

Classes and Objects

Python is Object-Oriented

Every value in Python is an object, meaning an instance of a class. Even values that are considered “primitive” in some other languages.

```
1 >>> type(1)
2 <class 'int'>
```

Class Definitions

```
1 class <class_name>(<superclasses>):  
2     <body>
```

- ▶ <class_name> is an identifier
- ▶ <superclasses> is a comma-separated list of superclasses. Can be empty, in which case `object` is implicit superclass
- ▶ <body> is a non-empty sequence of statements

A class definition creates a class object in much the same way that a function definition creates a function object.

Class Attributes

```
1 class Stark:
2     creator = "George R.R. Martin"
3     words = "Winter is coming"
4     sigil = "Direwolf"
5     home = "Winterfell"
6
7     def __init__(self, name=None):
8         self.name = name if name else "No one"
9
10    def full_name(self):
11        return "{} Stark".format(self.name)
```

`creator`, `words`, `sigil`, and `home` are *class attributes*. Class attributes belong to the class and are shared by all instances

Instance Attributes

```
1 class Stark:
2     creator = "George R.R. Martin"
3     words = "Winter is coming"
4     sigil = "Direwolf"
5     home = "Winterfell"
6
7     def __init__(self, name=None):
8         self.name = name if name else "No one"
9
10    def full_name(self):
11        return "{} Stark".format(self.name)
```

- ▶ `self.name` is an instance attribute because it is prefaced with `self`. and defined in a method that has a first parameter named `self`. Each instance of the class has its own copies of instance attributes.
- ▶ `full_name` is an instance method because it is defined in a class and has at least one parameter. The first parameter is implicitly a reference to the instance on which a method is called.

Classes and Objects

In this example, `ned` and `robb` are *instances* of `Stark`. Each instance has it's own `name`.

```
1 >>> import got
2 >>> ned = got.Stark("Eddard")
3 >>> ned.name
4 'Eddard'
5 >>> robb = got.Stark("Robb")
6 >>> robb.name
7 'Robb'
```

Invoking the `full_name()` method on an object implicitly passes the object as the first argument (`self`), which you could (but shouldn't) do explicitly:

```
1 >>> ned.full_name()           # This is normal
2 'Eddard Stark'
3 >>> got.Stark.full_name(ned) # This is only instructive
4 'Eddard Stark'
```

Class Members

Each instance shares the class attributes `creator`, `words`, `sigil`, and `home`.

```
1 >>> got.Stark.sigil
2 'Direwolf'
3 >>> ned.sigil
4 'Direwolf'
5 >>> robb.sigil
6 'Direwolf'
```

Remember that the `is` operator returns `True` if its operands reference the same object in memory. So this demonstrates that `sigil` is shared between the `Stark` class and all instances of the `Stark` class:

```
1 >>> got.Stark.sigil is ned.sigil
2 True
```

Superclasses

Superclasses, or parent classes, or base classes, define attributes that you wish to be common to a family of objects.

Notice that all of our noble houses have the same creator, and every instance has a name. We can represent this commonality by creating a base class for all house classes:

```
1 class GotCharacter:
2     creator = "George R.R. Martin"
3
4     def __init__(self, name=None):
5         self.name = name if name else "No one"
```


Here is `Stark` refactored to use the `GotCharacter` superclass:

```
1 class Stark(GotCharacter):
2     words = "Winter is coming"
3     sigil = "Direwolf"
4     home = "Winterfell"
5
6     def __init__(self, name):
7         # This is how you invoke a superclass method
8         super().__init__(name)
```

Exercise: refactor the other GoT houses to use the `GotCharacter` superclass.

Magic, a.k.a., Dunder Methods

Methods with names that begin and end with `--`

```
1 class SuperTrooper(Trooper):
2
3     def __init__(self, name, is_mustached):
4         super().__init__(name)
5         self.is_mustached = is_mustached
6
7     # Used by print()
8     def __str__(self):
9         return "<{} {}>".format(self.name, ":-{" if self.is_mustached else
10                                ":-|")
11
12     # Used by REPL
13     def __repr__(self):
14         return str(self)
15
16     # Makes instances of SuperTrooper orderable
17     def __lt__(self, other):
18         if self.is_mustached and not other.is_mustached:
19             return False
20         elif not self.is_mustached and other.is_mustached:
21             return True
22         else:
23             return self.name < other.name
```

Sortable SuperTroopers

With the definition of `__lt__(self, other)` in `SuperTrooper`, a list of `SuperTrooper` is sortable.

```
1     sts = [SuperTrooper("Thorny", True),
2             SuperTrooper("Mac", True),
3             SuperTrooper("Rabbit", True),
4             SuperTrooper("Farva", True),
5             SuperTrooper("Foster", False)]
6     print("SuperTroopers:")
7     print(sts)
8     print("SuperTroopers sorted by mustache, then by name:")
9     print(sorted(sts))
```

Produces:

```
1 SuperTroopers:
2 [<Thorny :-{>, <Mac :-{>, <Rabbit :-{>, <Farva :-{>, <Foster :-|>]
3 SuperTroopers sorted by mustache, then by name:
4 [<Foster :-|>, <Farva :-{>, <Mac :-{>, <Rabbit :-{>, <Thorny :-{>]
```

Final Thoughts

Recall the design of the Game of Thrones character types:

- ▶ A superclass `GotCharacter` with class attributes common to Got characters of all houses.
- ▶ A class for each house, subclassing `GotCharacter` and defining the common attributes of all house members.
 - ▶ Each character is an instance of one of these house classes, like `Lannister`, `Stark`, etc.

Is this a good design? What if you had an instance of a `Stark` and you later found out that they're a `Targaryen`? Refactor the design of the Got character classes to allow a character to change houses without having to modify the code and re-run the program.

Conclusion

Magic!