Programming Level-up

Lecture 2 - More advanced Python & Classes

Jay Morgan

2021-10-01

Outline

Setting up a proxy in Linux – environment variables

Environment variables are variables that are set in the Linux environment and are used to configure some high-level details in Linux.

The command to create/set an environment is:

```
export VARIABLE_NAME=""
```

Exporting a variable in this way will mean `VARIABLE_NAME` will be accessible while you're logged in. Every time you log in you will have to set this variable again.

Setting up a proxy in Linux – univ-tln specific

In the université de Toulon, you're required to use the university's proxy server to access the internet. Therefore, in Linux at least, you will have to tell the system where the proxy server is with an environment variable.

```
export HTTP_PROXY='<username>:<password>@proxy.univ-tln.fr
export HTTPS_PROXY='<username>:<password>@proxy.univ-tln.f
export FTP_PROXY='<username>:<password>@proxy.univ-tln.fr:
```

Setting up a proxy in the .bashrc

If you don't wish to set the variable every time log in, you should enter the same commands into a `.bashrc` in your home directory.

```
export HTTP_PROXY='...'
export HTTPS_PROXY='...'
export FTP_PROXY='...'
```

When you log in, the `.bashrc` file will be run and these variables will be set for you.

List comprehensions

We have seen previously how `for` loops work. Knowing the syntax of a `for` loop and wanting to populate a list with some data, we might be tempted to write:

```
x = []
for i in range(3):
    x.append(i)

print(x)
```

```
Results:
# => [0, 1, 2]
```

While this is perfectly valid Python code, Python itself provides 'List comprehensions' to make this process easier.

```
x = [i for i in range(3)]
```

List comprehensions – syntax

The syntax of a list comprehensions is:

```
[ <variable> for <variable> in <iterable> ]
```

We can also perform similar actions with a dictionary

```
[ <key>, <value> for <key>, <value> in <dictionary.items()
```

List comprehensions – dictionary

Python doesn't restrict us to list comprehensions, but we can do a similar operation to create a dictionary.

```
x = [2, 5, 6]
y = {idx: val for idx, val in enumerate(x)}
print(y)
```

```
Results:
# => {0: 2, 1: 5, 2: 6}
```

Here, every item in `x` has been associated with its numerical index as a key thanks to the `enumerate` function that returns both the index and value at iteration in the for loop.

List comprehensions – using if's

Perhaps we only want to optionally perform an action within the list comprehension? Python allows us to do this with the inline `if` statement we've seen in the previous lecture.

```
x = [i if i < 5 else -1 for i in range(7)]
print(x)
```

```
Results:
# => [0, 1, 2, 3, 4, -1, -1]
```

We add the inline `<var> if <condition> else <other-var>` before the `for` loop part of the comprehension.

List comprehension – using `if`'s

There is another type of `if` statement in a list comprehension, this occurs when we don't have an `else`.

```
x = [i for i in range(7) if i < 3]
print(x)
```

```
Results:
# => [0, 1, 2]
```

In this example, we're only 'adding' to the list if the condition ($i < 3$) is true, else the element is not included in the resulting list.

List comprehensions – multiple `for`'s

If we like, we can also use nested for loops by simply adding another for loop into the comprehension.

```
x = [(i, j) for i in range(2) for j in range(2)]

print(x)
```

```
Results:
# => [(0, 0), (0, 1), (1, 0), (1, 1)]
```

In this example, we're creating a tuple for each element, effectively each combination of 1 and 0.

Quick Exercise – List comprehension

- Using list comprehension, create a list of even numbers from 6 to 20, and assign this list to the variable named `even_numbers`.
- Create a new variable called `even_numbers_dict`, create a dictionary using the comprehension syntax. The keys of the dictionary should be the index of each element in `even_numbers`, while the value should be the even number.
- What is the 5th even number?

Dealing with Errors

When programming, its good to be defensive and handle errors gracefully. For example, if you're creating a program, that as part of its process, reads from a file, its possible that this file may not exist at the point the program tries to read it. If it doesn't exist, the program will crash giving an error such as: `FileNotfoundError`.

Perhaps this file is non-essential to the operation of the program, and we can continue without the file. In these cases, we will want to appropriately catch the error to prevent it from stopping Python.

Try-catch

Try-catches are keywords that introduce a scope where the statements are executed, and if an error (of a certain type IndexError in this example) occurs, different statements could be executed.

In this example, we are trying to access an element in a list using an index larger than the length of the list. This will produce an `IndexError`. Instead of exiting Python with an error, however, we can catch the error, and print a string.

```
x = [1, 2, 3]

try:
    print(x[3])
except IndexError:
    print("Couldn't access element")
```

```
Results:
# => Couldn't access element
```

Try-catch – capturing messages

If we wanted to include the original error message in the print statement, we can use the form:

```
except <error> as <variable>
```

This provides us with an variable containing the original error that we can use later on in the try-catch form.

```
x = [1, 2, 3]

try:
    print(x[3])
except IndexError as e:
    print(f"Couldn't access elements at index beacuse: {e}
```

```
Results:
# => Couldn't access elements at index beacuse: list index
```

Types of exceptions

There are numerous types of errors that could occur in a Python. Here are just some of the most common.

- IndexError – Raised when a sequence subscript is out of range.
- ValueError – Raised when an operation or function receives an argument that has the right type but an inappropriate value
- AssertionError – Raised when an assert statement fails.
- FileNotFoundError – Raised when a file or directory is requested but doesn't exist.

The full list of exceptions in Python 3 can be found at: `https://docs.python.org/3/library/exceptions.html`

Assertions

One of the previous errors (`AssertionError`) occurs when an assert statement fails. Assert is a keyword provided to test some condition and raise an error if the condition is false. It typically requires less code than an `if`-statement that raises an error, so they might be useful for checking the inputs to functions, for example:

```
def my_divide(a, b):
    assert b != 0
    return a / b

my_divide(1, 2)
my_divide(1, 0)
```

Here we are checking that the divisor is not a 0, in which case division is not defined.

More on lists

In a previous lecture, we found that we can add `.append()` to the end of a variable of a type list to add an element to the end of the list. Lists have many more methods associated with them that will be useful when programming in Python.

Lists have a number of other convenient functions[1].

Some of these include:

```
my_list.insert(0, "dog")   # insert "dog" at index 0
my_list.count(2)           # count the number of times 2 ap
my_list.reverse()          # reverse the list
```

[1] https://docs.python.org/3/tutorial/datastructures.html

More on sets – union

Sets, while containing only unique elements, have a number of useful functions to perform certain set operations. Take for example the union (elements that are in either sets) of two sets:

```
x = set([1, 2, 3, 4, 5])
y = set([5, 2, 6, -1, 10])

print(x.union(y))
```

```
Results:
# => {1, 2, 3, 4, 5, 6, 10, -1}
```

The syntax of using these methods follows:

```
<set_1>.function(<set_2>)
```

More on sets – intersection

Or the intersection (the elements that are in both) of two sets:

```
x = set([1, 2, 3, 4, 5])
y = set([5, 2, 6, -1, 10])

print(x.intersection(y))
```

```
Results:
# => {2, 5}
```

More on sets – set difference

And the set difference (the elements that in set 1, but not in set 2):

```
x = set([1, 2, 3, 4, 5])
y = set([5, 2, 6, -1, 10])

print(x.difference(y))
```

```
Results:
# => {1, 3, 4}
```

More on set – subsets

We can even return a boolean value if set 1 is a subset of set 2:

```
x = set([1, 2, 3, 4, 5])
y = set([5, 2, 6, -1, 10])
z = set([1, 2, 3, 4, 5, 6, 7])

print(x.issubset(y))
print(x.issubset(z))
```

```
Results:
# => False
# => True
```

For a full list of what methods are available with sets, please refer to:
https://realpython.com/python-sets/#operating-on-a-set

Better indexing – slices

If we wanted to access an element from a data structure, such as a list, we would use the `[ ]` accessor, specifying the index of the element we wish to retrieve (remember that indexes start at zero!). But what if we ranted to access many elements at once? Well to accomplish that, we have a slice or a range of indexes (not to be confused with the `range` function). A slice is defined as:

```
start_index:end_index
```

where the `end_index` is non inclusive – it doesn't get included in the result. Here is an example where we have a list of 6 numbers from 0 to 5, and we slice the list from index 0 to 3. Notice how the 3rd index is not included.

```
x = [0, 1, 2, 3, 4, 5]
print(x[0:3])
```

```
Results:
# => [0, 1, 2]
```

Better indexing – range

When we use `start_index:end_index`, the slice increments by 1 from `start_index` to `end_index`. If we wanted to increment by a different amount we can use the slicing form:

```
start_index:end_index:step
```

Here is an example where we step the indexes by 2:

```
x = list(range(100))
print(x[10:15:2])
```

```
Results:
# => [10, 12, 14]
```

Better indexing – reverse

One strange fact about the step is that if we specify a negative number for the step, Python will work backwards, and effectively reverse the list.

```
x = list(range(5))

print(x[::-1])
```

```
Results:
# => [4, 3, 2, 1, 0]
```

Better indexing – range

In a previous example, I created a slice like `0:3`. This was a little wasteful as we can write slightly less code. If we write `:end_index`, Python assumes and creates a slice from the first index (0) to the `end_index`. If we write `start_index:`, Python assumes and creates a slice from `start_index` to the end of the list.

```
x = list(range(100))

print(x[:10])
print(x[90:])
```

```
Results:
# => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# => [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Better indexing – backwards

Finally, we also work backwards from the end of list. If we use a negative number, such as -1, we are telling Python, take the elements from the end of the list. -1 is the final index, and numbers lower than -1 work further backwards through the list.

```
x = list(range(100))

print(x[-1])
print(x[-2])
```

```
Results:
# => 99
# => 98
```

Better indexing –backwards

Slicing with negative indexes, also works. Here we are creating a slice from the end of the list - 10, to the last (but not including) index.

```
x = list(range(100))

print(x[-10:-1])
```

```
Results:
# => [90, 91, 92, 93, 94, 95, 96, 97, 98]
```

Quick Exercise – Slicing

- Create a list of elements from 0 to 100, of every 3rd number (e.g. use a range with a step).
- First, slice the first 5 indexes.
- Second, get the last 10 indexes.
- Third, get the 50th to 55th (inclusive) indexes.
- Challenge get the last 10 indexes, but only using positive indexes up to 10.

Formatting strings

In many previous examples when we've printed strings, we've done something like:

```python
age = 35

print("The value of age is", age)
```

```
Results:
# => The value of age is 35
```

While this works in this small context, it can get pretty cumbersome if we have many variables we want to print, and we also want to change how they are displayed when they are printed.

We're going to take a look now at much better ways of printing.

Better ways of printing strings - %

The first method is using %. When we print, we first construct a string with special delimiters, such as %s that denotes a string, and %d that denotes a number. This is telling Python where we want the values to be placed in the string.

Once we've created the string, we need to specify the data, which we do with `% (...)`. Like, for example:

```
age = 35
name = "John"

print("%d years old" % age)  # no tuple for one variable
print("%s is %d years old" % (name, age))
```

```
Results:
# => 35 years old
# => John is 35 years old
```

Here we are specifying the a string %s and number %d, and then giving the variables that correspond with that data type.

Better ways of printing strings – data specifiers

The special delimiters correspond with a data type. Here are some of the most common:

- `%s` – For strings
- `%d` – For numbers
- `%f` – For floating point numbers.

There are others such as `%x` that prints the hexadecimal representation, but these are less common. You can find the full list at: `https://docs.python.org/3/library/stdtypes.html#old-string-formatting`

Better ways of printing strings – floating points

When using these delimiters, we can add modifiers to how they format and display the value. Take a very common example, where we have a floating point value, and, when printing it, we only want to print to 3 decimal places. To accomplish this, we again use `%f` but add a `.3` to between the % and `f`. In this example, we are printing π to 3 decimal places.

```
print("Pi to 3 digits is: %.3f" % 3.1415926535)
```

```
Results:
# => Pi to 3 digits is: 3.142
```

Better ways of printing strings – floating points

In the previous example, we used `.3` to specify 3 decimal places. If we put a number before the decimal, like `10.3` we are telling Python *make this float occupy 10 spaces and this float should have 3 decimal places printed*. When it gets printed, you will notice that it shifts to the right, it gets padded by space. If we use a negative number in front of the decimal place, we are telling python to shift it to the left.

```
print("Pi to 3 digits is: %10.3f" % 3.1415926535)
print("Pi to 3 digits is: %-10.3f" % 3.1415926535)
```

```
Results:
# => Pi to 3 digits is:      3.142
# => Pi to 3 digits is: 3.142
```

Quick Exercise – printing with %

- Creating a dictionary containing the following information:

Key	Value
name	Jane
age	35
lon	-3.52352
lat	2.25222

- Print (using the % operator) the values of this dictionary so that the result looks like: "Jane (located at -3.5, 2.2) is 35 years old"

Better ways of printing strings – `.format()`

Another way of performing 'string interpolation' where the values associated with variables are printed with strings is accomplished using the `.format()` method.

To use this method, create a string with `{}` delimiters, and after the string, call the `.format()` method, where the arguments to this method are the values you want to include in the string. The number of values passed to `.format()` should be the same as the number of `{}` in the string.

```
name = "Jane"
age = 35

print("{} is {} years old".format(name, age))
```

```
Results:
# => Jane is 35 years old
```

Better ways of printing strings – .format()

To be more explicit and clear with which values go where in the string, we can name them by putting some same into the {} tokens. When we call the .format() function, we then use the same name as named parameters.

```
name = "Jane"
age = 35

print("{the_name} is {the_age} years old".format(the_name=n
                                                 the_age=a
```

```
Results:
# => Jane is 35 years old
```

Better ways of printing strings – alignment

`.format()` allows us to some quite complicated things with the display of strings. Take this for example where we are setting the alignment of the values.

The syntax of formatting strings can be a language of it's own right! So we won't go too deep into it here. However, you can find all you need to know about formatting here: `https://docs.python.org/3/library/string.html#format-string-syntax`

```
print("|{:<10}|{:^10}|{:>10}|".format('all','dogs','bark')
print("-" * 34)

Results:
# => |all       |   dogs   |      bark|
# => ----------------------------------
```

Better ways of printing strings – f-strings

The final method of formatting strings is a newcomer within the language, it is the so-called `f-string`. Where a `f` character is prefixed to the beginning of the string you're creating. `f-string`'s allow you to use Python syntax within the string (again delimited by `{}`.

Take this for example where we are referencing the variables `name` and `age` directly.

```
name = "Jane"
age = 35

print(f"{name} is {age} years old")
```

```
Results:
# => Jane is 35 years old
```

Better ways of printing strings – f-strings

f-string's allow you to execute Python code within the string. Here we are accessing the value from the dictionary by specifying the key within the string itself! It certainly makes it a lot easier, especially if we only need to access the values for the string itself.

```
contact_info = {"name": "Jane", "age": 35}

print(f"{contact_info['name']} is {contact_info['age']} ye
```

```
Results:
# => Jane is 35 years old
```

https://pyformat.info/

Better ways of printing strings – `f-string`

We can still format the values when using `f-string`. The method is similar to those using the `%f` specifiers.

```
pi = 3.1415926535
print(f"Pi is {pi:.3f} to 3 decimal places")
```

```
Results:
# => Pi is 3.142 to 3 decimal places
```

Many more examples can be found at:
`https://zetcode.com/python/fstring/`

Quick Exercise – printing with `f-string`

- Creating a dictionary containing the following information:

Key	Value
name	Jane
age	35
lon	-3.52352
lat	2.25222

- Print (using an `f-string`) the values of this dictionary so that the result looks like: "Jane (located at -3.5, 2.2) is 35 years old"

Operations on strings – splitting

Apart from formatting, there are plenty more operations we can perform on strings. We are going to highlight some of the most common here.

The first we're going to look at is splitting a string by a delimiter character using the `.split()` method. If we don't pass any argument to the `.split()` method, then by default, it will split by spaces. However, we can change this by specifying the delimiter.

```
my_string = "This is a sentence, where each word is separa

print(my_string.split())
print(my_string.split(","))
```

```
Results:
# => ['This', 'is', 'a', 'sentence,', 'where', 'each', 'wo
# => ['This is a sentence', ' where each word is separated
```

Operations on strings – joining

As `.split()` splits a single string into a list, `.join()` joins a list of strings into a single string. To use `.join()`, we first create a string of the delimiter we want to use to join the list of strings by. In this example we're going to use `"-"`. Then we call the `.join()` method, passing the list as an argument.

The result is a single string using the delimiter to separate the items of the list.

```
x = ['This', 'is', 'a', 'sentence,', 'where', 'each', 'wor

print("-".join(x))
```

```
Results:
# => This-is-a-sentence,-where-each-word-is-separated-by-a
```

Operations on strings – changing case

Other common operations on strings involve change the case. For example:

- Make the entire string uppercase or lowercase
- Making the string title case (every where starts with a capital letter).
- Stripping the string by removing any empty spaces either side of the string.

Note we can chain many methods together by doing `.method_1().method_2()`, but only if they return string. If they return `None`, then chaining will not work.

```
x = "    this String Can change case"

print(x.upper())
print(x.lower())
print(x.title())
print(x.strip())
print(x.strip().title())
```

Operations on strings – replacing strings

To replace a substring, we use the `.replace()` method. The first argument is the old string you want to replace. The second argument is what you want to replace it with.

```
x = "This is a string that contains some text"

print(x.replace("contains some", "definitely contains some
```

Results:

```
# => This is a string that definitely contains some text
```

Operations on strings – does it contain a substring?

We can check if a string exists within another string using the `in` keyword. This returns a Boolean value, so we can use it as a condition to an `if` statement.

```
x = "This is a string that contains some text"

if "text" in x:
    print("It exists")
```

```
Results:
# => It exists
```

Introduction to classes

A class is some representation (can be abstract) of an object. Classes can be used to create some kind of structure that can be manipulated and changed, just like the ways you've seen with lists, dictionaries, etc.

Classes allow us to perform Object-oriented Programming (OOP), where we represent concepts by classes.

But to properly understand how classes work, and why we would want to use them, we should take a look at some examples.

Basic syntax

We're going to start off with the very basic syntax, and build up some more complex classes.

To create a class, we use the `class` keyword, and give our new class a name. This introduces a new scope in Python, the scope of the class.

Typically, the first thing we shall see in the class is the `__init__` function.

```
class <name_of_class>:
    def __init__(self, args*):
        <body>
```

Init method

The `__init__` function is a function that gets called automatically as soon as a class is made. This init function can take many arguments, but must always start with a `self`.

In this example, we are creating a class that represents an x, y coordinate. We've called this class `Coordinate`, and we've defined our init function to take an x and y values when the class is being created.

Note its more typical to use titlecase when specifying the class name. So when reading code its easy to see when you're creating a class versus calling a function. You should use this style.

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

Instantiating

To create an *instance* of this class, call the name of the class as you would a function, and pass any parameters you've defined in the init function.

In this example, we are creating a new vector using `Vector(...)` and we're passing the x and y coordinate.

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


point_1 = Vector(5, 2)
```

Class variables

In the previous example, we've been creating a class variables by using `self.<variable_name>`. This is telling Python *this class should have a variable of this name*.

It allows then to reference the variable when working with the class.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.length = self.x + self.y

point_1 = Vector(5, 2)
print(point_1.x)
print(point_1.y)
print(point_1.length)
```

```
Results:
# => 5
# => 2
```

Class Methods

A class can have many methods associated with it. To create a new method, we create a function within the scope of the class, remember that the first parameter of the function should be `self`.

Even in these functions, we can refer to our `self.x` and `self.y` within this new function.

You'll notice that to call this function, we using the `.length()` method similar to how we've worked with strings/lists/etc. This is because in Python, everything is an object!

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

    def length(self):
        return self.x + self.y
```

dunder-methods

While we could, for example, create a function called `.print()`, sometimes we would like to use the in built functions like `print()`. When creating a class, there is a set of *dunder-methods* (double-under to reference the two '`__`' characters either side of the function name).

One of these dunder-methods is `__repr__`, which allows us to specify how the object looks when its printed.

dunder-methods

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

print(OldVector(2, 5))

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

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

print(Vector(2, 5))
```

Results:

```
# => <  main  .OldVector object at 0x7f658721e250>
```

dunder-methods

There are many more dunder-methods you should know when creating classes. We shall go through:

- `__len__` – specify how the length of the class should be computed.
- `__getitem__` – how to index over the class
- `__call__` – how to use the class like a function
- `__iter__` – what to do when iteration starts
- `__next__` – what to do at the next step of the iteration

__len__

The `__len__` function allows us to specify how the `len()` function acts on the class. Take this hypothetical dataset. We create a `__len__` function that returns the length of the unique elements in the dataset.

```python
class Dataset:
    def __init__(self, data):
        self.data = data

    def __len__(self):
        """Return the length of unique elements"""
        return len(set(self.data))

data = Dataset([1, 2, 3, 3, 3, 5, 1])
print(len(data))
```

```
Results:
# => 4
```

__getitem__

Next `__getitem__` allows us to index over a class. This new function must include `self` and a variable to pass the index. Here I've used `idx`. In this function I am simply indexing on the on the classes `self.data`.

```python
class Dataset:
    def __init__(self, data):
        self.data = data

    def __getitem__(self, idx):
        return self.data[idx]

data = Dataset([1, 2, 3, 3, 3, 5, 1])
print(data[2])
```

```
Results:
# => 3
```

__call__

In a small number of cases, it is nice to use the class just like a function. This is what `__call__` allows us to do. In this function we specify what should happen when class is 'called' like a function. In this simple example, we are creating a function that prints the type of food being used as a parameter to the function.

```
class Jaguar:
    def __call__(self, food):
        print(f"The jaguar eats the {food}.")

food = "apple"
animal = Jaguar()

animal(food)
```

```
Results:
# => The jaguar eats the apple.
```

`__iter__` and `__next__`

`__iter__` and `__next__` allow us to make our class iterable, i.e. we can use it in a `for` loop for example.

The `__iter__` function should define what happens when we start the iteration, and `__next__` defines what happens at every step of the iteration.

Let's take a look at an example where we have an iterable set of prime numbers.

__iter__ and __next__

```python
class Primes:
    def __init__(self):
        self.primes = [2, 3, 5, 7, 11]

    def __iter__(self):
        self.idx = 0
        return self

    def __len__(self):
        return len(self.primes)

    def __next__(self):
        if self.idx < len(self):
            item = self.primes[self.idx]
            self.idx += 1
            return item
        else:
            raise StopIteration
```

__iter__ and __next__

And now we can iterate over this class

```
prime_numbers = Primes()

for prime_number in prime_numbers:
    print(prime_number)
```

```
Results:
# => 2
# => 3
# => 5
# => 7
# => 11
```

Inheritance

One special thing about OOP is that its normally designed to provide inheritance – this is true in Python. Inheritance is where you have a base class, and other classes inherit from this base class. This means that the class that inherits from the base class has access to the same methods and class variables. In some cases, it can override some of these features.

Let's take a look an example.

```python
class Animal:
    def growl(self):
        print("The animal growls")

    def walk(self):
        raise NotImplementedError
```

Here we have created a simple class called Animal, that has two functions, one of which will raise an error if its called.

Inheritance

We can inherit from this Animal class by placing our base class in () after the new class name.

Here we are creating two classes, Tiger and Duck. Both of these new classes inherit from Animal. Also, both of these classes are overriding the walk functions. But they are not creating a growl method themselves.

```python
class Tiger(Animal):
    def walk(self):
        print("The Tiger walks through the jungle")

class Duck(Animal):
    def walk(self):
        print("The Duck walks through the jungle")
```

Inheritance

Look at what happens when we create instances of these classes, and call the functions. First we see that the correct method has been called. I.e. for the duck class, the correct `walk` method was called.

```
first_animal = Tiger()
second_animal = Duck()

first_animal.walk()
second_animal.walk()
```

```
Results:
# => The Tiger walks through the jungle
# => The Duck walks through the jungle
```

Inheritance

But what happens if we call the `.growl()` method?

```
first_animal.growl()
second_animal.growl()
```

```
Results:
# => The animal growls
# => The animal growls
```

We see that it still works. Even though both Duck and Tiger didn't create a `.growl()` method, it inherited it from the base class Animal. This works for class methods and class variables.

An object based library system

We're going to improve on our library system from last lecture. Instead of a `functional` style of code, we're going to use a OOP paradigm to create our solution.

Like last time, we're going to create our solution one step at a time.

First, we need to create our class called `Database`. This database is going to take an optional parameter in its init function – the data. If the user specifies data (represented as a list of dictionaries like last time), then the class will populate a class variable called data, else this class variable will be set to an empty list.

Summary:

- Create a class called `Database`.
- When creating an instance of `Database`, the user can optionally specify a list of dictionaries to initialise the class variable `data` with. If no data is provided, this class variable will be initialised to an empty list.

Adding data

We will want to include a function to add data to our database.

Create a class method called `add`, that takes three arguments (in addition to `self` of course), the title, the author, and the release date.

This add function adds the new book entry to the end of `data`. Populate this database with the following information.

Title	Author	Release Date
Moby Dick	Herman Melville	1851
A Study in Scarlet	Sir Arthur Conan Doyle	1887
Frankenstein	Mary Shelley	1818
Hitchhikers Guide to the Galaxy	Douglas Adams	1879

Locating a book

Create a class method called locate by tile that takes the title of the book to look up, and returns the dictionary of all books that have this title. Unlike last time, we don't need to pass the `data` as an argument, as its contained within the class.

Updating our database

Create a class method called `update` that takes 4 arguments:, 1) the key of the value we want to update 2) the value we want to update it to 3) the key we want to check to find out if we have the correct book and 4) the value of the key to check if we have the correct book.

```
db.update(key="release year", value=1979, where_key="title
          where_value="Hitchhikers Guide to the Galaxy")
```

Use this to fix the release data of the Hitchhiker's book.

Printed representation

Using the `__str__` dunder-method (this is similar to `__repr__` as we saw before), create a function that prints out a formatted representation of the entire database as a string. Some of the output should look like:

```
Library System
--------------

Entry 1:
- Name: Moby Dick
- Author: Herman Melville
- Release Date: 1851
...
```

Extending our OOP usage

So far we've used a list of dictionaries. One issue with this is that there is no constraints on the keys we can use. This will certainly create problems if certain keys are missing.

Instead of using dictionaries. We can create another class called `Book` that will take three arguments when it is initialised: `name`, `author`, and `release_date`. The init function should initialise three class variables to save this information.

Modify the database to, instead of working with a list of dictionaries, work with a list of Book objects.

Printed representation – challenge.

Improve upon the printed representation of the last exercise but instead of bulleted lists, use formatted tables using `f-string` formatting (`https://zetcode.com/python/fstring/`).

The output should look like this:

```
Library System
--------------

| Name             | Author             | Release Data |
|------------------|--------------------|--------------|
| Moby Dick        | Herman Melville    |         1851 |
...
```

Notice how Release date is right justified, while Name and Author are left justified.