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.

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

Class Definitions

```
class <class_name>(<superclasses>):
    <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

```
class Stark:
    creator = "George R.R. Martin"
    words = "Winter is coming"
    sigil = "Direwolf"
    home = "Winterfell"

    def __init__(self, name=None):
        self.name = name if name else "No one"

    def full_name(self):
        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

```
class Stark:
    creator = "George R.R. Martin"
    words = "Winter is coming"
    sigil = "Direwolf"
    home = "Winterfell"

    def __init__(self, name=None):
        self.name = name if name else "No one"

    def full_name(self):
        return "{} Stark".format(self.name)
```

- `self.name` is an instance attribute becuase 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 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`.

```
>>> import got
>>> ned = got.Stark("Eddard")
>>> ned.name
'Eddard'
>>> robb = got.Stark("Robb")
>>> robb.name
'Robb'
```

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

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

Class Members

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

```
>>> got.Stark.sigil
'Direwolf'
>>> ned.sigil
'Direwolf'
>>> robb.sigil
'Direwolf'
```

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

```
>>> got.Stark.sigil is ned.sigil
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:

```
class GotCharacter:
    creator = "George R.R. Martin"

    def __init__(self, name=None):
        self.name = name if name else "No one"
```

Refactored `Stark`

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

```
class Stark(GotCharacter):
    words = "Winter is coming"
    sigil = "Direwolf"
    home = "Winterfell"

    def __init__(self, name):
        # This is how you invoke a superclass method
        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 __

```
class SuperTrooper(Trooper):

    def __init__(self, name, is_mustached):
        super().__init__(name)
        self.is_mustached = is_mustached

    # Used by print()
    def __str__(self):
        return "<{} {}>".format(self.name, ":-{" if self.is_mustached else
            ":-|")

    # Used by REPL
    def __repr__(self):
        return str(self)

    # Makes instances of SuperTrooper orderable
    def __lt__(self, other):
        if self.is_mustached and not other.is_mustached:
            return False
        elif not self.is_mustached and other.is_mustached:
            return True
        else:
            return self.name < other.name
```

Sortable SuperTroopers

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

```
    sts = [SuperTrooper("Thorny", True),
           SuperTrooper("Mac", True),
           SuperTrooper("Rabbit", True),
           SuperTrooper("Farva", True),
           SuperTrooper("Foster", False)]
    print("SuperTroopers:")
    print(sts)
    print("SuperTroopers sorted by mustache, then by name:")
    print(sorted(sts))
```

Produces:

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