Lecture 27

Object-Oriented Programming Review and List Comprehension

15 Floréal, CCXXX

Song of the day: ***AXIOM*** *by Ai Furihata (2021).*

Part 0: *(Vector) Space Oddity*

In physics and mathematics, a **vector** is a geometric object that has a magnitude (or length) and direction.

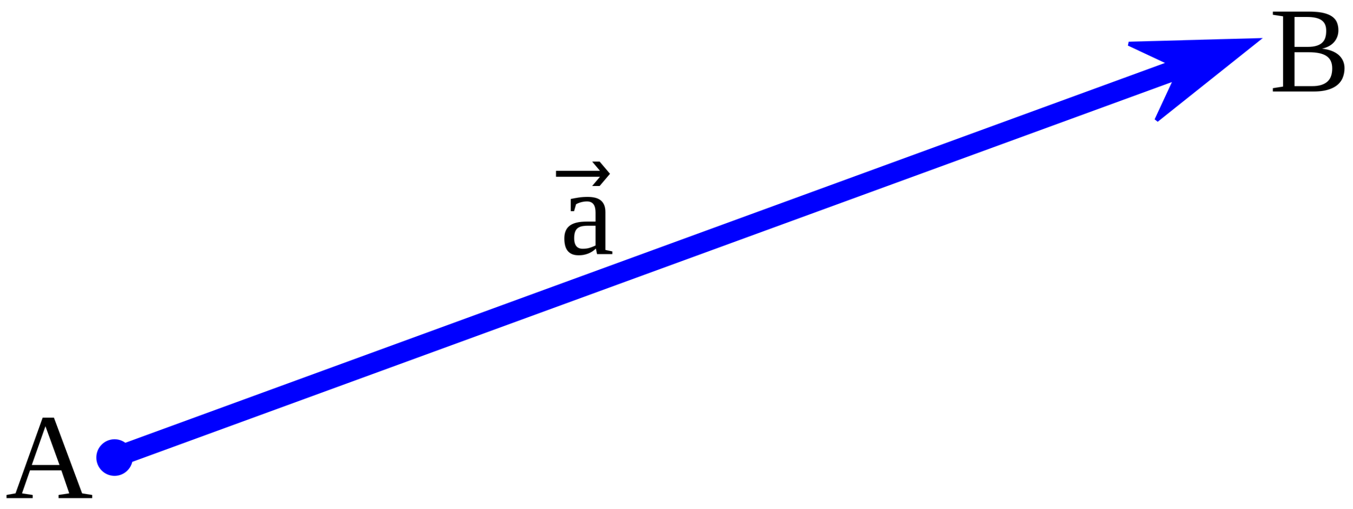

Figure 1: A two-dimensional vector pointing from point `A` to `B`. Each point has an x-coordinate and a y-coordinate.

These vectors are often used to represent forces acting on an object. For example, if one person is pushing a box the north direction, and another person is pushing the same box in the east direction, the resulting **net force** will cause the box to move somewhere in the northeast direction, with the combined forces of both pushes. In other words, you can perform *arithmetic on vectors*.

We will be simulating **three-dimensional** vectors using classes.

The `Point` *Class*

Since each vector is delineated by two points in 3-dimensional space, we will create a class to represent such a point.

In the file **point.py**, create a class called `Point` that will accept 3 argument when being instantiated: an x-coordinate, a y-coordinate, and a x-coordinate. These will also be the `Point` objects' arguments:

```
point_a = Point(-2, -3, 0)

print(point_a.x_coord)
print(point_a.y_coord)
print(point_a.z_coord)

print(point_a)
```

Output:

```
-2
-3
0
(-2, -3, 0)
```

Note the behaviour when `Point` objects are printed. Make sure your class behaves the same way.

`Point` objects must be able to subtract from each other. For example:

```
point_a = Point(3.0, -0.67, -6)
point_b = Point(34.0, -5.67, -6.06)
point_c = point_b - point_a

print(point_c)
```

Output:

```
(31.0, -5.0, -0.06)
```

Each of these parameters **must** have a default value of `0.0` .

The `Vector` *Class*

In the file **vector.py**, define a class called `Vector` . `Vector` objects will be created by passing in two `Point` objects as arguments:

```
from point import Point
from random import randrange

# Creating points with random coordinates
point_a = Point(randrange(-10, 10), randrange(-10, 10), randrange(-10, 10))
point_b = Point(randrange(-10, 10), randrange(-10, 10), randrange(-10, 10))

vector_a = Vector(point_b, point_a)
```

Each `Vector` object will only have one attribute, `vector` , whose value will be the difference between the second `Point` parameter and the first `Point` parameter. For example, if the first parameter was a point at the origin `(0, 0, 0)` and the second parameter was the point `(10.3, 0.0, -3.4)` , the following code:

```
vector_a = Vector(Point(), Point(10.3, 0.0, -3.4))
```

```
print(vector_a.vector)
```

Would print:

```
`(10.3, 0.0, -3.4)`
```

When printing `Vector` objects, make sure they look as follows:

```
vector_a = Vector(Point(), Point(10.3, 0.0, -3.4))

print(vector_a)
```

Output:

```
10.3x + 0.0y - 3.4z
```

Note here that the signs change with the value of the x-, y-, and z-coordinates.

Once you have gotten your initializer and printing behaviour to work, define a method for the `Vector` class called `get_magnitude()` which will simply return the value of this vector's magnitude. The magnitude of a vector `v`, denoted by the `|v|` notation, is calculated using the following formula:

> $|v| = (x^2 + y^2 + z^2)^{0.5}$

```
vector_a = Vector(Point(2, 45, 0.0), Point(10.3, 0.0, -3.4))

print(vector_a.get_magnitude())
```

Output:

```
45.88518279357727
```

Finally, make sure your `Vector` objects can **multiply**. For this, we will use the **dot product**:

> $a \cdot b = (x_a * x_b)\ x + (y_a * y_b)\ y + (z_a * z_b)\ z$

```
# Creating vector A with two points of random coordinates
point_a = Point(randrange(-10, 10), randrange(-10, 10), randrange(-10, 10))
point_b = Point(randrange(-10, 10), randrange(-10, 10), randrange(-10, 10))
vector_a = Vector(point_b, point_a)

# Creating vector B with two points of random coordinates
point_c = Point(randrange(-10, 10), randrange(-10, 10), randrange(-10, 10))
```

```
point_d = Point(randrange(-10, 10), randrange(-10, 10), randrange(-10, 10))
vector_b = Vector(point_d, point_c)

print("Vector A: {}".format(vector_a))
print("Vector B: {}".format(vector_b))

dot_product = vector_a * vector_b

print("A · B = {}".format(dot_product))
```

Possible output:

```
Vector A: -15x - 2y - 7z
Vector B: 2x + 5y + 7z
A · B = -30.0x - 10.0y - 49.0z
```

Part 1: *List Comprehension*

We often use `for` -loops to create lists. For instance, if we wanted a list of all the lower-case letters of the English alphabet, we would do:

```
alphabet = []

for ascii_code in range(ord('a'), ord('z') + 1):
    alphabet.append(chr(ascii_code))
```

Or perhaps for creating a list with specific requirements:

```
upper_limit = 101
divisible_by_5_and_7 = []

for number in range(upper_limit):
    if number % 5 == 0 or number % 7 == 0:
        divisible_by_5_and_7.append(number)
```

Or even wanting to create lists based on other lists:

```
names = ["Chopin", "Debussy", "Ravel", 123, None, False, "Saint-Saëns"]
uppercase_names = []

for name in names:
    if type(name) == str:
        uppercase_names.append(name.upper())
```

These are relatively common operations, and having to write multiline loops for such simple processes can clutter up and even obfuscate one's code. To that end, some languages (like Python) have this brilliant little thing called **list comprehensions**. The way I like to think of them is as "one-liner" `for` -loops that **create lists**. The general format for list comprehensions is as follows:

```
list_variable = [expression for loop_variable in iterable]
```

That is:

> Give me a **list** whose components will be the result of a certain **expression** for every **element (loop variable)** in an **iterable**.

Let's take a look at the above three problems and convert them into list comprehensions:

```
# Getting a list of lowercase letters in the alphabet

alphabet = [chr(ascii_code) for ascii_code in range(ord('a'), ord('z') + 1)]
```

Here, the expression is `chr(ascii_code)` —that is, the character equivalent to the loop variable value `ascii_code`. The iterable, of course, is `range(ord('a'), ord('z') + 1)`.

If you have a certain condition to an expression being added as a member of your list, you can add it at the end of the list comprehension as such:

```
# Get list of numbers from 0 to 100 IFF they are divisible by both 5 and 7

limit = 101

divisible_by_5_and_7 = [number for number in range(limit) if number % 5 and number % 7]
```

So this expands our general form to the following:

```
list_variable = [expression for loop_variable in iterable if condition]
```

That is:

> Give me a **list** whose components will be the result of a certain **expression** for every **element (loop variable)** in an **iterable** if it meets a certain **condition**.

Finally:

```
# Get a list of uppercased names if they are strings

names = ["Chopin", "Debussy", "Ravel", 123, None, False, "Saint-Saëns"]

uppercase_names = [name.upper() for name in names if type(name) == str]
```

List comprehensions are super convenient for things like these because they are concise and easy to read. However, as the name implies, they are generally used only to create lists. You can *technically* use them to execute a certain function for every element in an iterable. For instance, if all you wanted to do is print the elements of a list, you *could* do the following:

```
lst = ["This", "is", "a", "weird", "flex.\n"]
[print(element, end=" ") for element in lst]
```

Output:

```
This is a weird flex.
```

Which, I mean, *works*, but it looks kind of silly. It's kind of the equivalent of using a screwdriver to crack an egg open—sure, you can do it, but that's not really what a screwdriver is for. Using them this way might actually end up confusing people more than a regular `for` -loop might. And that's exactly the opposite of what list comprehensions are for.

If you're having trouble coming up with a list comprehension from scratch, try doing it with a `for` -loop first, and then see if you can fit the parts into the general `[expression for loop_variable in iterable if condition]` form. List comprehensions often help, but there's no reason to force them where you feel like they might not belong.

By the way, you can also do the equivalent of a nested `for` -loop with comprehensions. Let's say we wanted the multiplication tables from 1 to 10:

```
# Using a for-loop
table = []

for row_number in range(1, 11):
    row = []
    for col_number in range(1, 11):
        row.append(row_number * col_number)

    table.append(row)

# Using list comprehension
table = [[row_number * col_number for col_number in range(1, 11)] for row_number in range(1, 11)]
```

Part 2: *Dictionary Comprehension*

While ***this won't be on the final***, I just want to make you aware that you can also do this to create dictionaries.

The general form is as follows:

```
new_dict = { key_expr: value_expr for loop_variable in iterable }
```

Let's see an example of this. If we had a list of tuples where the first element was the country, and the second was the name of its capital, we could form a dictionary where the key is the name of the country, and the value is the name of the capital:

```
map_info = [("Mexico", "Mexico City"), ("France", "Paris"), ("Lisboa", "Portugal"), ("Tokyo", "Japan"), ("Alg

capitals = { country[0]: country[1] for country in map_info }

print(capitals)
```

Output:

```
{'Mexico': 'Mexico City', 'France': 'Paris', 'Lisboa': 'Portugal', 'Tokyo': 'Japan', 'Algeria': 'Algiers'}
```