4

POINTS AND VECTORS

Points and vectors are the basis of geometry. In this book, we'll use them as our *primitives*, the building blocks for the rest of our geometry library. For our geometry library to be usable, it's crucial that we implement points and vectors using bug-free code. A bug in our code will not only cause errors in the library's functions but could propagate to all the other libraries we build on top of it, giving us all sorts of false calculations.

In this chapter, we have two main tasks. First, we need to implement classes to represent both points and vectors. Then, we need to make sure our code is bug-free by unit testing, a process we'll repeat throughout this book. Before we can do either, though, we need to implement a few useful methods.

Comparing Numbers

When it comes to representing real numbers, computers don't have infinite precision. Most computers use floating-point numbers to store these values, which cannot represent every rational number, let alone irrational numbers.

Thus, when comparing floating-point numbers, you have to specify a *tolerance*: a number ϵ as small as you need such that

$$|a - b| < \epsilon$$

where a and b are the two numbers you want to compare.

A tolerance's order of magnitude needs to be consistent with the problem's magnitudes and your desired precision. For example, it wouldn't make much sense to use a tolerance of $1E^{-20}$mm when working with a planet's orbital lengths, which are on the order of millions of kilometers. Similarly, it would be pointless to use a tolerance of $1E^{-2}$cm when working with atomic distances.

Before we start writing our primitives, we'll need a way of knowing whether two floating-point numbers can be considered equal or not given a tolerance ϵ. But we can't rely on the computer to compare floating-point numbers, as a different digit in the hundredth decimal is logically considered to be a completely different number. So, we'll start this chapter by writing a function that compares two numbers using a given tolerance. For our geometrical calculations, we'll use a default tolerance of $1E^{-10}$, which is an acceptable level of precision for most of the calculations we'll do throughout the book.

Open your project in the IDE, right-click the project's root folder, and select **New ▶ Python Package**. Name it *geom2d* and click **OK**. This will be the package for all of our geometry code.

NOTE *Because the package name establishes that everything inside is in 2D, we won't repeat this piece of information when giving names to our files and classes. Inside the package, we'll use names like* `point` *or* `segment` *instead of* `point2d` *or* `segment2d`*. If we wanted to create a three-dimensional geometry package,* `geom3d`*, we'd still use* `point` *and* `segment`*, only with different, three-dimensional implementations.*

Create a new file by right-clicking the *geom2d* package folder and selecting **New ▶ Python File**. Name it `nums`, leave the Kind drop-down as is, and click **OK**.

With the file created, let's implement our first comparison function. Listing 4-1 has the code for our function.

```
import math


def are_close_enough(a, b, tolerance=1e-10):
    return math.fabs(a - b) < tolerance
```

Listing 4-1: Comparing numbers

First, we import the `math` module, part of Python's standard library that contains useful mathematical functions. Our function takes two numbers, `a` and `b`, and an optional `tolerance` parameter that will default to $1E^{-10}$ if no other value is provided. Last, we use the `math` library's `fabs` function to check

whether the absolute value of the difference between `a` and `b` is smaller than the tolerance, and we return the appropriate Boolean.

In practice, we'll find there are two particular values we're comparing against several: zero and one. To save us from repeatedly writing something like this:

```
are_close_enough(num, 1.0, 1e-5)
```

or this:

```
are_close_enough(num, 0.0, 1e-5)
```

let's implement them as functions. After the previous function, add the code in Listing 4-2.

```
--snip--

def is_close_to_zero(a, tolerance=1e-10):
    return are_close_enough(a, 0.0, tolerance)


def is_close_to_one(a, tolerance=1e-10):
    return are_close_enough(a, 1.0, tolerance)
```

Listing 4-2: Number to zero or one comparison

Functions like the ones in Listing 4-2 aren't strictly necessary, but they are convenient, and they make the code more readable.

The Point Class

A point, according to Euclid's first volume of the *Elements*, is "that of which there is no part." In other words, a point is an entity with no width, length, or depth. It is just a position in space, something you can't see with your naked eye. Points are the base of all Euclidean geometry, and everything else in his writings is based on this simple concept. Accordingly, our geometry library will also be based on this powerful primitive.

A point consists of two numbers, x and y. These are its coordinates, sometimes also called *projections*. Figure 4-1 depicts a point P and its coordinates in the Euclidean plane.

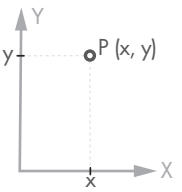

Figure 4-1: A point P in the plane

Let's implement a class representing a two-dimensional point. As before, we'll create a new file by right-clicking the *geom2d* package folder and selecting **New ▶ Python File**. Name it *point* and click **OK**. Inside the file, enter the code in Listing 4-3.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Listing 4-3: Our `Point` class

The coordinates are passed to the initializer method (`__init__`) and stored as attributes of the class.

With our initializer written, let's implement some functionality.

Calculating Distance Between Points

To compute the distance $d(P, Q)$ between the two points P and Q, we use Equation 4.1.

$$d(P, Q) = \sqrt{(Q_x - P_x)^2 + (Q_y - P_y)^2} \tag{4.1}$$

Here, P_x and P_y are P's coordinates, and Q_x and Q_y are Q's coordinates. We can see this graphically in Figure 4-2.

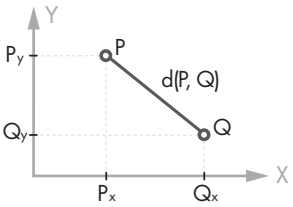

Figure 4-2: Distance between the points P and Q

We can implement our distance calculation in two ways. We could call the method on a point `p` to compute the distance to another point `q`, as in `p.distance_to(q)`. We could also implement the same calculation as a function where both points are given as arguments: `distance_between(p, q)`. The former is the object-oriented style; the latter is functional. Because we're doing object-oriented programming here, we'll go with the former.

Listing 4-4 has the code to implement Equation 4.1 in our class.

```
import math


class Point:
    --snip--
```

```
    def distance_to(self, other):
        delta_x = other.x - self.x
        delta_y = other.y - self.y
        return math.sqrt(delta_x ** 2 + delta_y ** 2)
```

Listing 4-4: Calculating the distance between two points

First, we need to import the `math` module, which loads a bunch of useful mathematical operations into our class. We define the `distance_to` method with `self` and `other` as arguments: `self` is the current point, and `other` is the point we want to calculate the distance to. We then calculate the distance (or *delta*) between the two coordinates and use the power (**) operator to square both deltas and return the square root of their sum.

Let's test this. Open the Python console from the IDE and try the following:

```
>>> from geom2d.point import Point
>>> p = Point(1, 3)
>>> q = Point(2, 4)
>>> p.distance_to(q)
1.4142135623730951
```

Exciting! We've taken the first major step in building our geometry library—Euclid would be proud. You can try that same operation with your calculator and see whether our implementation yields the correct result. Later in the chapter, we'll automate a test that checks that the distance method yields the right result.

While we have the console open and `p` and `q` loaded, try the following:

```
>>> p
<geom2d.point.Point object at 0x10f8a2588>

>>> p.__dict__
{'x': 1, 'y': 3}
```

Evaluating point `p` yields a string telling us `p` is an object of class `Point` at memory position `0x10f8a2588`. Note that the memory address you obtain will likely be different than mine. Without knowing everything in the computer's memory (and reading hexadecimal), this description isn't much help. You can also inspect the `__dict__` attribute of any class to get a dictionary of all the attributes it holds. That gives you more interesting information about the instance. Later in the chapter, we'll be implementing a special method that will help print a cleaner description of the object, something like `(2, 5)`.

Let's now focus our attention on overloading the `+` and `-` operators for the `Point` class.

The Addition and Subtraction Operators

The next basic operations we'll need are addition and subtraction, operations that we'll also implement for vectors. We'll use these basic methods quite often, both on their own and to build more complex methods. We could implement them as normal methods, calling them with something like `p.plus(q)` and `p.minus(q)`, but we can do better. Python allows us to overload + and − operators (as we learned in the "Magic Methods" section from Chapter 2) so that we can write `p + q` and `p - q` and have Python know to add and subtract the points correctly. Overloading operators makes code like this much easier to read and understand.

Overloading an operator in Python involves implementing a method using a specific name that corresponds to the operator. Then, when Python finds the operator, it will replace it with the method you've defined and call it. For the + operator, the name is `__add__`, and for −, it is `__sub__`. Table 4-1 contains common operators we can overload in our classes.

Table 4-1: Python's Overloadable Operators

Operator	Method Name	Description
`+`	`__add__(self, other)`	Addition
`-`	`__sub__(self, other)`	Subtraction
`*`	`__mul__(self, other)`	Multiplication
`/`	`__truediv__(self, other)`	Division
`%`	`__mod__(self, other)`	Modulo
`==`	`__eq__(self, other)`	Equality
`!=`	`__ne__(self, other)`	Inequality
`<`	`__lt__(self, other)`	Less than
`<=`	`__le__(self, other)`	Less than or equal to
`>`	`__gt__(self, other)`	Greater than
`>=`	`__ge__(self, other)`	Greater than or equal to

Let's implement the addition and subtraction operations as methods. Inside the `Point` class and after the `distance_to` method, add the code in Listing 4-5.

```
class Point:
    --snip--

    def __add__(self, other):
        return Point(
            self.x + other.x,
            self.y + other.y
        )

    def __sub__(self, other):
        return Point(
            self.x - other.x,
            self.y - other.y
```

```
        )
```

Listing 4-5: Adding and subtracting points

`__add__` creates and returns a new `Point` where its projections are the sum of the two parameters' projections. This operation doesn't make a lot of sense algebraically speaking, but we may find it useful later. `__sub__` does the same where the resulting projections are the subtraction of the input points' projections. Subtracting two points $P - Q$ yields a vector going from Q to P, but we haven't created a class for vectors yet. We'll refactor this code in the next section so that it returns a vector instance.

Let's implement our next major primitive: the vector.

The Vector Class

Similar to points, *vectors* in the Euclidean plane are composed of two numbers, called the coordinates, that encode a magnitude and a direction. The vector $\langle 3, 5 \rangle$, for instance, can be understood as the displacement achieved by moving 3 units in the positive direction of the horizontal axis and 5 units in the positive direction of the vertical axis. Figure 4-3 depicts a vector $\vec{p}$ in the Euclidean plane.

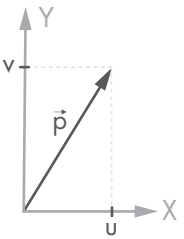

Figure 4-3: A vector $\vec{p}$ in the plane

Many physical quantities are vectorial: they require both a magnitude and a direction to be completely defined. For example, velocities, accelerations, and forces are all vector quantities. Since vectors are so common, let's create a class to represent them.

Right-click the *geom2d* package folder and select **New ▶ Python File**. Name it *vector* and click **OK**. Then enter the code in Listing 4-6.

```
class Vector:
    def __init__(self, u, v):
        self.u = u
        self.v = v
```

Listing 4-6: Vector class

The implementation of `Vector` is similar to that of the `Point` class. The coordinates are named `u` and `v` instead of `x` and `y`. This is just a convention to avoid mixing points and vectors unwittingly.

Before we move on, let's refactor the `Point`'s `__sub__` method so that it returns a `Vector`. Recall that subtracting two points $P - Q$ yields a vector going from Q to P. Modify your *point.py* file so that it now matches Listing 4-7.

```
import math

from geom2d.vector import Vector


class Point:
    --snip--

    def __sub__(self, other):
        return Vector(
            self.x - other.x,
            self.y - other.y
        )
```

Listing 4-7: Refactoring Point __sub__ method

We'll take a closer look at this operation in the "Vector Factories" section of this chapter, where we'll use this operation to create vectors.

Let's now implement some useful methods for the `Vector` class.

Addition and Subtraction Operators

Like with points, adding vectors and subtracting them are common operations. For example, we can get the sum of two forces (which are vector quantities) by summing the vectors representing them.

After the `__init__` method, enter the code in Listing 4-8.

```
class Vector:
    --snip--

    def __add__(self, other):
        return Vector(
            self.u + other.u,
            self.v + other.v
        )

    def __sub__(self, other):
        return Vector(
            self.u - other.u,
            self.v - other.v
        )
```

Listing 4-8: Vector addition and subtraction

In both the `__add__` and `__sub__` methods, we create a new instance of `Vector` to hold the addition or subtraction of projections.

Figure 4-4 depicts the addition and subtraction operations of two vectors, $\vec{p}$ and $\vec{q}$. Notice how subtracting $\vec{p} - \vec{q}$ can be interpreted as the sum of $\vec{p}$ and $-\vec{q}$.

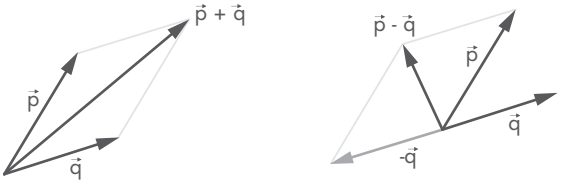

Figure 4-4: A sum of two vectors: $\vec{p} + \vec{q}$

Now you might be wondering if we'll do the same thing for the other operators. Addition and subtraction translate easily to the world of points and vectors, but for something like the `__mul__` operator (used to overload the multiplication operation), it's not as simple. It's unclear whether multiplication would be the dot product, the cross product, or a vector scaling operation. Instead of using a single operator, we'll simply implement these operations as methods with descriptive names: `scaled_by`, `dot`, and `cross`.

We'll begin with scaling.

Scaling Vectors

To *scale* a vector $\vec{u}$, you multiply it by a magnitude k called a *scalar*, which will stretch or shrink the vector. Mathematically, the scalar multiplication looks like this:

$$k \cdot \vec{u} = k \cdot \begin{Bmatrix} u_x \\ u_y \end{Bmatrix} = \begin{Bmatrix} k \cdot u_x \\ k \cdot u_y \end{Bmatrix} \tag{4.2}$$

Let's create a scaling method in the `Vector` class. Enter the code in Listing 4-9 under the `__sub__` method.

```
class Vector:
    --snip--

    def scaled_by(self, factor):
        return Vector(factor * self.u, factor * self.v)
```

Listing 4-9: Scaling a vector

In the previous code, we simply return a new `Vector` whose `u` and `v` attributes are multiplied by `factor`, the passed-in scalar.

Displacing Points

Using the `scaled` method, we can implement another operation: displacing a point P by a given vector $\vec{u}$ k times. Mathematically, that looks like this:

$$\begin{pmatrix} P_x \\ P_y \end{pmatrix} + k \cdot \begin{Bmatrix} u_x \\ u_y \end{Bmatrix} = \begin{pmatrix} P_x + k \cdot u_x \\ P_y + k \cdot u_y \end{pmatrix} \tag{4.3}$$

Graphically it looks like Figure 4-5.

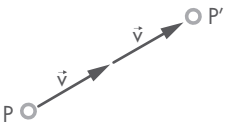

Figure 4-5: Displacing a point P by a vector $\vec{v}$ a given number times k (2 in this case)

Let's implement it programmatically inside our `Point` class, as the displacement subject is the point (Listing 4-10).

```
class Point:
    --snip--

    def displaced(self, vector: Vector, times=1):
        scaled_vec = vector.scaled_by(times)
        return Point(
            self.x + scaled_vec.u,
            self.y + scaled_vec.v
        )
```

Listing 4-10: Displacing a point P by a vector $\vec{v}$ a given number of times k

The method gets passed two arguments: a vector `vector` and a scalar `times`. The vector is scaled according to `times` to produce the net displacement. For instance, a vector $\langle 3, 5 \rangle$ scaled with `times = 2` would result in a displacement of $\langle 6, 10 \rangle$. Note the parameter `times` gets a default value of 1, as often the passed `vector` already has the desired length. The returned point results from adding the coordinates of the source point and the displacement vector's coordinates.

Let's try to move a point in the Python terminal. Restart the console so the previously imported `Point` and `Vector` classes don't get in the way, and enter the following:

```
>>> from geom2d.point import Point
>>> from geom2d.vector import Vector

>>> p = Point(2, 3)
>>> v = Vector(10, 20)
>>> p_prime = p.displaced(v, 2)
>>> p_prime.__dict__
{'x': 22, 'y': 43}
```

You can use a calculator to confirm that the math works as expected.

Vector Norms

A *norm* of a vector is its length. A *unitary norm* is a norm whose length is exactly one unit. Vectors with a unitary norm are useful for defining directions; hence, we'll frequently want to know whether a vector has a unitary norm (whether it's *normal*). We'll also frequently want to *normalize* a vector: keep its direction but scale it to have a length of 1. The norm of a two-dimensional vector is given by Equation4.4.

$$\|\vec{u}\| = \sqrt{u_x^2 + u_y^2} \tag{4.4}$$

Let's implement a property that returns the norm of `Vector`, and let's implement another property that checks whether the vector is normal. Both are included in Listing 4-11.

```
import math

from geom2d import nums


class Vector:
    --snip--

    @property
    def norm(self):
        return math.sqrt(self.u ** 2 + self.v ** 2)

    @property
    def is_normal(self):
        return nums.is_close_to_one(self.norm)
```

Listing 4-11: Norm of a vector

The value obtained from the `norm` property follows exactly the definition from Equation 4.4. To know whether a vector has a norm of 1, we use our numeric comparison `is_close_to_one` and pass in the vector's norm.

We'll implement two other important operations: a method that normalizes a vector $\vec{u}$, yielding a vector $\hat{u}$ with the same direction but unitary length, and a method that scales a vector to have a given length. A normalized version of a vector, which we'll call a *unit vector* or *versor*, can be obtained as follows:

$$\hat{u} = \frac{\vec{u}}{\|\vec{u}\|} = \frac{1}{\sqrt{u_x^2 + u_y^2}} \cdot \begin{Bmatrix} u_x \\ u_y \end{Bmatrix} = \begin{Bmatrix} \frac{u_x}{\sqrt{u_x^2+u_y^2}} \\ \frac{u_y}{\sqrt{u_x^2+u_y^2}} \end{Bmatrix} \tag{4.5}$$

A vector computed this way will have a length of 1. Multiplying that vector by a scalar k results in a vector $\vec{u}_k$, which has the same direction as the original but with a new length that's exactly the value of the scalar, as shown in Equation4.6.

$$\vec{u}_k = k\frac{\vec{u}}{||\vec{u}||} = \frac{k}{\sqrt{u_x^2 + u_y^2}} \cdot \begin{Bmatrix} u_x \\ u_y \end{Bmatrix} = \begin{Bmatrix} \frac{k \cdot u_x}{\sqrt{u_x^2+u_y^2}} \\ \frac{k \cdot u_y}{\sqrt{u_x^2+u_y^2}} \end{Bmatrix} \tag{4.6}$$

In Listing 4-12, we'll turn those equations into code.

```
class Vector:
    --snip--

    def normalized(self):
        return self.scaled_by(1.0 / self.norm)

    def with_length(self, length):
        return self.normalized().scaled_by(length)
```

Listing 4-12: Vectors with unit or chosen length

To normalize a vector, we scale it by the inverse of its norm (which is equivalent to dividing the vector's length by its norm). When we want a vector scaled to a given length, we simply normalize the vector and then scale it by the desired length.

Immutable Design

You may have realized by now that we never mutate the attributes of any of our objects but rather create and return a new `Point` or `Vector` instance. To normalize a vector, for instance, we could have used the code in Listing 4-13.

```
def normalize(self):
    norm = self.norm
    self.x = self.x / norm
    self.y = self.y / norm
```

Listing 4-13: Normalization of a vector in place

Calling that method would result in a *normalization in place*, that is, a mutation of the current object's attributes. Normalizing in place is faster and requires less memory but is also much more error-prone. It's easier than it seems for your program to mistakenly mutate an object that is being used by other parts of the program not expecting the change. Finding these kinds of bugs is really tricky and requires extensive debugging. Furthermore, programs using immutable data are much easier to understand and reason about, as you don't need to keep track of how objects change their state with respect to time.

Take a look at the following code. It implements the `normalize` method in a similar way to the previous one, but it contains a subtle error. In this case, the normalization would yield a wrong result. Can you spot why?

```
def normalize(self):
    self.x = self.x / self.norm
    self.y = self.y / self.norm
```

This is a tricky one. By mutating the `self.x` attribute in the first line, the second call to get the `self.norm` property will use the updated value for `self.x`. The first and second calls to `self.norm` yield different results, which is why we had to store the value of `self.norm` in a variable.

When the amount of data the object has is small, you're better off avoiding mutations altogether. Your program will behave correctly if executed concurrently, and your code will be simpler to understand. Reducing mutability to a minimum will make your code more robust; as you'll see throughout the book, we'll adhere to this principle as much as we can.

Naming Convention

Notice the naming convention for methods. Methods mutating the state of the object upon calling are named as follows:

- `normalize`: Normalizes the vector in place
- `scale_by`: Scales the vector in place

Methods creating a new object as their result are named as follows:

- `normalized`: Returns a new normalized vector
- `scaled_by`: Returns a new scaled vector

Next, we'll implement the dot and cross products in our `Vector` class. These simple products will open the door to some useful operations such as computing the angle between two vectors or testing for perpendicularity.

Dot Product

The *dot product* between two vectors $\vec{u}$ and $\vec{v}$ yields a scalar value, a measure of how different the directions of the two vectors are. In two dimensions, with θ being the angle between the vectors, this product is given by Equation 4.7.

$$\vec{u} \cdot \vec{v} = ||\vec{u}|| \cdot ||\vec{v}|| \cdot cos\theta = u_x \cdot v_x + u_y \cdot v_y \tag{4.7}$$

To understand the different values the dot product can have depending on the relative directions of the two operand vectors, let's take a look at Figure 4-6. This figure depicts a reference vector $\vec{v}$ and three other vectors: $\vec{a}$, $\vec{b}$, and $\vec{c}$. A line perpendicular to $\vec{v}$ divides the space in two half-planes. Vector $\vec{b}$ lies on that line, so the angle θ between $\vec{v}$ and $\vec{b}$ is 90°, and since $cos(90°) = 0$, then $\vec{v} \cdot \vec{b} = 0$. Perpendicular vectors yield a dot product of zero. Vector $\vec{a}$ happens to be on the same half-planes as $\vec{v}$; therefore, $\vec{v} \cdot \vec{a} > 0$. Lastly, $\vec{c}$ is on the opposite half-plane of $\vec{v}$; hence, $\vec{v} \cdot \vec{c} < 0$.

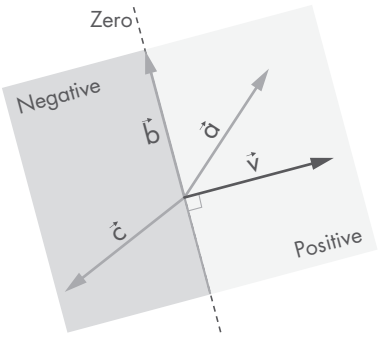

Figure 4-6: Vector directions with respect to $\vec{v}$ yield different dot products.

Implementing the dot product is straightforward from Equation 4.7. Inside the `Vector` class, enter the code in Listing 4-14.

```
class Vector:
    --snip--

    def dot(self, other):
        return (self.u * other.u) + (self.v * other.v)
```

Listing 4-14: Dot product

Before we move on to the cross product, let's stop for a minute and analyze one of its applications: obtaining the projection of a vector in a given direction.

Projecting Vectors

When one of the vectors involved in a dot product is a unit vector, this operation's result is the length of the projection of one vector over the other vector. To see why, let's use Equation 4.7. Given a vector $\vec{u}$ and a unit vector $\hat{v}$, the dot product is as follows:

$$\vec{u} \cdot \hat{v} = ||\vec{u}|| \cdot ||\hat{v}|| \cdot cos\theta = ||\vec{u}|| \cdot 1 \cdot cos\theta = ||\vec{u}|| \cdot cos\theta$$

where $||\vec{u}|| \cdot cos\theta$ is exactly the projection of $\vec{u}$ over the direction of $\hat{v}$. This will be handy for computing projections over a direction, which we could use to obtain the axial component of a force on a truss member, for example, as illustrated in Figure 4-7. In this case, we'd simply have to do $\vec{F}_a = \vec{F} \cdot \hat{u}$ to compute the axial component $\vec{F}_a$.

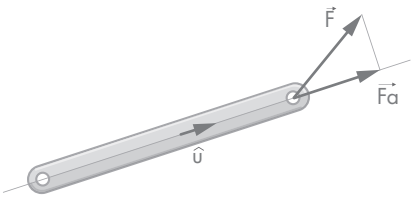

Figure 4-7: Projection of a force $\vec{F}$ in the axial direction $\hat{u}$ of a truss member

Let's implement this operation as a new method. Enter the code from Listing 4-15 into your class.

```
class Vector:
    --snip--

    def projection_over(self, direction):
        return self.dot(direction.normalized())
```

Listing 4-15: Projection of a vector over another vector

Note that the `direction` argument may not be a unit vector. To make sure our formula works, we normalize it.

Cross Product

The *cross product* of two three-dimensional vectors yields a new vector that is perpendicular to the plane containing the other two. The order of operands matters and defines the direction of the resulting vector. You can figure out the direction of the cross product using the right-hand rule. Notice that this product is therefore noncommutative, in fact: $\vec{u} \times \vec{v} = -\vec{v} \times \vec{u}$. Figure 4-8 illustrates this phenomenon.

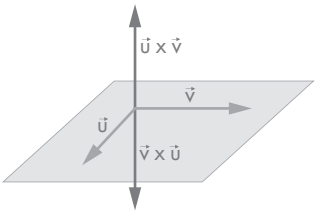

Figure 4-8: Cross products are noncommutative.

In 3D space, the cross product can be computed using Equation 4.8.

$$\vec{u} \times \vec{v} = \begin{Bmatrix} u_y \cdot v_z - u_z \cdot v_y \\ u_z \cdot v_x - u_x \cdot v_z \\ u_x \cdot v_y - u_y \cdot v_x \end{Bmatrix} \tag{4.8}$$

When working in two dimensions, every vector is contained in the same plane; thus, every cross product yields a vector perpendicular to that plane. That is easy to observe from the previous expression by simply noting that $u_z = v_z = 0$:

$$\vec{u} \times \vec{v} = \begin{Bmatrix} u_y \cdot 0 - 0 \cdot v_y \\ 0 \cdot v_x - u_x \cdot 0 \\ u_x \cdot v_y - u_y \cdot v_x \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ u_x \cdot v_y - u_y \cdot v_x \end{Bmatrix}$$

In two-dimensional applications, the cross product is therefore considered to yield a scalar value, which is the z coordinate of the previous expression's resulting vector. You can think of this coordinate as being the length

of the resulting vector. Since the x and y coordinates are zero, this magnitude given by the z coordinate is all we need to keep. Given θ as the angle between vectors $\vec{u}$ and $\vec{v}$, the cross product operation in two dimensions can be obtained by applying Equation 4.9.

$$\vec{u} \times \vec{v} = ||\vec{u}|| \cdot ||\vec{v}|| \cdot sin\theta = u_x \cdot v_y - u_y \cdot v_x \tag{4.9}$$

Let's implement the cross product. Enter the code in Listing 4-16.

```
class Vector:
    --snip--

    def cross(self, other):
        return (self.u * other.v) - (self.v * other.u)
```

Listing 4-16: Cross product

One important application of the cross product in two dimensions is determining the rotational direction of angles. From Figure 4-8 you can see that $\vec{u} \times \vec{v} > 0$, since going from $\vec{u}$ to $\vec{v}$ describes a positive (counterclockwise) angle. Conversely, going from $\vec{v}$ to $\vec{u}$ describes a negative angle resulting in a negative cross product $\vec{u} \times \vec{v} < 0$. Lastly, note that parallel vectors have a cross product of zero, which is easy to see because $sin0 = 0$. Let's take a closer look at this fact and write methods in our class that determine whether two vectors are parallel or perpendicular.

Parallel and Perpendicular Vectors

Using the dot and cross products, it's easy to test whether two vectors are parallel or perpendicular to each other. Listing 4-17 contains the code for these operations.

```
class Vector:
    --snip--

    def is_parallel_to(self, other):
        return nums.is_close_to_zero(
            self.cross(other)
        )

    def is_perpendicular_to(self, other):
        return nums.is_close_to_zero(
            self.dot(other)
        )
```

Listing 4-17: Checking whether vectors are parallel or perpendicular

Checking whether two vectors are parallel to each other is as simple as checking that their cross product is zero. Likewise, checking whether two

vectors are perpendicular is as simple as checking whether the dot product is zero. Notice we use the function `is_close_to_zero` to account for floating-point number comparison difficulties in the calculations.

Angles Between Vectors

Computing the angle between two vectors can be done with the help of the dot product expression:

$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \cdot \|\vec{v}\| \cdot cos\theta$$

Dividing the dot product term on one side by the norm product on the other and taking the inverse of the cosine of that expression, we get the following:

$$\theta = acos\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \cdot \|\vec{v}\|)}\right) \tag{4.10}$$

This expression computes only the magnitude of the angle; if we want to know the direction, we'll need to make use of the cross product. The sign of the angle can be obtained using this:

$$sgn(\vec{u} \times \vec{v})$$

where *sgn*, the sign function, is defined as follows:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0 \\ +1 & \text{if } x \geq 0 \end{cases}$$

To understand why we only get the magnitude of the angle using Equation 4.10, we need to remember an important property of the cosine function. Recall from elementary geometry that a unit vector's angle cosine is exactly the value of its horizontal projection. As you can see by inspecting the unit circle from Figure 4-9, two vectors with opposite angles (angles where the sum equals zero) get assigned the same cosine value. In other words, $cos\alpha = cos(-\alpha)$, which means that once an angle goes through the cosine function, its sign is forever lost. That makes it impossible to determine what the angle's sign is from a computed value of the dot product.

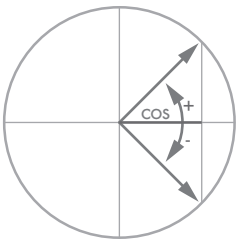

Figure 4-9: Cosines of opposite angles are equal.

For many of our applications, we'll be needing both the magnitude and sign of angles; with the help of the cross product, we can bring this informa-

tion back. Let's create two new methods, one that yields the absolute value of the angle (for those cases where the magnitude is enough) and another one that includes the sign. Enter the code in Listing 4-18 in your `Vector` class.

```
class Vector:
    --snip--

    def angle_value_to(self, other):
        dot_product = self.dot(other)
        norm_product = self.norm * other.norm
        return math.acos(dot_product / norm_product)

    def angle_to(self, other):
        value = self.angle_value_to(other)
        cross_product = self.cross(other)
        return math.copysign(value, cross_product)
```

Listing 4-18: Calculating the angle between two vectors

The first method, `angle_value_to`, computes the angle between `self` and `other` using Equation 4.10. We first obtain the dot product value and divide it by the product of norms. The angle is then the arc cosine of the result. The second method, `angle_to`, returns the value of the angle with the sign from the cross product. The `math.copysign(x, y)` function in Python returns the magnitude of `x` with the sign of `y`.

Let's try these two methods in the console. Reload it and write the following:

```
>>> from geom2d.vector import Vector
>>> u = Vector(1, 0)
>>> v = Vector(1, 1)

>>> v.angle_value_to(u)
0.7853981633974484 # result in radians

>>> v.angle_to(u)
-0.7853981633974484 # result in radians
```

Just for reference, the angle value of 0.78539... is $\pi/4$ rad (45°).

Now let's suppose we have a vector and want to create a new one by rotating the original some angle.

Rotating Vectors

Imagine that in the case of the bar subject to an external force, as we saw in Figure 4-7, we're also interested in knowing the projection of force $\vec{F}$ in the direction perpendicular to the bar. This is the force's shear component. To find the projection of the force, we first need to figure out a vector perpen-

dicular to the direction of the bar $\hat{u}$, which is obtained by rotating this vector $\pi/2$ radians, as illustrated in Figure 4-10.

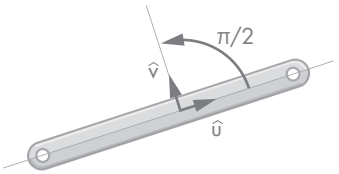

Figure 4-10: Rotating the bar's direction vector $\pi/2$ radians

A rotation preserves the length of the original vector because a rotation is a transformation that respects lengths. Assuming α is the angle that we want the vector rotated by, we can use the following expression:

$$\vec{u}\,|_{\alpha} = \begin{pmatrix} cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix} \cdot \begin{Bmatrix} u_x \\ u_y \end{Bmatrix} = \begin{Bmatrix} u_x \cdot cos\alpha - u_y \cdot sin\alpha \\ u_x \cdot sin\alpha + u_y \cdot cos\alpha \end{Bmatrix} \tag{4.11}$$

which in Python becomes the code in Listing 4-19.

```
class Vector:
    --snip--

    def rotated_radians(self, radians):
        cos = math.cos(radians)
        sin = math.sin(radians)
        return Vector(
            self.u * cos - self.v * sin,
            self.u * sin - self.v * cos
        )
```

Listing 4-19: Rotating a vector

The `rotated_radians` function returns a new vector, the result of rotating the original one by the given number of radians. Following our immutability guidelines, we never mutate the source vector; instead, we return a new one with the rotation applied.

There's one angle, $\pi/2$ rad (90 °), which is quite useful for rotating a vector. Using $\pi/2$ rad, we get a new vector perpendicular to the original one. To avoid writing `v.rotated_radians(math.pi / 2)` over and over again, we can define a new method in our `Vector` class. Knowing that $cos(\pi/2) = 0$ and $sin(\pi/2) = 1$, the angle in Expression 4.11 simplifies to the following:

$$\vec{u}\,|_{(\pi/2)} = \begin{Bmatrix} u_x \cdot 0 - u_y \cdot 1 \\ u_x \cdot 1 + u_y \cdot 0 \end{Bmatrix} = \begin{Bmatrix} -u_y \\ u_x \end{Bmatrix}$$

Let's call the method `perpendicular`. In Python, it looks like Listing 4-20.

```
class Vector:
    --snip--

    def perpendicular(self):
        return Vector(-self.v, self.u)
```

Listing 4-20: Obtaining a perpendicular vector

There's another angle we'll often use for rotations: π rad (180°). Rotating a vector π rad results in a vector that is co-linear but in the opposite direction. This time, $cos(\pi) = -1$ and $sin(\pi) = 0$. The angle expression 4.11 now looks as follows:

$$\vec{u}|_{(\pi)} = \left\{ \begin{array}{l} u_x \cdot (-1) - u_y \cdot 0 \\ u_x \cdot 0 + u_y \cdot (-1) \end{array} \right\} = \left\{ \begin{array}{l} -u_x \\ -u_y \end{array} \right\}$$

Let's call the method `opposite`. In Python, it looks like Listing 4-21.

```
class Vector:
    --snip--

    def opposite(self):
        return Vector(-self.u, -self.v)
```

Listing 4-21: Obtaining the opposite vector

These two methods, `perpendicular` and `opposite`, don't really add anything we didn't have before; we could just use `rotated_radians`. Nevertheless, they're convenient, and we'll be using them often.

Sine and Cosine

To project a vector quantity in the x- and y-axes, we use the sine or cosine values of the vector's angle, as depicted in Figure 4-11.

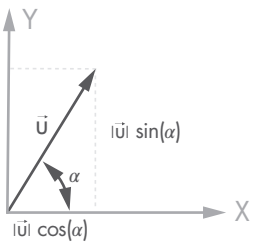

Figure 4-11: Vector projections

We'll use these to compute the stiffness matrices in global coordinates of truss structure bars in Part V of the book. The stiffness matrix of a bar is computed relative to a reference frame whose x-axis is in the direction of the bar's directrix, but we'll need to project this matrix in the direction of the global x- and y-axes to build the structure's global system of equations.

If the `Vector` class didn't provide these two properties, clients of this class could get its angle value and then compute the sine or cosine of it. Even though this is perfectly acceptable, it requires a few operations to first compute the angle and then one extra sine or cosine operation. But as you know, we can compute the sine and cosine values much more efficiently by their mathematical definition.

Say we have vector $\vec{a}$ with norm $\|\vec{a}\|$, whose projections are labeled u and v. The sine and cosine can be computed as follows:

$$\sin\theta = \frac{v}{\|\vec{a}\|} \qquad \cos\theta = \frac{u}{\|\vec{a}\|}$$

Let's implement these as attributes of the `Vector` class. Enter the code in Listing 4-22.

```
class Vector:
    --snip--

    @property
    def sine(self):
        return self.v / self.norm

    @property
    def cosine(self):
        return self.u / self.norm
```

Listing 4-22: Vector's direction sine and cosine

The implementation is straightforward given the previous expressions. Let's complete our `Point` and `Vector` classes adding the last touches.

Completing Our Classes

Our `Point` and `Vector` classes are looking good, but they're missing some small details. If we compare two instances of any of them, Python may not be able to determine whether they are equivalent; we'll fix that shortly. Also, if you remember, Python prints object instances to the console giving their class name accompanied with a memory address, which is not that helpful for us; we'll also fix this here.

Checking Equality

Try entering the following in the console (don't forget to reload it):

```
>>> from geom2d.point import Point
>>> p = Point(1, 0)
>>> p == p
❶ True
```

```
>>> q = Point(1, 0)
>>> p == q
❷ False
```

I bet ❶ didn't surprise you: a `Point` is equal to itself. What about ❷? Did you raise your eyebrows? We are comparing two points with the same coordinates, but Python states they are different. Shouldn't (1, 0) be equal to (1, 0)? It should, but first we have to teach Python how to compare two given instances of our class. By default Python considers two instances of a class to be equal if they're effectively the same instance, that is, if they live in the same memory region. To be more explicit, write this to the console:

```
>>> p
<geom2d.point.Point object at 0x10baa3f60>

>>> q
<geom2d.point.Point object at 0x10c63b438>
```

Python sees instance `p` as the one on the memory address starting at `0x10baa3f60` and instance `q` on `0x10c63b438`. Don't forget that the memory addresses of your instances will differ from these. We must instruct Python to compare our `Point` instances by checking whether the projections are close enough to be considered the same. If you recall from Table 4-1, by implementing a method called `__eq__(self, other)`, you are effectively overloading the `==` operator. Let's do this for both the `Point` and `Vector` classes.

Listing 4-23 contains the code for `Point` class (don't forget to import `nums`).

```
import math

from geom2d import nums


class Point:
    --snip--

    def __eq__(self, other):
        if self is other:
            return True

        if not isinstance(other, Point):
            return False

        return nums.are_close_enough(self.x, other.x) and \
               nums.are_close_enough(self.y, other.y)
```

Listing 4-23: Point equality implementation

Listing 4-24 contains the code for the `Vector` class.

```
import math

from geom2d import nums


class Vector:
    --snip--

    def __eq__(self, other):
        if self is other:
            return True

        if not isinstance(other, Vector):
            return False

        return nums.are_close_enough(self.u, other.u) and \
               nums.are_close_enough(self.v, other.v)
```

Listing 4-24: Implementing vector equality

As you can see, in both cases the idea is the same: comparing coordinates against another given instance. Prior to that, we do two important checks, though. The first one is to check for the case where we are comparing the same instance against itself, in which case we don't require any further comparison, so we directly return `True`. The second check is for the case where `other` is not an instance of the class. Since Python allows us to compare any two objects, we may be comparing an instance of `Vector` against a string, for example. If we detect this case where we try to compare instances from a different class, we return `False`, and we're done. You'll see this comparison pattern throughout the book, as all of our classes implementing `__eq__` will use this same approach.

To make sure we got it right, let's repeat the experiment. Don't forget to reload the console to import the last version of the code and enter the following code:

```
>>> from geom2d.point import Point
>>> p = Point(1, 0)
>>> p == p
True

>>> q = Point(1, 0)
>>> p == q
True
```

There you go! Now our `Point` and `Vector` classes comparison actually works as it is supposed to work.

String Representation

As you've seen in the console when evaluating an instance of a class, the output is not super helpful:

```
>>> from geom2d.vector import Vector
>>> v = Vector(2, 3)
>>> v
<geom2d.vector.Vector object at 0x10c63b438>
```

If we try to convert the instance to its string representation using the `str` function, we get the same result:

```
>>> str(p)
'<geom2d.vector.Vector object at 0x10c63b438>'
```

When printing the string representation of `Vector` instances to the console, we'd find something like the following much more useful:

```
>>> str(p)
'(2, 5) with norm 5.385164807134504'
```

That message has the information of the coordinate values and the value of the norm. Function `str()` in Python converts an instance of a class to its string representation. This function first checks whether the passed argument implements method `__str__`. If it does, the function calls it and returns the result. If it doesn't, the function simply returns the default string representation, which in our case is that unhelpful memory position mess.

Let's implement `__str__` in our classes. Enter Listing 4-25 inside class `Point`.

```
class Point:
    --snip--

    def __str__(self):
        return f'({self.x}, {self.y})'
```

Listing 4-25: Overriding string representation for `Point`

Then end Listing 4-26 in class `Vector`.

```
class Vector:
    --snip--

    def __str__(self):
        return f'({self.u}, {self.v}) with norm {self.norm}'
```

Listing 4-26: Overriding string representation for `Vector`

We include instance attributes into the string using *f-strings* (`f''`). The attributes are inserted between curly braces, and Python calls their `__str__` methods to get their string representation and concatenate the result. For example, you can think of the f-string, shown here:

```
f'({self.x}, {self.y})'
```

as being translated by Python to something like this:

```
"(" + str(self.x) + ", " + str(self.y) + ")"
```

Now, when using `str()` on instances of our classes, a much nicer description will be printed. Let's reload the Python terminal and give it a second try:

```
>>> from geom2d.vector import Vector
>>> v = Vector(2, 3)
>>> str(v)
'(2, 3) with norm 3.605551275463989'
```

Much better, isn't it?

Vector Factories

A *factory function* is just a function that builds an object. Factory functions are a good option for initializing objects that require some calculation. An initializer should ideally only set its class attributes and avoid any computation; for that we will use factories.

A factory function is also helpful to improve the readability of the code. For instance, if you wanted to create a `Vector` from a point P to another point Q, this code:

```
make_vector_between(p, q)
```

reads much better than this code:

```
Vector(q.x - p.x, q.y - p.y)
```

Not only that, but the latter is likely to be written many times, which should tell you there is an algorithm that needs to be abstracted into its own concept. In this particular case, the algorithm is the formula to create a vector between two ordered points (see Equation 4.12).

NOTE *A missing abstraction is a common problem. It happens when an algorithm representing a concrete concept is not properly encapsulated into its own function or class with a descriptive name. Its main hazards are that it takes longer for our brains to understand code when abstractions are not well encapsulated and that the same algorithm is copy and pasted in many places, making it difficult to maintain.*

Create a new file inside *geom2d*, call it *vectors*, and enter the code from Listing 4-27.

```
from geom2d.point import Point
from geom2d.vector import Vector


def make_vector_between(p: Point, q: Point):
    return q - p


def make_versor(u: float, v: float):
    return Vector(u, v).normalized()


def make_versor_between(p: Point, q: Point):
    return make_vector_between(p, q).normalized()
```

Listing 4-27: Vector factory functions

This file defines several functions, all of which have the purpose of creating vectors. The first function we define, `make_vector_between`, creates a vector going from a point `p` to another `q`. We've harnessed our `Point`'s class `__sub__` implementation to create the vector between the points. That is one handy way of creating vectors, expressed mathematically as shown in Equation4.12.

$$\vec{u}_{P \to Q} = \begin{Bmatrix} Q_x - P_x \\ Q_y - P_y \end{Bmatrix} \tag{4.12}$$

Next, we have a function called `make_versor`, which creates versors, or vectors of unit length. Versors are frequently used to express direction or orientation, so we'll want a convenient way of creating them. Note versors are written with a hat over them, as in $\hat{u}$, signifying their length is unitary.

Lastly, we have `make_versor_between` to create a versor between two points, which reuses the `make_vector_between` function to return the normalized result of it. The resulting versor could also be computed with Equation 4.13.

$$\hat{u}_{P \to Q} = \frac{1}{\sqrt{(Q_x - P_x)^2 + (Q_y - P_y)^2}} \cdot \begin{Bmatrix} Q_x - P_x \\ Q_y - P_y \end{Bmatrix} \tag{4.13}$$

Unit Testing

So far we've implemented a couple methods on classes `Point` and `Vector`, and we've tested some of them in the console by hand, but now we face some big questions: how can we convince someone else that our code always works as expected? How can we be sure what we've written works all the time? How can we make sure we don't break anything when we modify existing code or add new code?

Often enough, you'll need to go back to some piece of code you wrote a long time ago to fix a bug. The problem comes when you want change that code but don't know whether making that change will break what's already working. In fact, you may not be aware of what all the code is supposed to be doing, so you end up changing something you shouldn't have and break something else. This phenomenon happens so regularly it has its own name: *regression*.

Testing code by hand in the console is tiresome and boring, both ensuring you probably won't test everything you need to test. Besides that, it's not a repeatable process: you'll forget about which tests you executed for each method, or if someone else needs to run them, they'll have to figure out what to test and how. But still, we really need to make sure our changes won't break anything. Code is entirely useless if it doesn't do what it's supposed to.

What would make our life much easier is an automated test we could execute, which takes a few milliseconds to run and spits out output that clearly states whether anything went wrong, where, and why. This is the basic idea behind *unit testing*, a crucial activity for any serious developer. Your code cannot be considered finished until it's accompanied with good unit tests that prove its quality. I consider this part of development so vital I want to cover it early in the book and make extensive use of it. Writing automated, unitary tests for our code is a simple process, and there's really no excuse for not doing it.

Creating unit tests for your code is simple: create a new file, and inside it add a new class with methods that test small portions of the *test subject*. Each test case has an *assertion* function that ensures a specific result is obtained given a set of inputs. The test is considered to pass when the assertion succeeds and to fail otherwise. When the test class is executed (as we'll see next), the methods are executed, and their assertions are checked.

Don't worry if this still doesn't make sense; we're going to use unit testing so much in this book you'll get to fully understand it.

Testing Distances

The first method we wrote for `Point` was `distance_to`, so let's start our unit test adventure there. In the `geom2d` package, create a new file named *point_test.py*. Your project's structure should look like the following:

```
Mechanics
  |- geom2d
  |    |- __init__.py
  |    |- nums.py
  |    |- point.py
  |    |- point_test.py
  |    |- vector.py
  |    |- vectors.py
```

In *point_test.py*, enter the code from Listing 4-28.

```
import unittest

from geom2d.point import Point


❶ class TestPoint(unittest.TestCase):

    ❷ def test_distance_to(self):
          p = Point(1, 2)
          q = Point(4, 6)
          expected = 5
          actual = p.distance_to(q)

        ❸ self.assertAlmostEqual(expected, actual)
```

Listing 4-28: Distance between points test

We start by importing the `unittest` module, shipped with Python. This module provides us with most of the infrastructure we need to write and execute unit tests. After importing our `Point` class, we define the class `TestPoint`, which inherits `unittest.TestCase` ❶. The `TestCase` class defines a good collection of assertion methods that we gain access to inside our class when we inherit it.

Next we have the `test_distance_to` method. It's important that the method name starts with the word *test_*, because this is how the class discovers which of its methods are tests to be executed. You can define other methods in the class, but as long as their name doesn't start with *test*, they won't be executed as tests. Inside the test we create two points that we know are 5 units apart from each other and assert that their distance `p.distance_to(q)` is close to that value.

NOTE *`unittest` module's choice of words may be confusing. The name `UnitTest` is used for the class even though the tests themselves are actually the methods inside the class. Our class extending `UnitTest` is just a way of grouping related test cases.*

The assertion method `assertAlmostEqual` (defined in the class we inherited from: `unittest.TestCase`) checks for floating-point number equality with a given tolerance, which is expressed as the number of decimal positions to compare. The default number of decimal positions to check is 7, and in this test, we'll stick to the default (as we didn't provide any other value). Remember that when dealing with floating-point number comparisons, a tolerance must be used or, in this case, a given number of decimal positions (see the "Comparing Numbers" section).

There are several ways to run tests. Let's explore how to do it from both PyCharm and the console.

Running the Test from PyCharm

If you take a look at your test file in PyCharm, you'll see a little green "play" button to the left of the class and method definitions. The class button executes all the tests inside of it (so far we have only one), whereas the button next to the method will run only that one test. Click the class one; from the menu, select **Run 'Unittest for point...'**. The Run pane appears in the lower part of the IDE, and the result of executing your tests is displayed. If you did everything right, you should see the following:

```
--snip--

Ran 1 test in 0.001s

OK

Process finished with exit code 0
```

Let's now learn how to run the same test from the console.

Running the Test from the Console

IDEs other than PyCharm may have their own way to run tests. But regardless of the IDE you use, you can always run tests from the console. Open the console or terminal and make sure you're in the *Mechanics* project directory. Then run the following command:

```
$ python3 -m unittest geom2d/point_test.py
```

You should see the following result:

```
Ran 1 tests in 0.000s

OK
```

We'll run most of the tests throughout the book from the IDE, but feel free to run them from the console if you prefer.

Assertion Errors

Let's see what would've happened if the assertion detected a wrong result. Inside *point_test.py*, change the expected value for the distance:

```
expected = 567
actual = p.distance_to(q)
```

This assertion is expecting points $(1, 2)$ and $(4, 6)$ to be 567 units apart, which is totally wrong. Now, execute the test again by clicking the green play button beside the class. This is the result you should see:

```
Ran 1 test in 0.006s
```

```
FAILED (failures=1)

Failure
Traceback (most recent call last):
  ...snip...
  File ".../geom2d/tests/point_test.py", line 14, in test_distance_to
    self.assertAlmostEqual(expected, actual)
  ...snip...

AssertionError: 567 != 5.0 within 7 places (562.0 difference)
```

The message with the most valuable information is the last one. It's telling us that there was an assertion error; that is, the assertion failed when it found 5.0 where 576 was expected. It used 7 decimal places in the comparison and still found a difference of 562.

Before this assertion error is the *traceback*, the execution path Python took until it got the error. As the message states, calls closer to the failure appear last in the list. As you can see, the test execution failed in file *point_test.py* (no surprise) on line 14 (yours may be different), in a test named `test_distance_to`. This information will prove invaluable when you modify existing code and run the tests only to find out whether a test fails, as it can tell you what exactly broke. These test failure messages will give you precise information.

Don't forget to put our unit test back to how we initially wrote it and make sure it still runs successfully.

Testing Vector Plus and Minus Operations

To ensure + and − operations work properly for vectors (doing the same for the `Point` class is left as an exercise for you), let's use the following test cases:

$$\begin{Bmatrix} 1 \\ 2 \end{Bmatrix} + \begin{Bmatrix} 4 \\ 6 \end{Bmatrix} = \begin{Bmatrix} 5 \\ 8 \end{Bmatrix}$$

and

$$\begin{Bmatrix} 1 \\ 2 \end{Bmatrix} - \begin{Bmatrix} 4 \\ 6 \end{Bmatrix} = \begin{Bmatrix} -3 \\ -4 \end{Bmatrix}$$

Create a new file inside package *geom2d* for testing the `Vector` class. Name it *vector_test* and enter the code from Listing 4-29.

```
import unittest

from geom2d.vector import Vector


class TestVector(unittest.TestCase):
    u = Vector(1, 2)
    v = Vector(4, 6)
```

```
    def test_plus(self):
        expected = Vector(5, 8)
        actual = self.u + self.v
        self.assertEqual(expected, actual)

    def test_minus(self):
        expected = Vector(-3, -4)
        actual = self.u - self.v
        self.assertEqual(expected, actual)
```

Listing 4-29: Tests for plus and minus operations

Run all tests using the green play button to the left of the class definition. If you got everything right, your two new tests should succeed. Yay! Our operations were properly implemented. The nice thing is, if there had been a bug in the implementation, these tests would have pointed out where and why.

It's worth noting that this time we're using assertion method `assertEqual`, which under the hood compares both arguments using the `==` operator. If we hadn't overloaded this operator in the `Vector` class, the tests would fail even if the results were right. Try this: comment out the `__eq__(self, other)` method definition (by appending a `#` character at the beginning of the line) in the `Vector` class and rerun the tests. You'll find how the last two tests fail with a message like the following:

```
<geom2d.vector.Vector object at 0x10fd8d198> !=
<geom2d.vector.Vector object at 0x10fd8d240>

Expected :<geom2d.vector.Vector object at 0x10fd8d240>
Actual   :<geom2d.vector.Vector object at 0x10fd8d198>
```

Familiar? That's Python assuming two objects from the class can be equal only if they are the same actual object living in the same memory position. Our `__eq__` operator overload explains to Python the rules to determine when two objects should be considered the same. Don't forget to uncomment the method.

Testing Vector Product Operations

Let's add two new test cases for dot and cross products using the same two vectors defined in the test class:

$$\begin{Bmatrix} 1 \\ 2 \end{Bmatrix} \cdot \begin{Bmatrix} 4 \\ 6 \end{Bmatrix} = 4 + 12 = 16$$

and

$$\begin{Bmatrix} 1 \\ 2 \end{Bmatrix} \times \begin{Bmatrix} 4 \\ 6 \end{Bmatrix} = 6 - 8 = -2$$

In code, this looks like Listing 4-30.

```
import unittest

from geom2d.vector import Vector


class TestVector(unittest.TestCase):

    --snip--

    def test_dot_product(self):
        expected = 16
        actual = self.u.dot(self.v)
        self.assertAlmostEqual(expected, actual)

    def test_cross_product(self):
        expected = -2
        actual = self.u.cross(self.v)
        self.assertAlmostEqual(expected, actual)
```

Listing 4-30: Tests vector dot and cross products

Run all test cases to make sure the new ones also succeed. Note that, as we're comparing numbers again, we use assertion method `assertAlmostEqual`.

Testing Vector Parallelism and Perpendicularity

Next we'll test the `is_parallel_to` and `is_perpendicular_to` methods. We'll do parallel first. In this case, as we're checking a Boolean expression, we want to have two tests, one checking that the two vectors are parallel (a positive test) and another one checking whether they're not (a negative test). For the positive case, we'll rely on the fact that a vector is always parallel to itself. Enter the code in Listing 4-31 inside class `TestVector`.

```
import unittest

from geom2d.vector import Vector


class TestVector(unittest.TestCase):

    --snip--

    def test_are_parallel(self):
        self.assertTrue(self.u.is_parallel_to(self.u))

    def test_are_not_parallel(self):
```

```
        self.assertFalse(self.u.is_parallel_to(self.v))
```

Listing 4-31: Testing vector parallelism

There are two new assertion methods in this listing that are interesting ones: `assertTrue`, which checks whether a given expression evaluates to `True`; and `assertFalse`, which checks whether a given expression evaluates to `False`.

We'll follow the same pattern for checking perpendicularity. After the last two tests, enter the two in Listing 4-32.

```
import unittest

from geom2d.vector import Vector


class TestVector(unittest.TestCase):

    --snip--

    def test_are_perpendicular(self):
        perp = Vector(-2, 1)
        self.assertTrue(self.u.is_perpendicular_to(perp))

    def test_are_not_perpendicular(self):
        self.assertFalse(self.u.is_perpendicular_to(self.v))
```

Listing 4-32: Testing vector perpendicularity

Run all tests inside class `TestVector` to make sure they succeed. Congratulations! You've implemented your first unit tests. These tests will ensure the methods in our geometry classes work as expected. Additionally, if you found a better implementation for any of the methods we covered with tests, to make sure it still works as expected, just run its tests. Tests also serve to document the expected behavior of your code. If at some point you need a reminder about what the code you wrote is supposed to do in a particular case, unit tests should help.

Writing good tests is not a simple endeavor. One gets good at it by writing many, but there are some guidelines we can follow that will help us. Let's take a look at three simple rules that will make our tests much more resilient.

Three Golden Rules for Unit Testing

We've covered tests for a small fraction of the methods from the `Point` and `Vector` classes. Now that you have the required knowledge, try testing all the methods that we wrote in both the `Point` and `Vector` classes. I'll leave this for you as an exercise, but you can take a look at the code provided with the book if you need help: it includes a lot of unit tests. Look for all the methods we didn't test and write the tests you think are needed to make sure they

work properly. I encourage you to try, but if you still feel like unit testing is foreign to you, don't worry, we'll be writing unit tests in some other chapters of this book.

As mentioned, I believe writing unit tests is an integral part of coding, and handing software not covered by unit tests should be considered a poor practice. Moreover, writing code for the *open source* community requires good unit tests. You've got to give the community a reason to believe what you did actually works. Proving this with automated tests that anybody can easily run and see for themselves is always a good approach, as it's unlikely anybody is going to take the time to think about how to test your code and then open the console and manually try it all.

You'll get better at writing reliable unit tests as you practice. For now, I'd like to give you some basic rules to follow. Don't expect to fully grasp their meanings now, but come back to this section from time to time as you move through the book.

Rule 1: One Reason to Fail

Unit tests should have one and only one reason to fail. This sounds simple, but in many cases the *test subject* (what you are testing) is complex and made up of several components working together.

If tests fail for only one reason, it's straightforward to find the bug in the code. Imagine the opposite: a test that could fail for, say, five different reasons. When that test fails, you'll find yourself spending too much time reading error messages and debugging code, trying to understand what made it fail this particular time.

Some developers and test professionals (testing is a profession on its own, which I spent several years doing) state that each test should have one and only one assertion. Being pragmatic, sometimes having more than one assertion is not that harmful, but if it's one, that's much better.

Let's analyze a particular case. Take the test we wrote for checking whether two vectors are perpendicular. If instead of this:

```
def test_are_perpendicular(self):
    perp = Vector(-2, 1)
    self.assertTrue(self.u.is_perpendicular_to(perp))
```

we had written this:

```
def test_are_perpendicular(self):
    perp = u.perpendicular()
    self.assertTrue(self.u.is_perpendicular_to(perp))
```

then the test could fail because of an error in the `is_perpendicular_to` method or because of an error in the implementation of `perpendicular`, which we use to compute a perpendicular vector to $\vec{u}$. See the difference?

Rule 2: Controlled Environment

We use the word *fixture* to refer to the environment where a test runs. The environment includes all pieces of data surrounding our test and the state of the test subject itself, all of which may alter the results of the test. This rule states that you should have total control of the fixture where your test runs. Inputs and expected outputs of the test should always be known beforehand. Everything happening inside your tests should be *deterministic*; that is, there should be no randomness or dependence on anything out of your control: dates or times, operating systems, machine environment variables not set by the test, and so on.

If your tests seem to fail at random, they are useless, and you should get rid of them. People get used to random failing tests fast and start ignoring them. The problem comes when they also ignore tests that are failing because of a bug in the code.

Rule 3: Test Independence

Tests should never depend on other tests. Each test should run on its own and never depend on a fixture set by other tests.

There are at least three reasons for this. First, you'll want to run or debug tests independently. Second, many test frameworks do not guarantee the execution order of tests. Finally, it's much simpler to read and understand tests that don't depend on other surrounding tests.

Let's illustrate this with the test class in Listing 4-33.

```
class TestSwitch(unittest.TestCase):

    switch = Switch()

    def test_switch_on(self):
     self.switch.on()
     self.assertTrue(self.switch.is_on())

    def test_switch_off(self):
     # Last test should have switched on
     self.switch.toggle()
     self.assertTrue(self.switch.is_off())
```

Listing 4-33: Test depending on another test

See how `test_switch_off` depends on `test_switch_on`? By using a method called `toggle`, we could get the wrong result if the tests run in a different order and the switch has a state of *off* when this test runs.

Never rely on test execution order; that's results in trouble. Tests should always run independently: they should work the same way no matter the order of execution.

Summary

In this chapter, we created two important classes: `Point` and `Vector`. The rest of our *geom2d* library will be built upon these simple but powerful abstractions. We taught Python how to determine whether two given instances of `Point` or `Vector` are logically equal by implementing the special method `__eq__`, and we provided a better textual representation with `__str__`. We covered some of the methods in these classes with unit tests, and I encouraged you to extend the coverage on your own. The best way to learn to write good unit tests is by practicing. In the next chapter, we'll add two new geometrical abstractions to *geom2d*: lines and segments. These provide a new dimension that can be used to construct more complex shapes.