Some Useful Annotations: `@staticmethod`  
- We saw annotations earlier, as examples of higher-order functions.
- For classes, Python defines a few specialized methods.
- The `@staticmethod` annotation denotes a class method (i.e., ordinary function), which does not apply to any particular object.

```python
class Account:
    total_deposits = 0

    @staticmethod
    def total_deposits():
        return Account.total_deposits
```

- Now we can write:
  ```python
  acct = Account(...)  
  acct.total_deposits()  # Total deposits in bank.  
  Account.total_deposits()  # Ditto
  ```

Some Useful Annotations: `@property`  
- I've said that generally, method calls are the preferred way for clients to access an object (rather than direct access to instance variables.)
- This practice allows the class implementor to hide details of implementation.
- Still it's cumbersome to have to say, e.g., `aPoint.getX()` rather than `aPoint.x`, and `aPoint.setX(v)` rather than `aPoint.x = v`.
- To alleviate this, Python introduced the idea of a property object.
- When a property object is an attribute of an object, it calls a function when it is fetched from its containing object by dot notation.
- The property object can also be defined to call a different function on assignment to the attribute.
- Attributes defined as property objects are called computed or managed attributes.

Properties (Long Form)  
```python
class rational:
    def __init__(self, num, den):
        g = gcd(num, den)
        self.num, self.den = num/g, den/g

    def getNumer(self):
        return self.num

    def setNumer(self, val):
        self.num = val / gcd(val, self.den)

    numer = property(getNumer, setNumer)
```

- As a result:
  ```python
  >>> a = rational(3, 4)
  >>> a.numer
  3
  >>> a.numer = 5
  >>> a.numer
  5
  ```

Properties (Short Form)  
```python
The built-in property function is also a decorator:
```
Inheritance

• Classes are often conceptually related, sharing operations and behavior.
• One important relation is the subtype or "is-a" relation.
• Examples: A car is a vehicle. A square is a plane geometric figure.
• When multiple types of object are related like this, one can often define operations that will work on all of them, with each type adjusting the operation appropriately.
• In Python (like C++ and Java), a language mechanism called inheritance accomplishes this.

Example: Geometric Plane Figures

• Want to define a collection of types that represent polygons (squares, trapezoids, etc.).
• First, what are the common characteristics that make sense for all polygons?
• A point here is mostly to document our concept of Polygon, since we don't know how to implement any of these in general.

Partial Implementations

• Even though we don't know anything about Polygons, we can give default implementations.

Specializing Polygons

• At this point, we can introduce simple (non-intersecting) polygons, for which there is a simple area formula.
• This says that a SimplePolygon is a kind of Polygon, and that the attributes of Polygon are to be inherited by SimplePolygon.
• So far, none of these Polygons are much good, since they have no defined vertices.
• We say that Polygon and SimplePolygon are abstract types.

A Concrete Type

• Finally, a square is a type of simple Polygon:

(Simple) Inheritance Explained

• We say that Polygon and SimplePolygon are abstract types.
• Inheritance (in Python) works like nested environment frames.
Do You Understand the Machinery?

```python
class Parent:
    def f(s):
        print("Parent.f")
    def g(s):
        s.f()

class Child(Parent):
    def f(me):
        print("Child.f")

aChild = Child()
aChild.g()
```

Multiple Inheritance

- A class describes some set attributes.
- One can imagine assembling a set of attributes from smaller clusters.
- A class describes some attributes.
- For example, many kinds of object represent some kind of collection.
- Many of these methods are similar; perhaps we can consolidate.

Multiple Inheritance Example

```python
class Printable:
    """A mixin class for creating a str method that prints a sequence object. Assumes that the type defines __getitem__."""
    def left_bracket(self):
        return type(self).name + "["
    def right_bracket(self):
        return "]
    def str(self):
        result = self.left_bracket()
        for i in range(len(self) - 1):
            result += str(self[i]) + ", "
        if len(self) > 0:
            result += str(self[-1])
        return result + self.right_bracket()
```

Multiple Inheritance Example

- I define a new kind of "sequence with benefits" and would like a distinct way of printing it.

Super

- Sometimes we just want to add to or use the behavior of our parent.

Example: "Memoization"

```python
class Evaluator:
    def value(self, x):
        some expensive computation that depends only on x

class FastEvaluator(Evaluator):
    def __init__(self):
        self.memo_table = {}  # Maps arguments to results
    def value(self, x):
        """A memoized value computation"""
        if x not in self.memo_table:
            self.memo_table[x] = Evaluator.value(self, x)
        return self.memo_table[x]
```

We must override __init__ to create a FastEvaluator object.

Example: "Super"

```python
class Transformer:
    def mogrify(self):
        """Do something"""

class CountedTransformer(Transformer):
    """A Transformer that counts the number of calls to its mogrify method."""
    def __init__(self):
        self.count = 0
    def mogrify(self):
        self.count += 1
        return Transformer.mogrify(self)
```

For example, suppose we have a class that uses:

```python
def count(self):
    return self.count
```

Example: "Memoization"

```python
class Evaluator:
    def value(self, x):
        some expensive computation that depends only on x
```

```python
class FastEvaluator(Evaluator):
    def __init__(self):
        self.memo_table = {}  # Maps arguments to results
    def value(self, x):
        """A memoized value computation"""
        if x not in self.memo_table:
            self.memo_table[x] = Evaluator.value(self, x)
        return self.memo_table[x]
```

FastEvaluator.value must call the .value method of its base (super) class, but we can't just say self.value(x), since that gives an infinite recursion.

Example: "Super"

```python
class Transformer:
    def mogrify(self):
        """Do something"""

class CountedTransformer(Transformer):
    """A Transformer that counts the number of calls to its mogrify method."""
    def __init__(self):
        self.count = 0
    def mogrify(self):
        self.count += 1
        return Transformer.mogrify(self)
```

For example, suppose we have a class that uses:

```python
def count(self):
    return self.count
```