Object-Oriented Python

The Basics

I'll assume you know the basics of OOP in Python.

```
class Pet:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return '' # silent by default
class Dog(Pet):
    def speak(self):
        return 'Woof!'
```

```
>>> fido = Dog("Fido")
>>> fido.speak()
'Woof!'
```

2 v. 3: object

Python 2:

```
# Must extend object
class Vehicle(object):
    def __init__(self, id_number):
        self.id_number = id_number
```

Python 3:

```
# No (object) needed
class Vehicle:
    def __init__(self, id_number):
        self.id_number = id_number
```

2 v. 3: super

Python 2:

```
class Car(Vehicle):
    def __init__(self, id_number, color):
        # Need to pass args to super
        super(Car, self).__init__(id_number)
        self.color = color
```

Python 3:

```
class Car(Vehicle):
    def __init__(self, id_number, color):
        # super doesn't need args for single inheritance
        super().__init__(id_number)
        self.color = color
```

2 v. 3: Ordering

Python 2:

```
>>> Car(42, "red") < Car(45, "red")
True
>>> Car(42, "red") < Car(45, "red")
False
```

Python 3:

```
>>> Car(42, "red") < Car(45, "red")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Car() < Car()
```

Properties

A hybrid between a method and attribute.

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last
    @property
    def full_name(self):
        return self.first + " " + self.last
```

Dynamic Attribute

Even though it's defined as a method, you access it like a member variable.

```
>>> guy = Person("Joe", "Smith")

>>> guy.full_name
'Joe Smith'

>>> guy.full_name()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

In fact, you **can't** call it like a method, even if you want to.

Read-Only

By default, a property is read-only.

```
>>> guy.full_name
'Joe Smith'
>>> guy.full_name = 'John Doe'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

This is either a feature or a bug, depending on what you want.

Setters

You can make a property writable with a **setter** - an extra, specially-marked method.

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def full_name(self):
        return self.first + " " + self.last

    @full_name.setter
    def full_name(self, value):
        first, last = value.split(" ")
        self.first = first
        self.last = last
```

Setting Properties

```
    @full_name.setter
    def full_name(self, value):
        first, last = value.split(" ")
        self.first = first
        self.last = last
```

```
>>> guy = Person("Joe", "Smith")
>>> guy.full_name = "Sam Jones"
>>> print(guy.first)
Sam
>>> print(guy.last)
Jones
>>> print(guy.full_name)
Sam Jones
```

Practice: getset.py

```
class Person: # or "Person(object):" for Python 2
    def __init__(self, first, last):
        self.first = first
        self.last = last
    @property
    def full_name(self):
        return self.first + " " + self.last
    @full_name.setter
    def full_name(self, value):
        first, last = value.split(" ")
        self.first = first
        self.last = last
guy = Person("Joe", "Smith")
print(guy.full_name)
guy.full_name = "Sam Jones"
print(guy.last + ", " + guy.first)
```

```
# Running "python3 getset.py" should have this output:
Joe Smith
Jones, Sam
```

Read-Only Pattern

A common Python design pattern: Use `@property` to create a read-only attribute.

```
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
```

```
>>> ticket = Ticket(42)
>>> ticket.price
42
>>> ticket.price = 41
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Validation

Useful setter pattern: ensure a value is set to the correct range.

```
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Validation

This will raise a run-time error if we try to cheat:

```
# In class Ticket...
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

```
>>> t = Ticket(42)
>>> t.price
42
>>> t.price = -1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in price
ValueError: Nice try
```

Validation

Pop quiz: Do you see a way to improve the constructor?

What's the potential problem lurking in this code?

```
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Use Setters In Your Methods

You can use an object's setters in your own methods.

For `Ticket`, this lets us get the validation check at object-creation time!

```
class Ticket:
    def __init__(self, price):
        # instead of "self._price = price"
        self.price = price

    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Lab: Properties

Lab file: `oop/properties.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `oop/properties_extra.py`

Properties And Refactoring

Here's a money class.

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    # And some other methods...
```

Imagine this is released as a library, and many teams are using and relying on this class.

Refactor

One day, we decide to internally just keep track of cents.

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
```

That creates a maintainability problem. Can you spot it?

The problem

Other code referencing its attributes suddenly breaks.

```
money = Money(27, 12)
message = "I have {:d} dollars and {:d} cents."
# This line breaks, because there's no longer
# dollars or cents attributes.
print(message.format(money.dollars, money.cents))
```

This could be a major impediment to changing the class interface.

The solution

But not in Python.

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
    # Getter and setter for dollars...
    @property
    def dollars(self):
        return self.total_cents // 100
    @dollars.setter
    def dollars(self, new_dollars):
        self.total_cents =
            100 * new_dollars + self.cents
```

Properties for Refactoring

```
    # And the getter and setter for cents.
    @property
    def cents(self):
        return self.total_cents % 100
    @cents.setter
    def cents(self, new_cents):
        self.total_cents =
            100 * self.dollars + new_cents
```