Object Properties

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

# Note: In Python 2.x, write "class Person(object):"
# Otherwise @property works only partially.
```

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.

The @property Decorator

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

`property` is a decorator, built into Python.

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: `properties.py`

- In `labs` folder
- When you are done, study the solution - compare to what you wrote.
- ... and then optionally do `oop/properties_extra.py`

Instructions: `LABS.txt` in courseware.