code, ran programs from the console, and learned that Python runs code **sequentially**, one line at a time. You printed output, stored data in variables of different types, did arithmetic, read input, made decisions with conditionals, repeated work with loops, and handled strings and lists. That is enough to write real programs.

Now we add the tools that let programs grow without becoming a mess: functions, classes, error handling, and recursion.

2.2 Functions

As our programs get larger, we begin to repeat the same sets of steps again and again. Rather than copying and pasting code (which is hard to maintain and easy to break), we can group instructions into *functions*. A *function* is a named block of code that performs a task. Once a function is defined, it can be *called* (used) as many times as needed.

You have already used functions before, even if you did not know it! For example, `print()`, `input()`, `len()`, and `type()` are built-in Python functions.

2.2.1 Defining and Calling a Function

To define a function in Python, we use the `def` keyword, followed by:

- a function name
- parentheses ()
- a colon :
- an indented block of code

```
1 def say_hello():  
2     print("Hello!")  
3     print("Welcome to Python.")
```

This code *defines* the function, but it does not run the code inside it yet. To execute the function, we must *call* it by using its name followed by parentheses:

```
1 say_hello()
```

When Python reaches a function call, it temporarily jumps into the function, runs the indented code, and then returns to the line after the call. Just like conditionals and loops, **indentation matters**.

2.2.2 Parameters and Arguments

Most functions are more useful when they can work with different values. A *parameter* is a variable name listed inside the parentheses of a function definition. An *argument* is the actual value you pass into the function when you call it.

```
1 def greet(name):  
2     print("Hello", name)
```

Here, `name` is a parameter. We can call the function with different arguments:

```
1 greet("Alex")  
2 greet("Chicago")
```

2.2.3 Return Values

Worked example: a function that returns a value.

```
1 def rectangle_area(width, height):  
2     return width * height  
3  
4 area = rectangle_area(3, 4)  
5 print(area)  
6 print(rectangle_area(2.5, 2))
```

The first call returns 12, and the second returns 5.0. The output is 12 and then 5.0. The result is a number that the caller can store or print. Because 2.5 is a float, the product is a float too.

Some functions *return* a value. Returning a value allows the function to produce a result that can be stored in a variable or used in an expression.

```
1 def add(a, b):  
2     return a + b
```

Now we can use the returned value:

```
1 x = add(3, 4)  
2 print(x)
```

Output:

```
1 7
```

We can alter the function to expect a certain parameter type, such as integers, and return a specific type as well:

```
1 def add(a: int, b: int) -> int:  
2     return a + b
```

It is important to note that Python will not raise a `TypeError` if you pass in the wrong type, but this is a helpful way to document your code for other programmers, which you may often work with, and yourself.

We can force the parameters to be a certain type, and raise an error if the wrong type is passed in:

```
1 def add(a: int, b: int) -> int:  
2     if not isinstance(a, int) or not isinstance(b, int):  
3         raise TypeError("Both arguments must be integers")  
4     return a + b
```

Important: `print()` displays information to the console, but it does not return a useful value. `return` sends a value back to where the function was called. Functions that do not have a `return` statement return `None` by default, and are often called *void* functions.

2.2.4 Scope

A variable created inside a function only exists inside that function. This idea is called *scope*. If you define a variable inside a function, you cannot use it outside of the function unless you return it.

```
1 def make_number():
2     x = 10
3     return x
4
5 y = make_number()
6 print(y)
```

The variable `x` exists only inside `make_number()`, but the value it returns is stored in `y` outside the function.

2.2.5 Summary

- Functions group code into reusable blocks
- Functions are defined with `def` and called using parentheses
- Parameters receive input values (arguments) when a function is called
- `return` sends a value back to the caller
- Variables inside a function have local scope

Exercise 3 Tracing a Function Call

Consider the following code:

Working Space

```
1 def double(x):
2     return x * 2
3
4 a = 3
5 b = double(a)
6 print(b)
```

What is printed? What is the value of `a`

after the program runs?

Answer on Page 57

Exercise 4 Write a Function

Write a function called `is_even(n)` that returns `True` if `n` is even and `False` otherwise. Then show an example call to your function using `n = 7`.

Working Space

Answer on Page 57

2.3 Classes and More Basic Object-Oriented Programming

So far, we have worked mostly with individual variables, lists, and functions. As programs grow larger, it becomes useful to group related data and behavior together. *Object-Oriented Programming* (OOP) is a programming style that organizes code around *objects* rather than individual functions.

In Python, objects are created from *classes*. A *class* is a blueprint for creating objects that contain both data (variables) and behavior (functions).

2.3.1 Defining a Class

A class is defined using the `class` keyword, followed by:

- the class name (by convention, written in CamelCase)
- a colon :
- an indented block of code

```
1 class Person:
2     pass
```

This defines an empty class. It does not do anything yet, but it gives Python a new type called `Person`.

2.3.2 Creating Objects

An *object* is an instance of a class. Objects are created by calling the class name like a function.

```
1 p1 = Person()  
2 p2 = Person()
```

Here, `p1` and `p2` are two separate objects, both created from the same class.

2.3.3 The `__init__` Method

Most classes need to store data. This is done using a special function called `__init__`. This function runs automatically when a new object is created.

```
1 class Person:  
2     def __init__(self, name, age):  
3         self.name = name  
4         self.age = age
```

The parameter `self` refers to the current object being created. Each object gets its own copy of the variables defined using `self`.

```
1 p = Person("Alex", 20)
```

Now the object `p` has two attributes:

- `p.name`
- `p.age`

2.3.4 Accessing Object Attributes

Attributes are accessed using dot notation.

```
1 print(p.name)
2 print(p.age)
```

Output:

```
1 Alex
2 20
```

Each object stores its own values. Creating another object does not overwrite existing ones.

2.3.5 Methods

Functions defined inside a class are called *methods*. Methods describe behavior that belongs to the object.

```
1 class Person:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6     def greet(self):
7         print("Hello, my name is", self.name)
```

Calling a method uses dot notation:

```
1 p = Person("Alex", 20)
2 p.greet()
```

Output:

```
1 Hello, my name is Alex
```

2.3.6 Printing Objects

By default, printing an object produces an unreadable result:

```
1 print(p)
```

Output (example):

```
1 <__main__.Person object at 0x7f9c1a3d>
```

That long number is where the object lives in the computer's memory, which is not much help to a human. To control how an object is printed, define the special method `__str__`.

```
1 class Person:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6     def __str__(self):
7         return f"{self.name} ({self.age} years old)"
```

Now printing the object gives a meaningful result:

```
1 p = Person("Alex", 20)
2 print(p)
```

Output:

```
1 Alex (20 years old)
```

2.3.7 Why Use Classes?

Classes allow us to:

- group related data and behavior together
- create many objects with the same structure
- write code that is easier to understand and maintain

2.3.8 Summary

- A class is a blueprint for creating objects
- Objects store data using attributes
- `__init__` initializes new objects
- Methods define behavior for objects
- `__str__` controls how objects are printed
- This is only the foundation of OOP; you will meet more advanced ideas, such as inheritance, later

Exercise 5 Tracing an Object

Consider the following code:

```
1 class Counter:
2     def __init__(self, value):
3         self.value = value
4
5     def increment(self):
6         self.value += 1
7
8 c = Counter(5)
9 c.increment()
10 c.increment()
11 print(c.value)
```

Working Space

What is printed?

Answer on Page 58

Exercise 6 Custom Printing

Write a class called `Book` that stores a title and an author. Add a `__str__` method so that printing a book displays:

```
1 Title by Author
```

Working Space

2.4 Errors

Everyone writes buggy code, even experts. When something goes wrong, Python stops and prints an *error* message that names the problem and the line where it happened. Learning to read these messages is the fastest way to *debug*, or fix, your programs.

You will encounter two main types of errors in your code: **Syntax Errors** and **Runtime Errors/Exceptions**.

- *Syntax errors* happen when your code breaks Python's grammar rules, or *syntax*. Python catches them before the program even starts running.
- Runtime Errors, or *Exceptions*, occur when operations in your code cause an invalid calculation, or attempt to do an invalid action. These can range anywhere from basic misnamed variables to imported libraries crashing from incorrect data.

2.4.1 Syntax Errors

What do you notice is immediately wrong with this code?

```
1 x = "Welcome Home"
2 print(type(x))
```

If we run it, we get the output:

```
1 Traceback (most recent call last):
2   File "<string>", line 1, in <module>
3   File "/usr/lib/python3.12/py_compile.py", line 150, in compile
4     raise py_exc
5 py_compile.PyCompileError:   File "./prog.py", line 2
6     print(type(x)
7         ^
8 SyntaxError: '(' was never closed
```

Every opening parenthesis, bracket, or brace needs a matching closing one. Forget one, and you get a syntax error.

Another type of Syntax Error can be found in incorrect indentation. Take for example the following code:

```
1 if True:
2 print("Hello!")
```

Output:

```
1 Traceback (most recent call last):
2   File "<string>", line 1, in <module>
3   File "/usr/lib/python3.12/py_compile.py", line 150, in compile
4     raise py_exc
5 py_compile.PyCompileError: Sorry: IndentationError: expected an indented block
   ↳ after 'if' statement on line 1 (prog.py, line 2)
```

Here, there is no indentation (usually obtained by pressing the Tab key on your keyboard) for lines under the conditional statement. This causes a Syntax Error to be raised.

2.4.2 Exceptions

Let us say I try to run the following code:

```
1 print(x)
2 x = "hello"
```

You may see the following output:

```
1 Traceback (most recent call last):
2   File "./program.py", line 1, in <module>
3     NameError: name 'x' is not defined
```

You have encountered a *NameError*, because *x* was not assigned before it was attempted to be printed. This brings up an important note on Python code: **code is executed line-by-line, sequentially**. So even if you define *x* after you try and print it, Python will not understand what you are trying to print. Let us look at another example:

2.4.3 Try-Except

```
1 try:
2     x = int(input("Enter a number: "))
3     print(10 / x)
4 except ValueError:
5     print("Please enter a valid integer.")
6 except ZeroDivisionError:
7     print("Cannot divide by zero.")
```

Here, we attempt to divide 10 by some input number x . There are two Exceptions that could cause this to fail:

- The user inputs a string of characters, or anything that is not an integer
- The user inputs 0, an invalid divisor.

We use a Try-Except block to check for different errors generated, similar to an if statement. The try block surrounds a block of code that may cause faulty runtime output or errors.

Exercise 7 Fix the Code

Working Space

```
1 items = ["pen", "book", "eraser", "ruler"]
2
3 index = input("Enter an index (0 to 3): ")
4 print("You chose:", items[index])
5
6 items.remove("pencil")
7 print("Updated list:", items)
```

Name two errors that could occur when this code is run. For each error, explain why it occurs and how to fix it.

Answer on Page 58

2.5 Recursive Functions

As we mentioned in the proofs chapter, *recursion* is when a function calls itself. Picture two mirrors facing each other: each reflection contains a smaller reflection, which contains a smaller one, and so on.

A *recursive function* can be as simple as this

```

1 def foo():
2     # recursive call
3     foo()
4
5 foo()
```

The line that calls `foo()` within its self-definition is called a *recursive call*. However, doing this may cause you to get an error as follows

```

1 RecursionError: maximum recursion depth exceeded
```

What does this mean? Well, we did not create a way to get out of our recursive call once we start running. The function would call itself forever, so Python gives up after about 1,000 nested calls and raises a **RecursionError**.¹ To prevent this, our recursive functions need to have an exit condition, called a *base case* or *base condition*. A base case is defined as a terminating condition that does not use a recursive call to provide an answer. This allows you to exit the function at some time, instead of continually calling a function forever. Let us look at the famous Fibonacci sequence.

The fibonacci sequence is a sequence of numbers where each term F_n is defined by adding the previous two terms:

$$\text{fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & n \geq 2 \text{ and } n \in \mathbb{Z} \end{cases}$$

In sequence notation we write the base cases as:

$$F_0 = 0, \quad F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

¹In Java, you may get a `StackOverflowError`. In C or C++, you may get a message saying `Segmentation fault (core dumped)`. In programming languages much closer to memory, we say that the call stack reaches a maximum, or is *overflowed*. The call stack is one which is governed by the amount of RAM or random access memory. That is where the website [Stack Overflow](#) gets its name!

If we did not have a base case, how would we start defining the terms? Simply, we could not. There would be no way to define recursively-called previous terms explicitly as we would not have a definite value for them.

2.5.1 Binary Search and Divide-and-Conquer Algorithms

If you look up information about recursive functions, you will find their most common use: *divide-and-conquer algorithms*. Divide-and-conquer algorithms are those that are run recursively to reduce problems into smaller versions of the same problem. Trees, graphs, and sorted arrays are natural fits, because each can be split into smaller pieces of the same kind.

A very common example of this is searching for a word in a dictionary. Let us say you are looking for the word *keyboardist*. For example purposes, let us say this specific dictionary is 2048 pages long (this will help our math). One way to find the word *keyboardist* would be to search every individual page. This is called a *linear search*. This would not take too long if we were searching for the word *apple*, but what about *zebra*? We would be searching almost all 2048 pages before reaching the word!

There is a much quicker and more efficient approach. Say we flip to approximately middle of the dictionary, page 1024. The first word is *muffin*. The word *keyboardist* must be to left of this of this page, since we know the dictionary is alphabetically sorted and *k* comes before *m*. Let us split the 1024 remaining pages in half; we flip to page 512. The first word in page 512 is *committee*. We need to go back towards the original middle, since *c* comes before *k*. So now we know *keyboardist* must lie somewhere between pages 512 and 1024.

We split that interval in half again and flip to page 768. Suppose the first word there is *gracious*. Since *g* comes before *k*, we know our word must be farther to the right, so we now restrict our search to pages 768 through 1024. We continue in this way, each time cutting the remaining search region in half and deciding whether to move left or right depending on alphabetical order.

Eventually, we get to *keyboardist* on page 980. This method is much faster because every comparison eliminates about half of the remaining pages. Instead of checking pages one by one, we narrow the possibilities dramatically with each step. This process is a *divide-and-conquer* approach called *binary search*. Notice that at each step, we are not actually doing one static action. We are reducing the possible page entries by half, and then recalling the same comparison on the function repeatedly until we reach our target. This is what makes our function a divide-and-conquer recursive algorithm.

Let us look at what a binary search for a list looks like in python:

```
1 def binary_search(arr, target, left, right):  
2     if left > right:
```

```

3         return -1 # not found
4
5     mid = (left + right) // 2
6
7     if arr[mid] == target:
8         return mid
9     elif target < arr[mid]:
10        return binary_search(arr, target, left, mid - 1)
11    else:
12        return binary_search(arr, target, mid + 1, right)

```

Let us prove this algorithm is correct, just as we talked about in the proofs chapter. Since this algorithm is recursive and it functions by operating on smaller versions of itself, the best proof to use here is a *proof by induction*.

Theorem 1. *For any indices left and right, `binary_search(arr, target, left, right)` returns an index i with $\text{left} \leq i \leq \text{right}$ and $\text{arr}[i] = \text{target}$ if the target occurs within $\text{arr}[\text{left} \dots \text{right}]$, or returns -1 if the target is not found within the array.*

Proof. We prove the statement by induction on $n = \text{right} - \text{left} + 1$, the length of the interval to be searched.

Basis Step. If $n \leq 0$, then $\text{left} > \text{right}$. The function immediately returns -1 via line 3, which is correct. An empty interval contains no elements, so the target is not present.

Inductive Hypothesis. Assume the function works correctly for all intervals of length less than n . (Notice we are reducing the problem size).

Inductive Step. Now consider an interval of length $n > 0$, so $\text{left} \leq \text{right}$. The function computes $\text{mid} = \left\lfloor \frac{\text{left} + \text{right}}{2} \right\rfloor$. There are three cases:

- $\text{arr}[\text{mid}] == \text{target}$: If the target we are searching for is at the middle of the new computed index, we correctly return that index on line 8.
- $\text{target} < \text{arr}[\text{mid}]$: Because the array is sorted, every element at positions $\text{mid}, \text{mid} + 1, \dots, \text{right}$ is *at least* $\text{arr}[\text{mid}]$, so all of them are greater than the target. Therefore, if the target exists at all in $\text{arr}[\text{left} \dots \text{right}]$, it must lie in $\text{arr}[\text{left} \dots \text{mid}-1]$. The function then calls `binary_search(arr, target, left, mid - 1)`. The new interval has length strictly smaller than n , so by the inductive hypothesis, that recursive call gives the correct answer. Hence the original call is also correct.
- $\text{target} > \text{arr}[\text{mid}]$: Again, because the array is sorted, every element at positions $\text{left}, \text{left} + 1, \dots, \text{mid}$ is *at most* $\text{arr}[\text{mid}]$, so all of them are less than the target. Therefore, if the

target exists at all in `arr[left...right]`, it must lie in `arr[mid+1...right]`. The function then calls `binary_search(arr, target, mid + 1, right)`. The new interval, again, has length strictly smaller than n , so by the inductive hypothesis, that recursive call gives the correct answer. Therefore, the original call is also correct.

Conclusion. The proof covers every case, and each recursive call searches a strictly shorter interval. By induction on the interval length n , the function is correct for every valid call. $\square$

Exercise 8 Recursive Factorial

Working Space

Write a pythonic function that recursively finds the factorial of a given number n . Assume $n \geq 0$.

Answer on Page 58

Making Plots with matplotlib

A table of a thousand numbers tells you almost nothing at a glance; a picture of them can tell you everything. You have graphed functions in algebra class by plotting points on a grid. Matplotlib, the most widely used Python plotting library, does the same job for you: it draws the grid, the points, the lines, and the labels. We will use it throughout the series to visualize data, check formulas, and watch simulations unfold.

Matplotlib is a *library*, so we start with what that means.

3.1 Libraries

A *library* is a collection of pre-written code that adds features to Python, so you do not have to write everything from scratch. Python comes with many built-in libraries for common tasks such as math, reading files, and working with dates, and thousands more can be installed. Before you write something big, check whether someone has already written it: there is a free, open-source Python library for almost everything.

To use a library, you must first import it. The simplest way is the `import` keyword:

```
1 import math
2 print(math.sqrt(16))
```

The `math` library provides functions such as square roots. Notice that you write the library's name, a dot, and then the function.

You can also import a specific value or function, and then use it without the library's name:

```
1 from math import pi
2 print(pi)
```

3.2 Installing and Importing Matplotlib

Matplotlib does not come with Python. If you are working on your own computer, install it from the command line:

```
1 pip install matplotlib
```

In most programs, you will import its plotting interface, `pyplot`, and give it the short name `plt`:

```
1 import matplotlib.pyplot as plt
```

3.3 A Simple Line Plot

A line plot is ideal for showing how a value changes over time or across an ordered variable. Create a file called `line.py`:

```
1 import matplotlib.pyplot as plt
2
3 # two lists containing values
4 x = [0, 1, 2, 3, 4, 5]
5 y = [0, 1, 4, 9, 16, 25]
6
7 plt.plot(x, y)                # draw y versus x
8 plt.title("A Simple Line Plot")
9 plt.xlabel("x")
10 plt.ylabel("y = x^2")
11 plt.show()                   # open a window with the plot
```

Here, `x` and `y` are the coordinates of the points, and `plt.plot` connects them with lines. The title and axis labels tell a reader what they are looking at; a plot without them is just a squiggle. Running `python3 line.py` opens an interactive window with the plot:

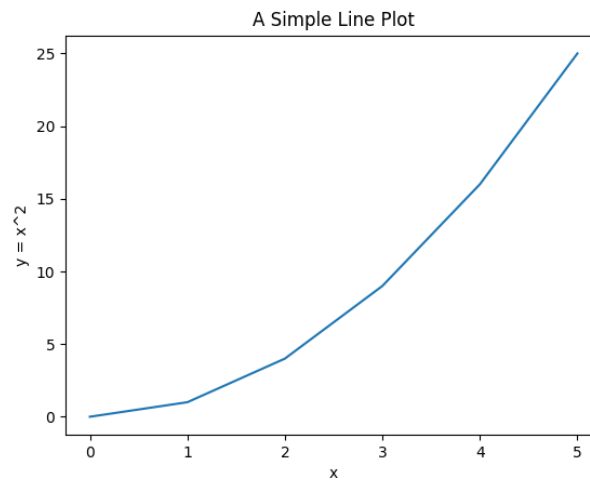


Figure 3.1: The output of line.py.

Exercise 9 A Line Plot

Write a program that plots $y = 2x$ for $x = 0, 1, 2, 3, 4$. Add a title and axis labels.

Working Space

Answer on Page 59

Exercise 10 Cubes

Plot the numbers 1 through 5 against their cubes. Label the axes and give the plot a title.

Working Space

Answer on Page 59

3.4 Bar Plots

A bar plot compares amounts across categories:

```
1 import matplotlib.pyplot as plt
2
3 labels = ["A", "B", "C", "D", "E"]
4 values = [5, 7, 9, 11, 13]
5
6 plt.bar(labels, values)
7 plt.show()
```

Here, labels are the categories, and values are the heights of their bars.

Exercise 11 Votes

Change the bar plot example so that the categories are the names of five of your favorite foods, and the values are the number of votes each food got in a class survey. Use your own numbers.

Working Space

Answer on Page 59

3.5 Several Lines on One Plot

Call `plt.plot` more than once before `plt.show`, and the lines share the same axes. Give each a label, and `plt.legend` draws a key. The marker and linestyle options make the lines easy to tell apart, even in black and white, and `plt.grid` adds grid lines:

```
1 import matplotlib.pyplot as plt
2
3 x = [1, 2, 3, 4, 5]
4 plt.plot(x, [1, 4, 9, 16, 25], marker="o", label="x squared")
5 plt.plot(x, [1, 8, 27, 64, 125], marker="s", linestyle="--", label="x cubed")
6 plt.xlabel("x")
7 plt.ylabel("y")
8 plt.title("Squares and Cubes")
9 plt.grid(True)
10 plt.legend()
```

```
11 plt.show()
```

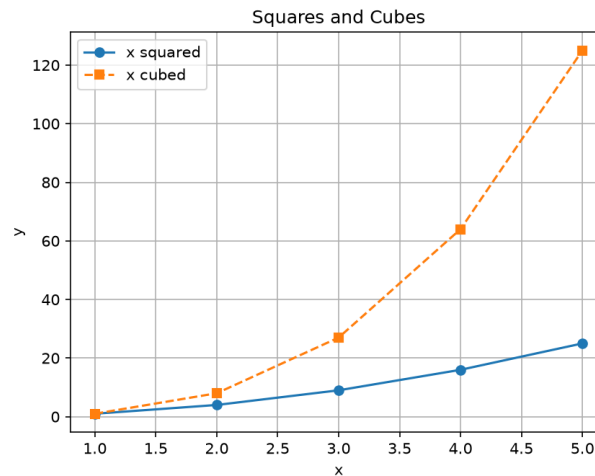


Figure 3.2: Two lines with markers and a legend.

To save a plot as an image file, call `plt.savefig("myplot.png")` *before* `plt.show()`; after the window closes, the figure is gone.

3.6 Scatter Plots

A *scatter plot* draws one dot per data point, with no connecting lines. It is the right tool for asking how two variables relate. Here are six students' study times and test scores:

```
1 import matplotlib.pyplot as plt
2
3 hours = [1, 2, 3, 4, 5, 6]
4 scores = [55, 60, 66, 72, 78, 85]
5
6 plt.scatter(hours, scores)
7 plt.title("Study Time vs. Score")
8 plt.xlabel("hours studied")
9 plt.ylabel("score")
10 plt.show()
```

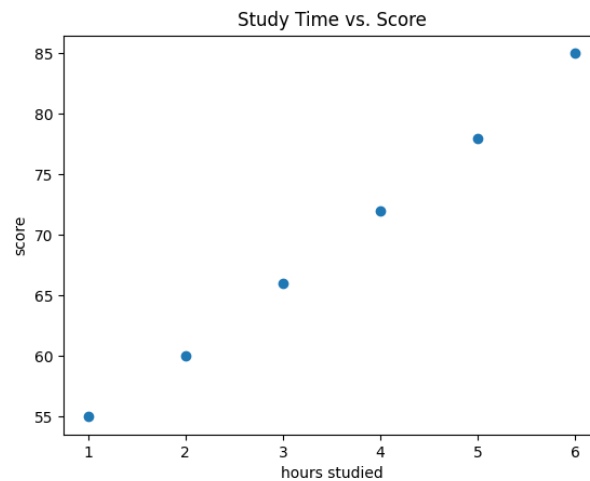


Figure 3.3: More study time goes with higher scores.

3.6.1 Plotting Data from a File

Real data usually lives in a file. Your digital resources include `dataset.csv`, a CSV (comma-separated values) file with 50 rows. Each row has a Category (A, B, C, or D), an `X_values` column, and a `Y_values` column. The line `with open("dataset.csv") as f:` opens the file and calls it `f`; when the indented block ends, Python closes the file for you. Python's built-in `csv` library then reads it for us. `csv.DictReader` turns each row into a dictionary whose keys are the column names. Because the file stores text, we convert the numbers with `float`. This program draws each category in its own color:

```
1 import csv
2 import matplotlib.pyplot as plt
3
4 # Read the file into a list of dictionaries, one per row
5 with open("dataset.csv") as f:
6     rows = list(csv.DictReader(f))
7
8 for category in ["A", "B", "C", "D"]:
9     xs = [float(r["X_values"]) for r in rows if r["Category"] == category]
10    ys = [float(r["Y_values"]) for r in rows if r["Category"] == category]
11    plt.scatter(xs, ys, label=category)
12
13 plt.xlabel("X value")
14 plt.ylabel("Y value")
15 plt.title("Points by Category")
16 plt.legend()
17 plt.show()
```

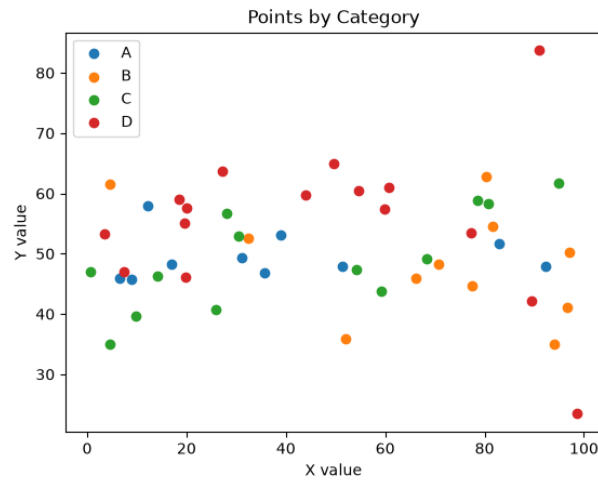


Figure 3.4: The points of `dataset.csv`, colored by category.

The dots are spread all over, with no clear upward or downward trend: in this data set, knowing X does not help you predict Y .

3.7 Histograms

A *histogram* shows how one set of numbers is distributed. It divides the range into equal *bins* and draws a bar for how many values land in each:

```

1 import matplotlib.pyplot as plt
2
3 # for example, maybe these are quiz scores
4 data = [4, 5, 5, 6, 7, 7, 7, 8, 8, 9, 10, 10, 10, 11, 12]
5
6 plt.hist(data, bins=5)
7 plt.title("Histogram of Values")
8 plt.xlabel("value")
9 plt.ylabel("frequency")
10 plt.show()

```

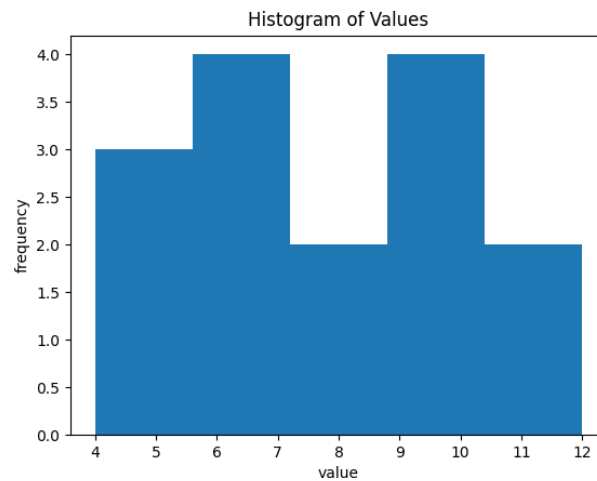


Figure 3.5: A histogram of 15 values in 5 bins.

Worked example: reading a histogram. The histogram above splits the 15 values into 5 bins of equal width, from 4 to 12, so each bin is $\frac{12-4}{5} = 1.6$ units wide. The counts are 3, 4, 2, 4, 2, which add up to 15. Matplotlib chooses the bin edges for you. The tallest bars show where the values cluster.

Here is the distribution of the `Y_values` column of `dataset.csv`:

```
1 import csv
2 import matplotlib.pyplot as plt
3
4 with open("dataset.csv") as f:
5     ys = [float(r["Y_values"]) for r in csv.DictReader(f)]
6
7 plt.hist(ys, bins=10, edgecolor="black")
8 plt.xlabel("Y value")
9 plt.ylabel("Number of points")
10 plt.title("Distribution of Y Values")
11 plt.show()
```

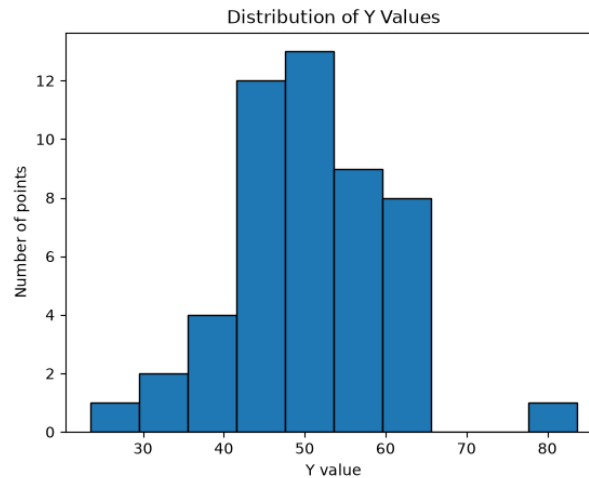


Figure 3.6: Most Y values fall between 40 and 65, with one outlier above 80.

Try changing `bins=10` to 5 or 25. Too few bins hide the shape; too many make it jagged.

Exercise 12 Averages by Category

Using `dataset.csv`, make a bar plot of the average `Y_values` for each category. Which category has the highest average?

Working Space

Answer on Page 59

3.8 The Object-Oriented Approach

Matplotlib has two styles:

- **State-based** (`plt.plot`, `plt.title`, ...): quick and convenient. Everything so far has used this style.
- **Object-oriented** (creating a `Figure` and its `Axes`): more explicit, and easier to manage when a figure has several plots.

For layouts with several plots, prefer the object-oriented approach:

```
1 import matplotlib.pyplot as plt
2
3 x = [0, 1, 2, 3, 4, 5]
4 y1 = [0, 1, 4, 9, 16, 25]
5 y2 = [0, 1, 8, 27, 64, 125]
6
7 # one figure with two plots side by side; axes is a list of them
8 fig, axes = plt.subplots(nrows=1, ncols=2)
9
10 axes[0].plot(x, y1)
11 axes[0].set_title("x^2")
12 axes[0].set_xlabel("x")
13 axes[0].set_ylabel("value")
14
15 axes[1].plot(x, y2)
16 axes[1].set_title("x^3")
17 axes[1].set_xlabel("x")
18 axes[1].set_ylabel("value")
19
20 fig.suptitle("Two Subplots")
21 fig.tight_layout()
22 plt.show()
```

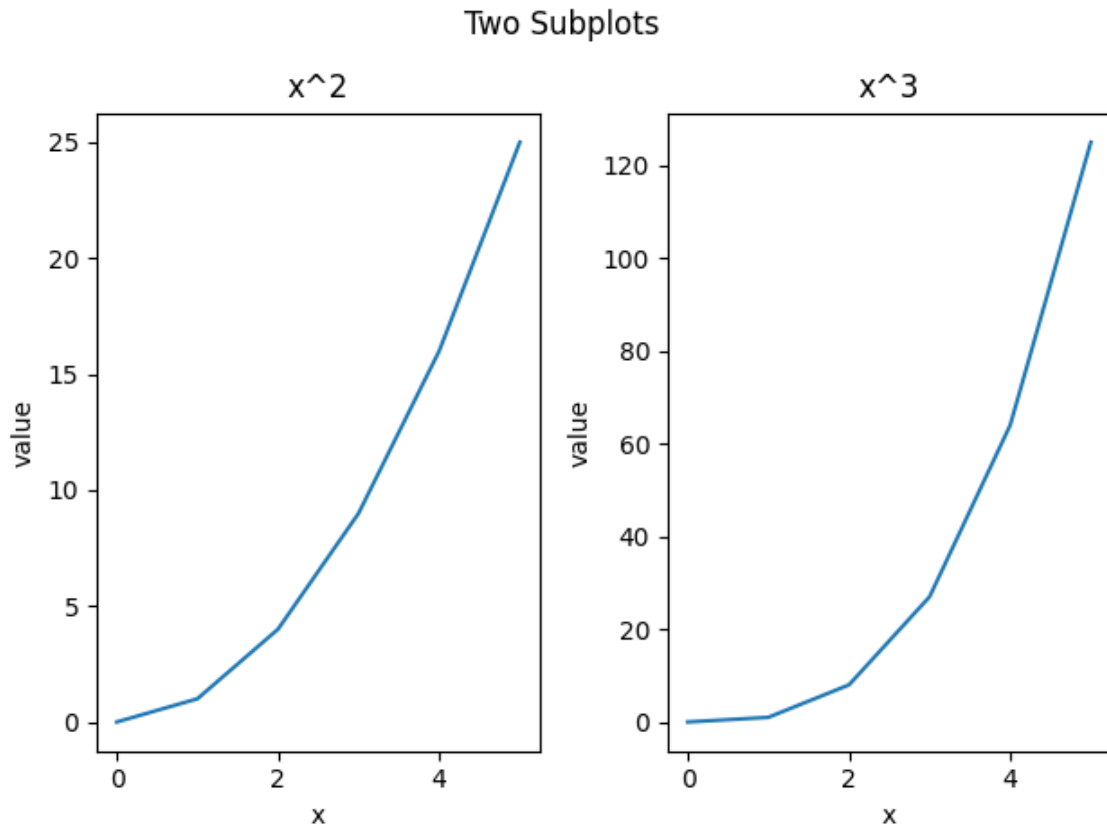


Figure 3.7: The two plots, generated side by side.

3.9 Simulating a Falling Hammer

In the Kinematics chapter, you tossed a hammer straight up, at 12 m/s from 2 meters above the ground, and used formulas to find its height and speed. A computer can get the same answers a different way: by *simulating* the motion in tiny steps. Every hundredth of a second, change the height by the speed times the time step, and change the speed by the acceleration times the time step.

First, we are just going to print out the position, speed, and acceleration of the hammer for every 1/100th of a second after it leaves your hand. (Later, we will make a graph.)

Create a file called `falling.py` and type this into it:

```
1 # Acceleration on earth
2 acceleration = -9.8 # m/s/s
3
4 # Size of time step
5 time_step = 0.01 # seconds
```

```

6
7 # Initial values
8 speed = 12 # m/s upward
9 height = 2 # m above the ground
10 current_time = 0.0 # seconds after release
11
12 # Is the hammer still aloft?
13 while height > 0.0:
14
15     # Show the values
16     print(f"{current_time:.2f} s:")
17     print(f"\tacceleration: {acceleration:.2f} m/s/s")
18     print(f"\tspeed: {speed:.2f} m/s")
19     print(f"\theight: {height:.2f} m")
20
21     # Update height
22     height = height + time_step * speed
23
24     # Update speed
25     speed = speed + time_step * acceleration
26
27     # Update time
28     current_time = current_time + time_step
29
30
31 print(f"Hit the ground: Complete")

```

When you run it, you will see something like this:

```

1 0.00 s:
2     acceleration: -9.80 m/s/s
3     speed: 12.00 m/s
4     height: 2.00 m
5 0.01 s:
6     acceleration: -9.80 m/s/s
7     speed: 11.90 m/s
8     height: 2.12 m
9 0.02 s:
10    acceleration: -9.80 m/s/s
11    speed: 11.80 m/s
12    height: 2.24 m
13 0.03 s:
14    acceleration: -9.80 m/s/s
15    speed: 11.71 m/s
16    height: 2.36 m
17 ...
18 2.60 s:
19    acceleration: -9.80 m/s/s
20    speed: -13.48 m/s
21    height: 0.20 m
22 2.61 s:

```

```
23         acceleration: -9.80 m/s/s
24         speed: -13.58 m/s
25         height: 0.07 m
26 Hit the ground: Complete
```

Note that the acceleration is not changing at all, but it is changing the speed, and the speed is changing the height.

We can see that the hammer in our simulation hits the ground just after 2.61 seconds.

3.9.1 Graphing the Simulation

To graph the acceleration, speed, and height, the program needs to save every value in a list as it goes. Duplicate the file `falling.py` and name the new copy `falling_graph.py`

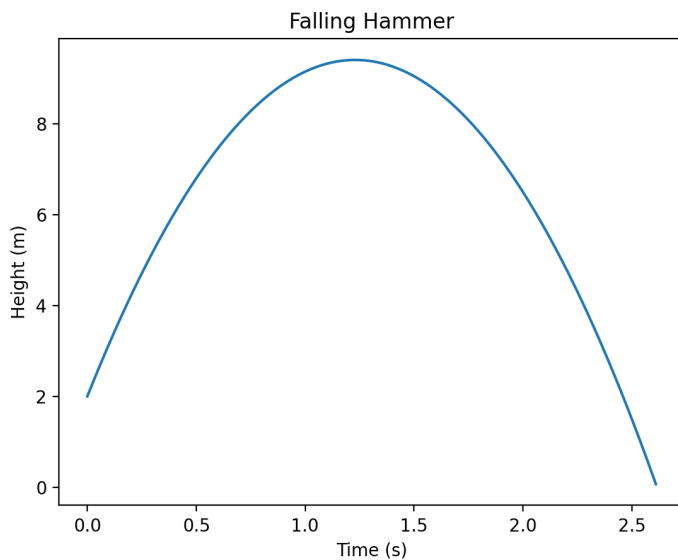
We are going to make a plot of the height over time. At the start of the program, you will import the `matplotlib` library. At the end of the program, you will create a plot and show it to the user.

Change `falling_graph.py` to this:

```
1  import matplotlib.pyplot as plt
2
3  # Acceleration on earth
4  acceleration = -9.8 # m/s/s
5
6  # Size of time step
7  time_step = 0.01 # seconds
8
9  # Initial values
10 speed = 12 # m/s upward
11 height = 2 # m above the ground
12 current_time = 0.0 # seconds after release
13
14 # Create empty lists
15 accelerations = []
16 speeds = []
17 heights = []
18 times = []
19
20 # Is the hammer still aloft?
21 while height > 0.0:
22
23     # Add the data to the lists
24     times.append(current_time)
25     accelerations.append(acceleration)
26     speeds.append(speed)
```

```
27 heights.append(height)
28
29 # Update height
30 height = height + time_step * speed
31
32 # Update speed
33 speed = speed + time_step * acceleration
34
35 # Update time
36 current_time = current_time + time_step
37
38 # Make a plot
39 fig, ax = plt.subplots()
40 fig.suptitle("Falling Hammer")
41 ax.set_xlabel("Time (s)")
42 ax.set_ylabel("Height (m)")
43 ax.plot(times, heights)
44 plt.show()
```

When you run the program, you should see a graph of the height over time.

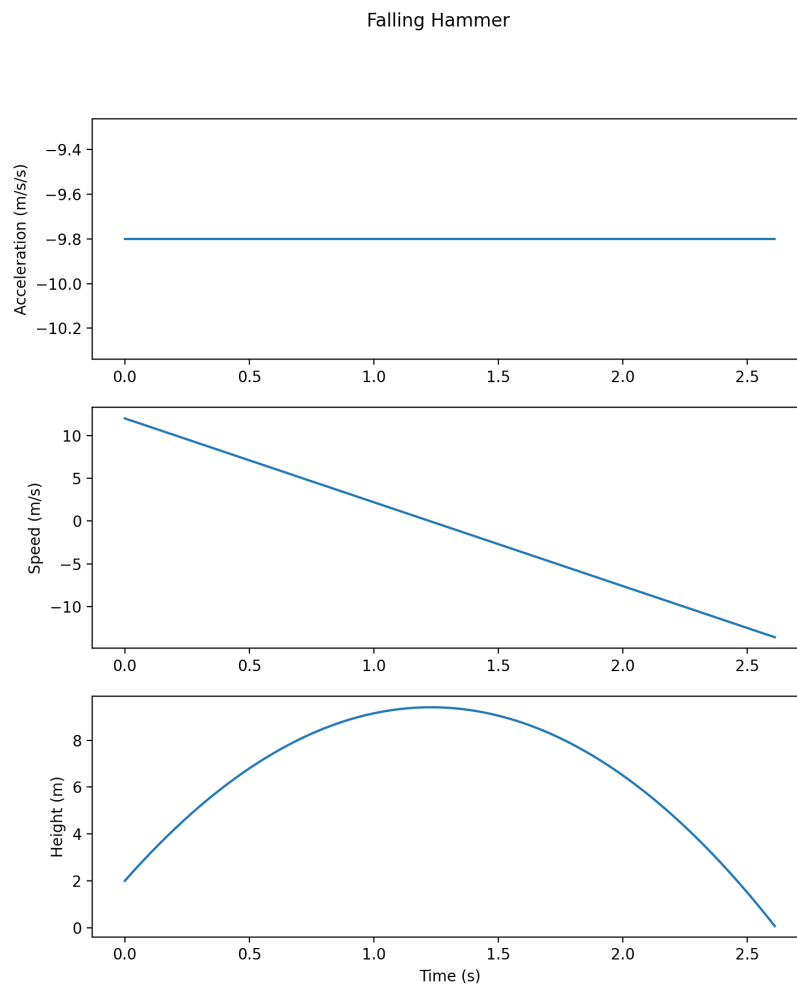


It is more interesting if we can see all three: acceleration, speed, and height. So, let us make three stacked plots. Change the plotting code in `falling_graph.py` to:

```
1 # Make a plot with three subplots
2 fig, ax = plt.subplots(3,1)
3 fig.suptitle("Falling Hammer")
4
5 # The first subplot is acceleration
6 ax[0].set_ylabel("Acceleration (m/s/s)")
```

```
7 ax[0].plot(times, accelerations)
8
9 # Second subplot is speed
10 ax[1].set_ylabel("Speed (m/s)")
11 ax[1].plot(times, speeds)
12
13 # Third subplot is height
14 ax[2].set_xlabel("Time (s)")
15 ax[2].set_ylabel("Height (m)")
16 ax[2].plot(times, heights)
17 plt.show()
```

You will now get plots of all three variables:



This is what we expected, right? The acceleration is a constant negative number. The speed is a straight line with a negative slope. The height is a parabola. **The slope of the**

height graph is the speed, and the slope of the speed graph is acceleration.

Matplotlib can do much more: pie charts, box plots, 3D surfaces, animations, and endless customization of colors and styles. The official documentation at <https://matplotlib.org> has a gallery of examples, each with its code.

Sorting and Big-O Notation

4.1 Why Sorting Matters

Sorting a list is one of the most common things a program does. Phone books, search results, and leaderboards all depend on items being in order. Python already sorts for you:

```
1 scores = [38, 27, 43, 3, 9, 82, 10]
2 print(sorted(scores)) # [3, 9, 10, 27, 38, 43, 82]
```

This chapter shows how sorting can be done by hand, using three simple algorithms. Then it introduces a way to measure how much work an algorithm does as the list grows: *Big-O notation*.

4.2 Selection Sort

Selection sort repeats one idea. Find the smallest item in the unsorted part of the list, and swap it into the next position at the front.

```
1 def selection_sort(items):
2     items = list(items)
3     n = len(items)
4     for i in range(n - 1):
5         smallest = i
6         for j in range(i + 1, n):
7             if items[j] < items[smallest]:
8                 smallest = j
9         items[i], items[smallest] = items[smallest], items[i]
10    return items
11
12 print(selection_sort([5, 2, 9, 1, 5, 6])) # [1, 2, 5, 5, 6, 9]
```

The outer loop fixes one position per pass. The inner loop looks at every remaining item to find the smallest. Notice that the inner loop always runs to the end of the list, even if the list is already sorted.

Worked example: tracing selection sort. Sort [5, 2, 9, 1]. The first pass finds the smallest

item, 1, and swaps it with the 5, giving [1, 2, 9, 5]. The second pass finds 2, which is already in place, so nothing changes: [1, 2, 9, 5]. The third pass swaps 9 and 5, giving [1, 2, 5, 9].

4.3 Insertion Sort

Insertion sort works the way many people sort a hand of playing cards. Take the next card, and slide it left past every larger card until it fits.

```
1 def insertion_sort(items):
2     items = list(items)
3     for i in range(1, len(items)):
4         current = items[i]
5         j = i - 1
6         while j >= 0 and items[j] > current:
7             items[j + 1] = items[j]
8             j -= 1
9         items[j + 1] = current
10    return items
11
12 print(insertion_sort([5, 2, 9, 1, 5, 6])) # [1, 2, 5, 5, 6, 9]
```

Insertion sort is fast when the list is almost sorted, because the inner loop stops early. When the list is in reverse order, it does the most work.

4.4 Merge Sort

Merge sort uses a different strategy: divide and conquer. Split the list in half, sort each half, and then merge the two sorted halves into one. A list of one item is already sorted, which is where the recursion stops.

```
1 def merge_sort(items):
2     if len(items) <= 1:
3         return list(items)
4     mid = len(items) // 2
5     left = merge_sort(items[:mid])
6     right = merge_sort(items[mid:])
7     merged = []
8     i = j = 0
9     while i < len(left) and j < len(right):
10        if left[i] <= right[j]:
11            merged.append(left[i]); i += 1
12        else:
13            merged.append(right[j]); j += 1
14    merged.extend(left[i:])
```

```

15     merged.extend(right[j:])
16     return merged
17
18 print(merge_sort([38, 27, 43, 3, 9, 82, 10])) # [3, 9, 10, 27, 38, 43, 82]

```

4.5 Big-O Notation

We want to compare algorithms without timing them on one computer. Big-O notation describes how the number of steps grows as the input size n grows. We count the steps in terms of n , and then keep only the term that grows fastest, dropping constants and smaller terms.

We write $f(n) = O(g(n))$ when $f(n)$ is no larger than a constant multiple of $g(n)$ for all large enough n . Formally, there are constants $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ whenever $n \geq n_0$.

For example, $f(n) = 3n^2 + 100n + 5$ is $O(n^2)$. With $c = 4$ and $n_0 = 101$, we have $3n^2 + 100n + 5 \leq 4n^2$. The $100n$ term matters for small n , but for large n the n^2 term dominates.

Here are the classes you will meet most often, from fastest to slowest:

Notation	Name	Example
$O(1)$	constant	reading one item from a list by its index
$O(\log n)$	logarithmic	binary search on a sorted list
$O(n)$	linear	checking every item once
$O(n \log n)$	linearithmic	merge sort, and Python's <code>sorted()</code> in the worst case
$O(n^2)$	quadratic	selection sort and insertion sort

4.6 Comparing the Sorts

Selection sort's inner loop runs $n - 1$, then $n - 2$, and so on down to 1 times. That is a total of $\frac{n(n-1)}{2}$ comparisons, which is $O(n^2)$ in every case, even when the list is already sorted. Insertion sort can do far fewer comparisons on nearly sorted lists, but its worst case, a list in reverse order, also takes $\frac{n(n-1)}{2}$ comparisons, so it is also $O(n^2)$.

Merge sort halves the list at each level of recursion, so there are about $\log_2 n$ levels. Each level does about n work to merge. That gives $O(n \log n)$ work in every case.

The table below shows the approximate number of steps for each class. The values are rounded.

n	n	$n \log_2 n$	n^2
10	10	33	100
100	100	664	10,000
1,000	1,000	9,966	1,000,000
1,000,000	1,000,000	about 20,000,000	1,000,000,000,000

When n is a million, a quadratic algorithm does a trillion steps, while a linearithmic algorithm does about twenty million. That is why Big-O matters more than the exact count for large inputs.

Exercise 13 Trace selection sort

Trace the first pass of selection sort on $[4, 1, 3]$. What is the list after the first pass? What is the final sorted list?

Working Space

Answer on Page 60

Exercise 14 Count the comparisons

How many comparisons does selection sort make on a list of 5 items, no matter what order the items are in?

Working Space

Answer on Page 60

Exercise 15 Prove a Big-O bound

Show that $f(n) = 3n^2 + 100n + 5$ is $O(n^2)$ by giving constants c and n_0 .

Working Space

Answer on Page 60

Exercise 16 **Which is faster?**

A program takes $n \log_2 n$ steps and another takes n^2 steps. For $n = 1,000,000$, which one is faster, and roughly how many times faster is it?

Working Space

Answer on Page 60

Testing and Debugging

5.1 Why Test?

Programs fail in small ways. A function that returns the wrong answer for one input can look fine for every other input you try by hand. Testing means writing code that checks your code, so you find errors before someone else does. Debugging means finding the cause of an error once you know it exists.

5.2 Print Debugging

Worked example: finding a bug with print statements. The quickest way to find a bug is to print the values that matter. Suppose `average` returns the wrong answer:

```
1 def average(values):
2     total = sum(values)
3     print("total:", total, "count:", len(values))
4     return total / len(values)
5
6 print(average([1, 2]))
```

The printed values show where the output diverges from what you expected. Remove the print statements when you are done.

5.3 Assertions

An *assertion* checks a condition and stops the program if it is false:

```
1 from mathtools import average
2
3 x = [1, 2, 3]
4 assert average(x) == 2, "average should be 2"
5 print("ok")
```

Assertions are good for stating what must always be true. They are not a replacement for tests, because they run only when the program runs.

5.4 Unit Tests

A *unit test* checks one function on known inputs and expected outputs. Python's `unittest` module runs them for you. Given this file, `mathtools.py`:

```
1 def average(values):
2     """Return the mean of a non-empty list of numbers."""
3     if len(values) == 0:
4         raise ValueError("average() needs at least one value")
5     return sum(values) / len(values)
6
7 def is_even(n):
8     return n % 2 == 0
```

Put the tests in `test_mathtools.py`:

```
1 import unittest
2 from mathtools import average, is_even
3
4 class TestAverage(unittest.TestCase):
5     def test_simple_list(self):
6         self.assertEqual(average([2, 4, 6]), 4)
7
8     def test_single_value(self):
9         self.assertEqual(average([7]), 7)
10
11     def test_empty_list_raises(self):
12         with self.assertRaises(ValueError):
13             average([])
14
15 class TestIsEven(unittest.TestCase):
16     def test_even_and_odd(self):
17         self.assertTrue(is_even(10))
18         self.assertFalse(is_even(7))
19
20 if __name__ == '__main__':
21     unittest.main()
```

Run `python3 -m unittest test_mathtools.py -v`. Each test prints a line, and the final line says OK when every test passes.

5.5 Reading a Failing Test

A test that fails shows what it expected and what it got. If you change the expected value in the test to 2 for `average([1, 2])`, the output reads:

```
1 AssertionError: 1.5 != 2
```

The function is correct and the test is wrong. Before you change the code, decide which one is wrong. Check the function's definition, then the expected value.

5.6 A Debugging Routine

1. Reproduce the error with a specific input.
2. Write a test that fails for that input.
3. Form a guess about the cause, and check it with a print statement or an assertion.
4. Fix the code, and run all the tests again, not just the new one.

Exercise 17 Write a test

Add a test to `test_mathtools.py` that checks that `average([-1, 1])` equals 0.

Working Space

Answer on Page 61

Exercise 18 Which test fails?

A student wrote `self.assertEqual(average([1, 2, 3]), 3)`. What does the test output say, and what is the mistake?

Working Space

Answer on Page 61

Exercise 19 Edge case

What should `average([])` do, and

Working Space

which test checks it?

Answer on Page 61

Version Control with Git

6.1 Why Use Version Control?

Have you ever saved files named `essay_final.docx`, `essay_final2.docx`, and `essay_REALLY_final.docx`? Programmers have the same problem, only worse: one change can break code that worked yesterday, and you need a way back. *Git* records the history of a project so you can compare and undo changes. A project tracked by Git is a *repository*.

6.2 Creating a Repository

Make a folder for your project, then run:

```
1 git init
```

This creates a hidden `.git` folder that stores the history. Nothing is recorded yet.

6.3 Saving a Snapshot: Commits

A *commit* is a saved snapshot of the files you chose to track. Create a file and save it:

```
1 echo "hello" > notes.txt
2 git add notes.txt
3 git commit -m "Add notes"
```

`git add` stages a file, which means it will be included in the next commit. `git commit` saves the staged files with a message that describes the change. Write messages that explain why the change was made.

To see the history, run `git log --oneline`. Each commit has a short identifier and its message.

Worked example: a first commit. Start in an empty folder and run `git init`. Create the file, stage it, and commit it:

```
1 echo "hello" > notes.txt
2 git add notes.txt
3 git commit -m "Add notes"
4 git log --oneline
```

The last command shows one line for each commit. Your line will show a different identifier, but it will show your message: Add notes. The identifier is a short form of the commit's unique hash.

6.4 Branches

A *branch* is a separate line of work. Making a branch lets you try an idea without changing the main line of your project:

```
1 git branch new-idea
2 git switch new-idea
```

Commits made on `new-idea` do not appear on the main branch. Run `git switch` with the name of the main branch to return to it. If the idea works, you can merge it back in. If it does not, you can delete the branch.

6.5 A Routine for Each Session

1. Check what changed with `git status`.
2. Stage the files you want to save with `git add`.
3. Commit them with a clear message.
4. Repeat after each small step that works.

Exercise 20 Record a change

You edit `notes.txt` and want to save the change. Which three commands do you run, in order?

Working Space

Answer on Page 61

Exercise 21 **Branch or main?**

You want to try a new version of a function without risking the working code. What should you do?

Working Space

Answer on Page 61

Answers to Exercises

Answer to Exercise 1 (on page 9)

On Earth, holding a 100 kg person aloft requires 980 newtons of force.

$980/700 = 1.4$, so you need a cable with a cross-section area of 1.4 square millimeters.

$$\pi r^2 = 1.4$$

$r = \sqrt{1.4/\pi} \approx .67$ millimeters. This means the cable would have to have a diameter of at least 1.34 millimeters.

Answer to Exercise 2 (on page 9)

$$F = \sigma A = (700 \times 10^6)(4 \times 10^{-6}) = 2800 \text{ N.}$$

Answer to Exercise 3 (on page 14)

The function returns $3 * 2$, which is 6, so the program prints:

```
1 6
```

The value of `a` is still 3 because the function does not change `a`; it only uses its value to compute a result.

Answer to Exercise 4 (on page 15)

```
1 def is_even(n):  
2     return n % 2 == 0  
3
```

```
4 print(is_even(7))
```

This prints `False` because 7 is not divisible by 2.

Answer to Exercise 5 (on page 19)

The initial value is 5. The `increment()` method is called twice, so the value becomes 7. The program prints:

```
1 7
```

Answer to Exercise 6 (on page 19)

```
1 class Book:
2     def __init__(self, title, author):
3         self.title = title
4         self.author = author
5
6     def __str__(self):
7         return f"{self.title} by {self.author}"
```

Answer to Exercise 7 (on page 22)

1. **TypeError**: The `input()` function returns a string, so when the user inputs an index, it is treated as a string. Attempting to use this string as an index for the list will raise a **TypeError**. To fix this, we can cast the input to an integer using `int()` and handle the potential **ValueError** if the input is not a valid integer.
2. Pencil is not in the list, so attempting to remove it will raise a **ValueError**. To fix this, we can use a try-except block to catch the **ValueError** and print a message indicating that the item was not found in the list.

Answer to Exercise 8 (on page 26)

```
1 def factorial(n):
2     if n == 0:
```

```
3     return 1
4     return n * factorial(n - 1)
```

Answer to Exercise 9 (on page 29)

```
1 import matplotlib.pyplot as plt
2
3 x = [0, 1, 2, 3, 4]
4 y = [2 * v for v in x]
5
6 plt.plot(x, y)
7 plt.title("y = 2x")
8 plt.xlabel("x")
9 plt.ylabel("y")
10 plt.show()
```

The plot is a straight line through (0,0) and (4,8).

Answer to Exercise 10 (on page 29)

```
1 import matplotlib.pyplot as plt
2
3 x = [1, 2, 3, 4, 5]
4 y = [1, 8, 27, 64, 125]
5
6 plt.plot(x, y)
7 plt.xlabel("x")
8 plt.ylabel("x cubed")
9 plt.title("Cubes of the First Five Numbers")
10 plt.show()
```

Answer to Exercise 11 (on page 30)

Replace the labels and values lists with your own data. For example, labels = ["Pizza", "Tacos", "Ramen", "Salad", "Pasta"] and values = [12, 9, 7, 4, 10]. Then call plt.bar(labels, values) and plt.show(), as in the example above.

Answer to Exercise 12 (on page 35)

```
1 import csv
2 import matplotlib.pyplot as plt
3
4 with open("dataset.csv") as f:
5     rows = list(csv.DictReader(f))
6
7 categories = ["A", "B", "C", "D"]
8 averages = []
9 for category in categories:
10     ys = [float(r["Y_values"]) for r in rows if r["Category"] == category]
11     averages.append(sum(ys) / len(ys))
12     print(category, round(averages[-1], 1))
13
14 plt.bar(categories, averages)
15 plt.xlabel("Category")
16 plt.ylabel("Average Y value")
17 plt.title("Average Y Value by Category")
18 plt.show()
```

The averages are about A: 49.5, B: 48.4, C: 49.1, and D: 55.5. Category D is highest.

Answer to Exercise 13 (on page 46)

The first pass finds that 1 is the smallest item, and swaps it with the 4. The list becomes [1, 4, 3]. The second pass swaps 4 and 3, giving the final list [1, 3, 4].

Answer to Exercise 14 (on page 46)

$\frac{5 \cdot 4}{2} = 10$ comparisons.

Answer to Exercise 15 (on page 46)

Take $c = 4$ and $n_0 = 101$. For $n \geq 101$, we have $100n + 5 \leq n^2$, because $n^2 - 100n - 5 = n(n - 100) - 5 \geq 101 \cdot 1 - 5 > 0$. So $3n^2 + 100n + 5 \leq 3n^2 + n^2 = 4n^2$.

Answer to Exercise 16 (on page 47)

The $n \log_2 n$ program is faster. It takes about 2×10^7 steps, while the n^2 program takes 10^{12} steps. The first is about 50,000 times faster.

Answer to Exercise 17 (on page 51)

```
1 def test_negative_values(self):  
2     self.assertEqual(average([-1, 1]), 0)
```

This test passes, because $(-1 + 1)/2 = 0$.

Answer to Exercise 18 (on page 51)

`average([1, 2, 3])` is $\frac{6}{3} = 2$, not 3. The output says `AssertionError: 2.0 != 3`. The expected value in the test is wrong.

Answer to Exercise 19 (on page 51)

It should raise a `ValueError`. The test `test_empty_list_raises` checks this.

Answer to Exercise 20 (on page 54)

`git status` (optional, to check), then `git add notes.txt`, then `git commit -m "Describe the change"`.

Answer to Exercise 21 (on page 55)

Create a branch with `git branch` and switch to it with `git switch`. Work and commit there. Switch back to the main branch if the experiment fails.



INDEX

- aggregate, 7
- alloy, 8
- ammonia, 6
- argument, 12
- assertion, 49
- attributes, 16

- bar plot, 30
- base case, 23
- base condition, 23
- Big-O notation, 43
 - definition, 45
- binary search, 24
- bins, 33
- branch, 54

- cement, 7
- class, 15
- classes, 15
 - defining, 15
 - definition, 15
- clinker, 7
- commit, 53
- compressive stress, 7
- concrete, 7
- corrosion, 10
- creating charts in python, *see* matplotlib
- CSV, 32

- debug, 20
- debugging, 49
 - print statements, 49
- divide-and-conquer, 24
- divide-and-conquer algorithms, 24

- error, 20
- Errors, 20
- exceptions, 20

- fertilizer, 5
- fibonacci, 23
- function, 11
- functions, 11
 - argument, 12
 - def, 11
 - definition, 11
 - parameter, 12
 - parameters, 12
 - return, 12

- Git, 53

- Haber-Bosch process, 6
- histogram, 33

- init, 16
- insertion sort, 44

- libraries, 27
- library, 27
- line plot, 28
- linear search, 24

- malnutrition, 5

- matplotlib, 27
 - subplots, 40
- megapascal, 9
- merge sort, 44
- metals, 8
- methods, 17
 - definition, 17
- mortar, 7
- NameError, 21
- natural gas, 6
- nitrogen, 5
- nitrogen cycle, 6
- nitrogen fixation, 5, 6
- object, 16
- Object-Oriented Programming, 15
- object-oriented programming, 15
- objects, 16
 - definition, 16
- parameter, 12
- phosphorus, 5
- photosynthesis, 6
- Portland cement, 7
- potash, 5
- potassium, 5
- printing objects, 17
- rebar, 8
- recursion, 23, 24
- recursive call, 23
- recursive function, 23
- repository, 53
- rhizobia, 6
- scatter plot, 31
- scope, 14
- selection sort, 43
- simulation
 - falling body, 37
- sorting, 43
 - comparison, 45
- stainless steel, 8
- steel, 8
- steel reinforced concrete, 8
- steel-reinforced concrete, 8
- symbiosis, 6
- syntax, 20
- syntax errors, 20
- tensile, 7
- tensile strength, 9
- tensile stress, 7
- testing, 49
- titanium, 10
- try-except, 22
- type hints, 13
- unit test, 50
- urea, 6
- urine, 6
- version control, 53