Using these setters and getters is simple, but it's not Pythonic:

```
r0 = OldResistor(50e3)
print('Before:', r0.get_ohms())
r0.set_ohms(10e3)
print('After: ', r0.get_ohms())
```

```
>>>
Before: 50000.0
After:  10000.0
```

Such methods are especially clumsy for operations like incrementing in place:

```
r0.set_ohms(r0.get_ohms() - 4e3)
assert r0.get_ohms() == 6e3
```

These utility methods do, however, help define the interface for a class, making it easier to encapsulate functionality, validate usage, and define boundaries. Those are important goals when designing a class to ensure that you don't break callers as the class evolves over time.

In Python, however, you never need to implement explicit setter or getter methods. Instead, you should always start your implementations with simple public attributes, as I do here:

```
class Resistor:
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0

r1 = Resistor(50e3)
r1.ohms = 10e3
```

These attributes make operations like incrementing in place natural and clear:

```
r1.ohms += 5e3
```

Later, if I decide I need special behavior when an attribute is set, I can migrate to the `@property` decorator (see Item 26: "Define Function Decorators with `functools.wraps`" for background) and its corresponding `setter` attribute. Here, I define a new subclass of `Resistor` that lets me vary the `current` by assigning the `voltage` property. Note that in order for this code to work properly, the names of both the setter and the getter methods must match the intended property name:

```
class VoltageResistance(Resistor):
    def __init__(self, ohms):
```

```
        super().__init__(ohms)
        self._voltage = 0

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, voltage):
        self._voltage = voltage
        self.current = self._voltage / self.ohms
```

Now, assigning the `voltage` property will run the `voltage` setter method, which in turn will update the `current` attribute of the object to match:

```
r2 = VoltageResistance(1e3)
print(f'Before: {r2.current:.2f} amps')
r2.voltage = 10
print(f'After:  {r2.current:.2f} amps')
```

```
>>>
Before: 0.00 amps
After:  0.01 amps
```

Specifying a `setter` on a property also enables me to perform type checking and validation on values passed to the class. Here, I define a class that ensures all resistance values are above zero ohms:

```
class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if ohms <= 0:
            raise ValueError(f'ohms must be > 0; got {ohms}')
        self._ohms = ohms
```

Assigning an invalid resistance to the attribute now raises an exception:

```
r3 = BoundedResistance(1e3)
r3.ohms = 0
```

```
>>>
Traceback ...
ValueError: ohms must be > 0; got 0
```

An exception is also raised if I pass an invalid value to the constructor:

```
BoundedResistance(-5)
```

```
>>>
Traceback ...
ValueError: ohms must be > 0; got -5
```

This happens because `BoundedResistance.__init__` calls `Resistor.__init__`, which assigns `self.ohms = -5`. That assignment causes the `@ohms.setter` method from `BoundedResistance` to be called, and it immediately runs the validation code before object construction has completed.

I can even use `@property` to make attributes from parent classes immutable:

```
class FixedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if hasattr(self, '_ohms'):
            raise AttributeError("Ohms is immutable")
        self._ohms = ohms
```

Trying to assign to the property after construction raises an exception:

```
r4 = FixedResistance(1e3)
r4.ohms = 2e3
```

```
>>>
Traceback ...
AttributeError: Ohms is immutable
```

When you use `@property` methods to implement setters and getters, be sure that the behavior you implement is not surprising. For example, don't set other attributes in getter property methods:

```
class MysteriousResistor(Resistor):
    @property
    def ohms(self):
```

```
        self.voltage = self._ohms * self.current
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        self._ohms = ohms
```

Setting other attributes in getter property methods leads to extremely bizarre behavior:

```
r7 = MysteriousResistor(10)
r7.current = 0.01
print(f'Before: {r7.voltage:.2f}')
r7.ohms
print(f'After:  {r7.voltage:.2f}')
```

```
>>>
Before: 0.00
After:  0.10
```

The best policy is to modify only related object state in `@property.setter` methods. Be sure to also avoid any other side effects that the caller may not expect beyond the object, such as importing modules dynamically, running slow helper functions, doing I/O, or making expensive database queries. Users of a class will expect its attributes to be like any other Python object: quick and easy. Use normal methods to do anything more complex or slow.

The biggest shortcoming of `@property` is that the methods for an attribute can only be shared by subclasses. Unrelated classes can't share the same implementation. However, Python also supports *descriptors* (see Item 46: "Use Descriptors for Reusable `@property` Methods") that enable reusable property logic and many other use cases.

Things to Remember

- Define new class interfaces using simple public attributes and avoid defining setter and getter methods.
- Use `@property` to define special behavior when attributes are accessed on your objects, if necessary.
- Follow the rule of least surprise and avoid odd side effects in your `@property` methods.
- Ensure that `@property` methods are fast; for slow or complex work—especially involving I/O or causing side effects—use normal methods instead.

Item 45: Consider `@property` Instead of Refactoring Attributes

The built-in `@property` decorator makes it easy for simple accesses of an instance's attributes to act smarter (see Item 44: "Use Plain Attributes Instead of Setter and Getter Methods"). One advanced but common use of `@property` is transitioning what was once a simple numerical attribute into an on-the-fly calculation. This is extremely helpful because it lets you migrate all existing usage of a class to have new behaviors without requiring any of the call sites to be rewritten (which is especially important if there's calling code that you don't control). `@property` also provides an important stopgap for improving interfaces over time.

For example, say that I want to implement a leaky bucket quota using plain Python objects. Here, the `Bucket` class represents how much quota remains and the duration for which the quota will be available:

```
from datetime import datetime, timedelta

class Bucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.quota = 0

    def __repr__(self):
        return f'Bucket(quota={self.quota})'
```

The leaky bucket algorithm works by ensuring that, whenever the bucket is filled, the amount of quota does not carry over from one period to the next:

```
def fill(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
    bucket.quota += amount
```

Each time a quota consumer wants to do something, it must first ensure that it can deduct the amount of quota it needs to use:

```
def deduct(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        return False  # Bucket hasn't been filled this period
    if bucket.quota - amount < 0:
        return False  # Bucket was filled, but not enough
```

```
    bucket.quota -= amount
    return True       # Bucket had enough, quota consumed
```

To use this class, first I fill the bucket up:

```
bucket = Bucket(60)
fill(bucket, 100)
print(bucket)
```

```
>>>
Bucket(quota=100)
```

Then, I deduct the quota that I need:

```
if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
print(bucket)
```

```
>>>
Had 99 quota
Bucket(quota=1)
```

Eventually, I'm prevented from making progress because I try to deduct more quota than is available. In this case, the bucket's quota level remains unchanged:

```
if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')
print(bucket)
```

```
>>>
Not enough for 3 quota
Bucket(quota=1)
```

The problem with this implementation is that I never know what quota level the bucket started with. The quota is deducted over the course of the period until it reaches zero. At that point, `deduct` will always return `False` until the bucket is refilled. When that happens, it would be useful to know whether callers to `deduct` are being blocked because the `Bucket` ran out of quota or because the `Bucket` never had quota during this period in the first place.

To fix this, I can change the class to keep track of the `max_quota` issued in the period and the `quota_consumed` in the period:

```
class NewBucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
```

```
        self.reset_time = datetime.now()
        self.max_quota = 0
        self.quota_consumed = 0

    def __repr__(self):
        return (f'NewBucket(max_quota={self.max_quota}, '
                f'quota_consumed={self.quota_consumed})')
```

To match the previous interface of the original `Bucket` class, I use a `@property` method to compute the current level of quota on-the-fly using these new attributes:

```
    @property
    def quota(self):
        return self.max_quota - self.quota_consumed
```

When the `quota` attribute is assigned, I take special action to be compatible with the current usage of the class by the `fill` and `deduct` functions:

```
    @quota.setter
    def quota(self, amount):
        delta = self.max_quota - amount
        if amount == 0:
            # Quota being reset for a new period
            self.quota_consumed = 0
            self.max_quota = 0
        elif delta < 0:
            # Quota being filled for the new period
            assert self.quota_consumed == 0
            self.max_quota = amount
        else:
            # Quota being consumed during the period
            assert self.max_quota >= self.quota_consumed
            self.quota_consumed += delta
```

Rerunning the demo code from above produces the same results:

```
bucket = NewBucket(60)
print('Initial', bucket)
fill(bucket, 100)
print('Filled', bucket)

if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
```

```
print('Now', bucket)

if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')

print('Still', bucket)
```

```
>>>
Initial NewBucket(max_quota=0, quota_consumed=0)
Filled NewBucket(max_quota=100, quota_consumed=0)
Had 99 quota
Now NewBucket(max_quota=100, quota_consumed=99)
Not enough for 3 quota
Still NewBucket(max_quota=100, quota_consumed=99)
```

The best part is that the code using `Bucket.quota` doesn't have to change or know that the class has changed. New usage of `Bucket` can do the right thing and access `max_quota` and `quota_consumed` directly.

I especially like `@property` because it lets you make incremental progress toward a better data model over time. Reading the `Bucket` example above, you may have thought that `fill` and `deduct` should have been implemented as instance methods in the first place. Although you're probably right (see Item 37: "Compose Classes Instead of Nesting Many Levels of Built-in Types"), in practice there are many situations in which objects start with poorly defined interfaces or act as dumb data containers. This happens when code grows over time, scope increases, multiple authors contribute without anyone considering long-term hygiene, and so on.

`@property` is a tool to help you address problems you'll come across in real-world code. Don't overuse it. When you find yourself repeatedly extending `@property` methods, it's probably time to refactor your class instead of further paving over your code's poor design.

Things to Remember

- Use `@property` to give existing instance attributes new functionality.
- Make incremental progress toward better data models by using `@property`.
- Consider refactoring a class and all call sites when you find yourself using `@property` too heavily.

Item 46: Use Descriptors for Reusable @property Methods

The big problem with the @property built-in (see Item 44: "Use Plain Attributes Instead of Setter and Getter Methods" and Item 45: "Consider @property Instead of Refactoring Attributes") is reuse. The methods it decorates can't be reused for multiple attributes of the same class. They also can't be reused by unrelated classes.

For example, say I want a class to validate that the grade received by a student on a homework assignment is a percentage:

```
class Homework:
    def __init__(self):
        self._grade = 0

    @property
    def grade(self):
        return self._grade

    @grade.setter
    def grade(self, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._grade = value
```

Using @property makes this class easy to use:

```
galileo = Homework()
galileo.grade = 95
```

Say that I also want to give the student a grade for an exam, where the exam has multiple subjects, each with a separate grade:

```
class Exam:
    def __init__(self):
        self._writing_grade = 0
        self._math_grade = 0

    @staticmethod
    def _check_grade(value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
```

This quickly gets tedious. For each section of the exam I need to add a new @property and related validation:

```
    @property
    def writing_grade(self):
        return self._writing_grade

    @writing_grade.setter
    def writing_grade(self, value):
        self._check_grade(value)
        self._writing_grade = value

    @property
    def math_grade(self):
        return self._math_grade

    @math_grade.setter
    def math_grade(self, value):
        self._check_grade(value)
        self._math_grade = value
```

Also, this approach is not general. If I want to reuse this percentage validation in other classes beyond homework and exams, I'll need to write the @property boilerplate and _check_grade method over and over again.

The better way to do this in Python is to use a *descriptor*. The *descriptor protocol* defines how attribute access is interpreted by the language. A descriptor class can provide __get__ and __set__ methods that let you reuse the grade validation behavior without boilerplate. For this purpose, descriptors are also better than mix-ins (see Item 41: "Consider Composing Functionality with Mix-in Classes") because they let you reuse the same logic for many different attributes in a single class.

Here, I define a new class called Exam with class attributes that are Grade instances. The Grade class implements the descriptor protocol:

```
class Grade:
    def __get__(self, instance, instance_type):
        ...

    def __set__(self, instance, value):
        ...
```

```
class Exam:
    # Class attributes
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()
```

Before I explain how the `Grade` class works, it's important to understand what Python will do when such descriptor attributes are accessed on an `Exam` instance. When I assign a property:

```
exam = Exam()
exam.writing_grade = 40
```

it is interpreted as:

```
Exam.__dict__['writing_grade'].__set__(exam, 40)
```

When I retrieve a property:

```
exam.writing_grade
```

it is interpreted as:

```
Exam.__dict__['writing_grade'].__get__(exam, Exam)
```

What drives this behavior is the `__getattribute__` method of `object` (see Item 47: "Use `__getattr__`, `__getattribute__`, and `__setattr__` for Lazy Attributes"). In short, when an `Exam` instance doesn't have an attribute named `writing_grade`, Python falls back to the `Exam` class's attribute instead. If this class attribute is an object that has `__get__` and `__set__` methods, Python assumes that you want to follow the descriptor protocol.

Knowing this behavior and how I used `@property` for grade validation in the `Homework` class, here's a reasonable first attempt at implementing the `Grade` descriptor:

```
class Grade:
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._value = value
```

Unfortunately, this is wrong and results in broken behavior. Accessing multiple attributes on a single Exam instance works as expected:

```
class Exam:
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()

first_exam = Exam()
first_exam.writing_grade = 82
first_exam.science_grade = 99
print('Writing', first_exam.writing_grade)
print('Science', first_exam.science_grade)
```

```
>>>
Writing 82
Science 99
```

But accessing these attributes on multiple Exam instances causes unexpected behavior:

```
second_exam = Exam()
second_exam.writing_grade = 75
print(f'Second {second_exam.writing_grade} is right')
print(f'First  {first_exam.writing_grade} is wrong; '
      f'should be 82')
```

```
>>>
Second 75 is right
First  75 is wrong; should be 82
```

The problem is that a single Grade instance is shared across all Exam instances for the class attribute writing_grade. The Grade instance for this attribute is constructed once in the program lifetime, when the Exam class is first defined, not each time an Exam instance is created.

To solve this, I need the Grade class to keep track of its value for each unique Exam instance. I can do this by saving the per-instance state in a dictionary:

```
class Grade:
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return self._values.get(instance, 0)
```

```
    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._values[instance] = value
```

This implementation is simple and works well, but there's still one gotcha: It leaks memory. The `_values` dictionary holds a reference to every instance of `Exam` ever passed to `__set__` over the lifetime of the program. This causes instances to never have their reference count go to zero, preventing cleanup by the garbage collector (see Item 81: "Use `tracemalloc` to Understand Memory Usage and Leaks" for how to detect this type of problem).

To fix this, I can use Python's `weakref` built-in module. This module provides a special class called `WeakKeyDictionary` that can take the place of the simple dictionary used for `_values`. The unique behavior of `WeakKeyDictionary` is that it removes `Exam` instances from its set of items when the Python runtime knows it's holding the instance's last remaining reference in the program. Python does the bookkeeping for me and ensures that the `_values` dictionary will be empty when all `Exam` instances are no longer in use:

```
from weakref import WeakKeyDictionary

class Grade:
    def __init__(self):
        self._values = WeakKeyDictionary()

    def __get__(self, instance, instance_type):
        ...

    def __set__(self, instance, value):
        ...
```

Using this implementation of the `Grade` descriptor, everything works as expected:

```
class Exam:
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()

first_exam = Exam()
first_exam.writing_grade = 82
second_exam = Exam()
second_exam.writing_grade = 75
print(f'First  {first_exam.writing_grade} is right')
print(f'Second {second_exam.writing_grade} is right')
```

```
>>>
First  82 is right
Second 75 is right
```

Things to Remember

- Reuse the behavior and validation of `@property` methods by defining your own descriptor classes.
- Use `WeakKeyDictionary` to ensure that your descriptor classes don't cause memory leaks.
- Don't get bogged down trying to understand exactly how `__getattribute__` uses the descriptor protocol for getting and setting attributes.

Item 47: Use __getattr__, __getattribute__, and __setattr__ for Lazy Attributes

Python's `object` hooks make it easy to write generic code for gluing systems together. For example, say that I want to represent the records in a database as Python objects. The database has its schema set already. My code that uses objects corresponding to those records must also know what the database looks like. However, in Python, the code that connects Python objects to the database doesn't need to explicitly specify the schema of the records; it can be generic.

How is that possible? Plain instance attributes, `@property` methods, and descriptors can't do this because they all need to be defined in advance. Python makes this dynamic behavior possible with the `__getattr__` special method. If a class defines `__getattr__`, that method is called every time an attribute can't be found in an object's instance dictionary:

```
class LazyRecord:
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        value = f'Value for {name}'
        setattr(self, name, value)
        return value
```

Here, I access the missing property `foo`. This causes Python to call the `__getattr__` method above, which mutates the instance dictionary `__dict__`:

```
data = LazyRecord()
print('Before:', data.__dict__)
```

```
print('foo:   ', data.foo)
print('After: ', data.__dict__)
```

```
>>>
Before: {'exists': 5}
foo:    Value for foo
After:  {'exists': 5, 'foo': 'Value for foo'}
```

Here, I add logging to `LazyRecord` to show when `__getattr__` is actually called. Note how I call `super().__getattr__()` to use the superclass's implementation of `__getattr__` in order to fetch the real property value and avoid infinite recursion (see Item 40: "Initialize Parent Classes with `super`" for background):

```
class LoggingLazyRecord(LazyRecord):
    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r}), '
              f'populating instance dictionary')
        result = super().__getattr__(name)
        print(f'* Returning {result!r}')
        return result

data = LoggingLazyRecord()
print('exists:     ', data.exists)
print('First foo:  ', data.foo)
print('Second foo: ', data.foo)
```

```
>>>
exists:      5
* Called __getattr__('foo'), populating instance dictionary
* Returning 'Value for foo'
First foo:   Value for foo
Second foo:  Value for foo
```

The `exists` attribute is present in the instance dictionary, so `__getattr__` is never called for it. The `foo` attribute is not in the instance dictionary initially, so `__getattr__` is called the first time. But the call to `__getattr__` for `foo` also does a `setattr`, which populates `foo` in the instance dictionary. This is why the second time I access `foo`, it doesn't log a call to `__getattr__`.

This behavior is especially helpful for use cases like lazily accessing schemaless data. `__getattr__` runs once to do the hard work of loading a property; all subsequent accesses retrieve the existing result.

Say that I also want transactions in this database system. The next time the user accesses a property, I want to know whether the corresponding record in the database is still valid and whether the

transaction is still open. The __getattr__ hook won't let me do this reliably because it will use the object's instance dictionary as the fast path for existing attributes.

To enable this more advanced use case, Python has another `object` hook called __getattribute__. This special method is called every time an attribute is accessed on an object, even in cases where it *does* exist in the attribute dictionary. This enables me to do things like check global transaction state on every property access. It's important to note that such an operation can incur significant overhead and negatively impact performance, but sometimes it's worth it. Here, I define `ValidatingRecord` to log each time __getattribute__ is called:

```
class ValidatingRecord:
    def __init__(self):
        self.exists = 5

    def __getattribute__(self, name):
        print(f'* Called __getattribute__({name!r})')
        try:
            value = super().__getattribute__(name)
            print(f'* Found {name!r}, returning {value!r}')
            return value
        except AttributeError:
            value = f'Value for {name}'
            print(f'* Setting {name!r} to {value!r}')
            setattr(self, name, value)
            return value

data = ValidatingRecord()
print('exists:     ', data.exists)
print('First foo:  ', data.foo)
print('Second foo: ', data.foo)
```

```
>>>
* Called __getattribute__('exists')
* Found 'exists', returning 5
exists:      5
* Called __getattribute__('foo')
* Setting 'foo' to 'Value for foo'
First foo:   Value for foo
* Called __getattribute__('foo')
* Found 'foo', returning 'Value for foo'
Second foo:  Value for foo
```

In the event that a dynamically accessed property shouldn't exist, I can raise an `AttributeError` to cause Python's standard missing property behavior for both `__getattr__` and `__getattribute__`:

```
class MissingPropertyRecord:
    def __getattr__(self, name):
        if name == 'bad_name':
            raise AttributeError(f'{name} is missing')
        ...

data = MissingPropertyRecord()
data.bad_name
```

```
>>>
Traceback ...
AttributeError: bad_name is missing
```

Python code implementing generic functionality often relies on the `hasattr` built-in function to determine when properties exist, and the `getattr` built-in function to retrieve property values. These functions also look in the instance dictionary for an attribute name before calling `__getattr__`:

```
data = LoggingLazyRecord()  # Implements __getattr__
print('Before:          ', data.__dict__)
print('Has first foo:   ', hasattr(data, 'foo'))
print('After:           ', data.__dict__)
print('Has second foo:  ', hasattr(data, 'foo'))
```

```
>>>
Before:          {'exists': 5}
* Called __getattr__('foo'), populating instance dictionary
* Returning 'Value for foo'
Has first foo:   True
After:           {'exists': 5, 'foo': 'Value for foo'}
Has second foo:  True
```

In the example above, `__getattr__` is called only once. In contrast, classes that implement `__getattribute__` have that method called each time `hasattr` or `getattr` is used with an instance:

```
data = ValidatingRecord()  # Implements __getattribute__
print('Has first foo:   ', hasattr(data, 'foo'))
print('Has second foo:  ', hasattr(data, 'foo'))
```

```
>>>
* Called __getattribute__('foo')
* Setting 'foo' to 'Value for foo'
Has first foo:   True
```

```
* Called __getattribute__('foo')
* Found 'foo', returning 'Value for foo'
Has second foo:  True
```

Now, say that I want to lazily push data back to the database when values are assigned to my Python object. I can do this with `__setattr__`, a similar `object` hook that lets you intercept arbitrary attribute assignments. Unlike when retrieving an attribute with `__getattr__` and `__getattribute__`, there's no need for two separate methods. The `__setattr__` method is always called every time an attribute is assigned on an instance (either directly or through the `setattr` built-in function):

```
class SavingRecord:
    def __setattr__(self, name, value):
        # Save some data for the record
        ...
        super().__setattr__(name, value)
```

Here, I define a logging subclass of `SavingRecord`. Its `__setattr__` method is always called on each attribute assignment:

```
class LoggingSavingRecord(SavingRecord):
    def __setattr__(self, name, value):
        print(f'* Called __setattr__({name!r}, {value!r})')
        super().__setattr__(name, value)

data = LoggingSavingRecord()
print('Before: ', data.__dict__)
data.foo = 5
print('After:  ', data.__dict__)
data.foo = 7
print('Finally:', data.__dict__)
```

```
>>>
Before:  {}
* Called __setattr__('foo', 5)
After:   {'foo': 5}
* Called __setattr__('foo', 7)
Finally: {'foo': 7}
```

The problem with `__getattribute__` and `__setattr__` is that they're called on every attribute access for an object, even when you may not want that to happen. For example, say that I want attribute accesses on my object to actually look up keys in an associated dictionary:

```
class BrokenDictionaryRecord:
    def __init__(self, data):
        self._data = {}
```

```
    def __getattribute__(self, name):
        print(f'* Called __getattribute__({name!r})')
        return self._data[name]
```

This requires accessing `self._data` from the `__getattribute__` method. However, if I actually try to do that, Python will recurse until it reaches its stack limit, and then it'll die:

```
data = BrokenDictionaryRecord({'foo': 3})
data.foo
```

```
>>>
* Called __getattribute__('foo')
* Called __getattribute__('_data')
* Called __getattribute__('_data')
* Called __getattribute__('_data')
...
Traceback ...
RecursionError: maximum recursion depth exceeded while calling
➥a Python object
```

The problem is that `__getattribute__` accesses `self._data`, which causes `__getattribute__` to run again, which accesses `self._data` again, and so on. The solution is to use the `super().__getattribute__` method to fetch values from the instance attribute dictionary. This avoids the recursion:

```
class DictionaryRecord:
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        print(f'* Called __getattribute__({name!r})')
        data_dict = super().__getattribute__('_data')
        return data_dict[name]

data = DictionaryRecord({'foo': 3})
print('foo: ', data.foo)
```

```
>>>
* Called __getattribute__('foo')
foo:  3
```

`__setattr__` methods that modify attributes on an object also need to use `super().__setattr__` accordingly.

Things to Remember

- ✦ Use `__getattr__` and `__setattr__` to lazily load and save attributes for an object.

- Understand that __getattr__ only gets called when accessing a missing attribute, whereas __getattribute__ gets called every time any attribute is accessed.
- Avoid infinite recursion in __getattribute__ and __setattr__ by using methods from super() (i.e., the object class) to access instance attributes.

Item 48: Validate Subclasses with __init_subclass__

One of the simplest applications of metaclasses is verifying that a class was defined correctly. When you're building a complex class hierarchy, you may want to enforce style, require overriding methods, or have strict relationships between class attributes. Metaclasses enable these use cases by providing a reliable way to run your validation code each time a new subclass is defined.

Often a class's validation code runs in the __init__ method, when an object of the class's type is constructed at runtime (see Item 44: "Use Plain Attributes Instead of Setter and Getter Methods" for an example). Using metaclasses for validation can raise errors much earlier, such as when the module containing the class is first imported at program startup.

Before I get into how to define a metaclass for validating subclasses, it's important to understand the metaclass action for standard objects. A metaclass is defined by inheriting from type. In the default case, a metaclass receives the contents of associated class statements in its __new__ method. Here, I can inspect and modify the class information before the type is actually constructed:

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print(f'* Running {meta}.__new__ for {name}')
        print('Bases:', bases)
        print(class_dict)
        return type.__new__(meta, name, bases, class_dict)

class MyClass(metaclass=Meta):
    stuff = 123

    def foo(self):
        pass

class MySubclass(MyClass):
    other = 567

    def bar(self):
        pass
```

The metaclass has access to the name of the class, the parent classes it inherits from (`bases`), and all the class attributes that were defined in the class's body. All classes inherit from `object`, so it's not explicitly listed in the `tuple` of base classes:

```
>>>
* Running <class '__main__.Meta'>.__new__ for MyClass
Bases: ()
{'__module__': '__main__',
 '__qualname__': 'MyClass',
 'stuff': 123,
 'foo': <function MyClass.foo at 0x105a05280>}
* Running <class '__main__.Meta'>.__new__ for MySubclass
Bases: (<class '__main__.MyClass'>,)
{'__module__': '__main__',
 '__qualname__': 'MySubclass',
 'other': 567,
 'bar': <function MySubclass.bar at 0x105a05310>}
```

I can add functionality to the `Meta.__new__` method in order to validate all of the parameters of an associated class before it's defined. For example, say that I want to represent any type of multisided polygon. I can do this by defining a special validating metaclass and using it in the base class of my polygon class hierarchy. Note that it's important not to apply the same validation to the base class:

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Polygon class
        if bases:
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(metaclass=ValidatePolygon):
    sides = None  # Must be specified by subclasses

    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Triangle(Polygon):
    sides = 3
```

```
class Rectangle(Polygon):
    sides = 4

class Nonagon(Polygon):
    sides = 9

assert Triangle.interior_angles() == 180
assert Rectangle.interior_angles() == 360
assert Nonagon.interior_angles() == 1260
```

If I try to define a polygon with fewer than three sides, the validation will cause the `class` statement to fail immediately after the `class` statement body. This means the program will not even be able to start running when I define such a class (unless it's defined in a dynamically imported module; see Item 88: "Know How to Break Circular Dependencies" for how this can happen):

```
print('Before class')

class Line(Polygon):
    print('Before sides')
    sides = 2
    print('After sides')

print('After class')
```

```
>>>
Before class
Before sides
After sides
Traceback ...
ValueError: Polygons need 3+ sides
```

This seems like quite a lot of machinery in order to get Python to accomplish such a basic task. Luckily, Python 3.6 introduced simplified syntax—the `__init_subclass__` special class method—for achieving the same behavior while avoiding metaclasses entirely. Here, I use this mechanism to provide the same level of validation as before:

```
class BetterPolygon:
    sides = None  # Must be specified by subclasses

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.sides < 3:
            raise ValueError('Polygons need 3+ sides')
```

```
    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Hexagon(BetterPolygon):
    sides = 6

assert Hexagon.interior_angles() == 720
```

The code is much shorter now, and the `ValidatePolygon` metaclass is gone entirely. It's also easier to follow since I can access the `sides` attribute directly on the `cls` instance in `__init_subclass__` instead of having to go into the class's dictionary with `class_dict['sides']`. If I define an invalid subclass of `BetterPolygon`, the same exception is raised:

```
print('Before class')

class Point(BetterPolygon):
    sides = 1

print('After class')
```

```
>>>
Before class
Traceback ...
ValueError: Polygons need 3+ sides
```

Another problem with the standard Python metaclass machinery is that you can only specify a single metaclass per class definition. Here, I define a second metaclass that I'd like to use for validating the fill color used for a region (not necessarily just polygons):

```
class ValidateFilled(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Filled class
        if bases:
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
        return type.__new__(meta, name, bases, class_dict)

class Filled(metaclass=ValidateFilled):
    color = None  # Must be specified by subclasses
```

When I try to use the Polygon metaclass and Filled metaclass together, I get a cryptic error message:

```
class RedPentagon(Filled, Polygon):
    color = 'red'
    sides = 5

>>>
Traceback ...
TypeError: metaclass conflict: the metaclass of a derived
➥class must be a (non-strict) subclass of the metaclasses
➥of all its bases
```

It's possible to fix this by creating a complex hierarchy of metaclass type definitions to layer validation:

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate non-root classes
        if not class_dict.get('is_root'):
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(metaclass=ValidatePolygon):
    is_root = True
    sides = None  # Must be specified by subclasses

class ValidateFilledPolygon(ValidatePolygon):
    def __new__(meta, name, bases, class_dict):
        # Only validate non-root classes
        if not class_dict.get('is_root'):
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
        return super().__new__(meta, name, bases, class_dict)

class FilledPolygon(Polygon, metaclass=ValidateFilledPolygon):
    is_root = True
    color = None  # Must be specified by subclasses
```

This requires every FilledPolygon instance to be a Polygon instance:

```
class GreenPentagon(FilledPolygon):
    color = 'green'
    sides = 5

greenie = GreenPentagon()
assert isinstance(greenie, Polygon)
```

Validation works for colors:

```
class OrangePentagon(FilledPolygon):
    color = 'orange'
    sides = 5
```

```
>>>
Traceback ...
ValueError: Fill color must be supported
```

Validation also works for number of sides:

```
class RedLine(FilledPolygon):
    color = 'red'
    sides = 2
```

```
>>>
Traceback ...
ValueError: Polygons need 3+ sides
```

However, this approach ruins composability, which is often the purpose of class validation like this (similar to mix-ins; see Item 41: "Consider Composing Functionality with Mix-in Classes"). If I want to apply the color validation logic from `ValidateFilledPolygon` to another hierarchy of classes, I'll have to duplicate all of the logic again, which reduces code reuse and increases boilerplate.

The `__init_subclass__` special class method can also be used to solve this problem. It can be defined by multiple levels of a class hierarchy as long as the `super` built-in function is used to call any parent or sibling `__init_subclass__` definitions (see Item 40: "Initialize Parent Classes with `super`" for a similar example). It's even compatible with multiple inheritance. Here, I define a class to represent region fill color that can be composed with the `BetterPolygon` class from before:

```
class Filled:
    color = None  # Must be specified by subclasses

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.color not in ('red', 'green', 'blue'):
            raise ValueError('Fills need a valid color')
```

I can inherit from both classes to define a new class. Both classes call `super().__init_subclass__()`, causing their corresponding validation logic to run when the subclass is created:

```
class RedTriangle(Filled, Polygon):
    color = 'red'
    sides = 3
```

```
ruddy = RedTriangle()
assert isinstance(ruddy, Filled)
assert isinstance(ruddy, Polygon)
```

If I specify the number of sides incorrectly, I get a validation error:

```
print('Before class')

class BlueLine(Filled, Polygon):
    color = 'blue'
    sides = 2

print('After class')
```

```
>>>
Before class
Traceback ...
ValueError: Polygons need 3+ sides
```

If I specify the color incorrectly, I also get a validation error:

```
print('Before class')

class BeigeSquare(Filled, Polygon):
    color = 'beige'
    sides = 4

print('After class')
```

```
>>>
Before class
Traceback ...
ValueError: Fills need a valid color
```

You can even use __init_subclass__ in complex cases like diamond inheritance (see Item 40: "Initialize Parent Classes with super"). Here, I define a basic diamond hierarchy to show this in action:

```
class Top:
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Top for {cls}')

class Left(Top):
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Left for {cls}')

class Right(Top):
    def __init_subclass__(cls):
```

```
        super().__init_subclass__()
        print(f'Right for {cls}')

class Bottom(Left, Right):
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Bottom for {cls}')
```

```
>>>
Top for <class '__main__.Left'>
Top for <class '__main__.Right'>
Top for <class '__main__.Bottom'>
Right for <class '__main__.Bottom'>
Left for <class '__main__.Bottom'>
```

As expected, `Top.__init_subclass__` is called only a single time for each class, even though there are two paths to it for the `Bottom` class through its `Left` and `Right` parent classes.

Things to Remember

- The `__new__` method of metaclasses is run after the `class` statement's entire body has been processed.
- Metaclasses can be used to inspect or modify a class after it's defined but before it's created, but they're often more heavyweight than what you need.
- Use `__init_subclass__` to ensure that subclasses are well formed at the time they are defined, before objects of their type are constructed.
- Be sure to call `super().__init_subclass__` from within your class's `__init_subclass__` definition to enable validation in multiple layers of classes and multiple inheritance.

Item 49: Register Class Existence with `__init_subclass__`

Another common use of metaclasses is to automatically register types in a program. Registration is useful for doing reverse lookups, where you need to map a simple identifier back to a corresponding class.

For example, say that I want to implement my own serialized representation of a Python `object` using JSON. I need a way to turn an `object` into a JSON string. Here, I do this generically by defining a

base class that records the constructor parameters and turns them into a JSON dictionary:

```
import json

class Serializable:
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({'args': self.args})
```

This class makes it easy to serialize simple, immutable data structures like `Point2D` to a string:

```
class Point2D(Serializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point2D({self.x}, {self.y})'

point = Point2D(5, 3)
print('Object:    ', point)
print('Serialized:', point.serialize())
```

```
>>>
Object:     Point2D(5, 3)
Serialized: {"args": [5, 3]}
```

Now, I need to deserialize this JSON string and construct the `Point2D` object it represents. Here, I define another class that can deserialize the data from its `Serializable` parent class:

```
class Deserializable(Serializable):
    @classmethod
    def deserialize(cls, json_data):
        params = json.loads(json_data)
        return cls(*params['args'])
```

Using `Deserializable` makes it easy to serialize and deserialize simple, immutable objects in a generic way:

```
class BetterPoint2D(Deserializable):
    ...
```

```
before = BetterPoint2D(5, 3)
print('Before:    ', before)
data = before.serialize()
print('Serialized:', data)
after = BetterPoint2D.deserialize(data)
print('After:     ', after)
```

```
>>>
Before:     Point2D(5, 3)
Serialized: {"args": [5, 3]}
After:      Point2D(5, 3)
```

The problem with this approach is that it works only if you know the intended type of the serialized data ahead of time (e.g., `Point2D`, `BetterPoint2D`). Ideally, you'd have a large number of classes serializing to JSON and one common function that could deserialize any of them back to a corresponding Python `object`.

To do this, I can include the serialized object's class name in the JSON data:

```
class BetterSerializable:
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({
            'class': self.__class__.__name__,
            'args': self.args,
        })

    def __repr__(self):
        name = self.__class__.__name__
        args_str = ', '.join(str(x) for x in self.args)
        return f'{name}({args_str})'
```

Then, I can maintain a mapping of class names back to constructors for those objects. The general `deserialize` function works for any classes passed to `register_class`:

```
registry = {}

def register_class(target_class):
    registry[target_class.__name__] = target_class

def deserialize(data):
    params = json.loads(data)
```

```
    name = params['class']
    target_class = registry[name]
    return target_class(*params['args'])
```

To ensure that deserialize always works properly, I must call register_class for every class I may want to deserialize in the future:

```
class EvenBetterPoint2D(BetterSerializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

register_class(EvenBetterPoint2D)
```

Now, I can deserialize an arbitrary JSON string without having to know which class it contains:

```
before = EvenBetterPoint2D(5, 3)
print('Before:    ', before)
data = before.serialize()
print('Serialized:', data)
after = deserialize(data)
print('After:     ', after)
```

```
>>>
Before:     EvenBetterPoint2D(5, 3)
Serialized: {"class": "EvenBetterPoint2D", "args": [5, 3]}
After:      EvenBetterPoint2D(5, 3)
```

The problem with this approach is that it's possible to forget to call register_class:

```
class Point3D(BetterSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x = x
        self.y = y
        self.z = z

# Forgot to call register_class! Whoops!
```

This causes the code to break at runtime, when I finally try to deserialize an instance of a class I forgot to register:

```
point = Point3D(5, 9, -4)
data = point.serialize()
deserialize(data)
```

```
>>>
Traceback ...
KeyError: 'Point3D'
```

Even though I chose to subclass `BetterSerializable`, I don't actually get all of its features if I forget to call `register_class` after the `class` statement body. This approach is error prone and especially challenging for beginners. The same omission can happen with *class decorators* (see Item 51: "Prefer Class Decorators Over Metaclasses for Composable Class Extensions" for when those are appropriate).

What if I could somehow act on the programmer's intent to use `BetterSerializable` and ensure that `register_class` is called in all cases? Metaclasses enable this by intercepting the `class` statement when subclasses are defined (see Item 48: "Validate Subclasses with `__init_subclass__`" for details on the machinery). Here, I use a metaclass to register the new type immediately after the class's body:

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        cls = type.__new__(meta, name, bases, class_dict)
        register_class(cls)
        return cls

class RegisteredSerializable(BetterSerializable,
                             metaclass=Meta):
    pass
```

When I define a subclass of `RegisteredSerializable`, I can be confident that the call to `register_class` happened and `deserialize` will always work as expected:

```
class Vector3D(RegisteredSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x, self.y, self.z = x, y, z

before = Vector3D(10, -7, 3)
print('Before:    ', before)
data = before.serialize()
print('Serialized:', data)
print('After:     ', deserialize(data))
```

```
>>>
Before:     Vector3D(10, -7, 3)
Serialized: {"class": "Vector3D", "args": [10, -7, 3]}
After:      Vector3D(10, -7, 3)
```

An even better approach is to use the __init_subclass__ special class method. This simplified syntax, introduced in Python 3.6, reduces the visual noise of applying custom logic when a class is defined. It also makes it more approachable to beginners who may be confused by the complexity of metaclass syntax:

```
class BetterRegisteredSerializable(BetterSerializable):
    def __init_subclass__(cls):
        super().__init_subclass__()
        register_class(cls)

class Vector1D(BetterRegisteredSerializable):
    def __init__(self, magnitude):
        super().__init__(magnitude)
        self.magnitude = magnitude

before = Vector1D(6)
print('Before:    ', before)
data = before.serialize()
print('Serialized:', data)
print('After:     ', deserialize(data))
```

```
>>>
Before:     Vector1D(6)
Serialized: {"class": "Vector1D", "args": [6]}
After:      Vector1D(6)
```

By using __init_subclass__ (or metaclasses) for class registration, you can ensure that you'll never miss registering a class as long as the inheritance tree is right. This works well for serialization, as I've shown, and also applies to database object-relational mappings (ORMs), extensible plug-in systems, and callback hooks.

Things to Remember

- Class registration is a helpful pattern for building modular Python programs.
- Metaclasses let you run registration code automatically each time a base class is subclassed in a program.
- Using metaclasses for class registration helps you avoid errors by ensuring that you never miss a registration call.
- Prefer __init_subclass__ over standard metaclass machinery because it's clearer and easier for beginners to understand.

Item 50: Annotate Class Attributes with __set_name__

One more useful feature enabled by metaclasses is the ability to modify or annotate properties after a class is defined but before the class is actually used. This approach is commonly used with *descriptors* (see Item 46: "Use Descriptors for Reusable `@property` Methods") to give them more introspection into how they're being used within their containing class.

For example, say that I want to define a new class that represents a row in a customer database. I'd like to have a corresponding property on the class for each column in the database table. Here, I define a descriptor class to connect attributes to column names:

```
class Field:
    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

With the column name stored in the `Field` descriptor, I can save all of the per-instance state directly in the instance dictionary as protected fields by using the `setattr` built-in function, and later I can load state with `getattr`. At first, this seems to be much more convenient than building descriptors with the `weakref` built-in module to avoid memory leaks.

Defining the class representing a row requires supplying the database table's column name for each class attribute:

```
class Customer:
    # Class attributes
    first_name = Field('first_name')
    last_name = Field('last_name')
    prefix = Field('prefix')
    suffix = Field('suffix')
```

Using the class is simple. Here, you can see how the `Field` descriptors modify the instance dictionary `__dict__` as expected:

```
cust = Customer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
```

```
cust.first_name = 'Euclid'
print(f'After:  {cust.first_name!r} {cust.__dict__}')
```

```
>>>
Before: '' {}
After:  'Euclid' {'_first_name': 'Euclid'}
```

But the class definition seems redundant. I already declared the name of the field for the class on the left (`'field_name ='`). Why do I also have to pass a string containing the same information to the `Field` constructor (`Field('first_name')`) on the right?

```
class Customer:
    # Left side is redundant with right side
    first_name = Field('first_name')
    ...
```

The problem is that the order of operations in the `Customer` class definition is the opposite of how it reads from left to right. First, the `Field` constructor is called as `Field('first_name')`. Then, the return value of that is assigned to `Customer.field_name`. There's no way for a `Field` instance to know upfront which class attribute it will be assigned to.

To eliminate this redundancy, I can use a metaclass. Metaclasses let you hook the `class` statement directly and take action as soon as a `class` body is finished (see Item 48: "Validate Subclasses with `__init_subclass__`" for details on how they work). In this case, I can use the metaclass to assign `Field.name` and `Field.internal_name` on the descriptor automatically instead of manually specifying the field name multiple times:

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        for key, value in class_dict.items():
            if isinstance(value, Field):
                value.name = key
                value.internal_name = '_' + key
        cls = type.__new__(meta, name, bases, class_dict)
        return cls
```

Here, I define a base class that uses the metaclass. All classes representing database rows should inherit from this class to ensure that they use the metaclass:

```
class DatabaseRow(metaclass=Meta):
    pass
```

To work with the metaclass, the `Field` descriptor is largely unchanged. The only difference is that it no longer requires arguments to be passed

to its constructor. Instead, its attributes are set by the `Meta.__new__` method above:

```
class Field:
    def __init__(self):
        # These will be assigned by the metaclass.
        self.name = None
        self.internal_name = None

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

By using the metaclass, the new `DatabaseRow` base class, and the new `Field` descriptor, the class definition for a database row no longer has the redundancy from before:

```
class BetterCustomer(DatabaseRow):
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()
```

The behavior of the new class is identical to the behavior of the old one:

```
cust = BetterCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Euler'
print(f'After:  {cust.first_name!r} {cust.__dict__}')
```

```
>>>
Before: '' {}
After:  'Euler' {'_first_name': 'Euler'}
```

The trouble with this approach is that you can't use the `Field` class for properties unless you also inherit from `DatabaseRow`. If you somehow forget to subclass `DatabaseRow`, or if you don't want to due to other structural requirements of the class hierarchy, the code will break:

```
class BrokenCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()
```

```
cust = BrokenCustomer()
cust.first_name = 'Mersenne'
```

```
>>>
Traceback ...
TypeError: attribute name must be string, not 'NoneType'
```

The solution to this problem is to use the `__set_name__` special method for descriptors. This method, introduced in Python 3.6, is called on every descriptor instance when its containing class is defined. It receives as parameters the owning class that contains the descriptor instance and the attribute name to which the descriptor instance was assigned. Here, I avoid defining a metaclass entirely and move what the `Meta.__new__` method from above was doing into `__set_name__`:

```
class Field:
    def __init__(self):
        self.name = None
        self.internal_name = None

    def __set_name__(self, owner, name):
        # Called on class creation for each descriptor
        self.name = name
        self.internal_name = '_' + name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

Now, I can get the benefits of the `Field` descriptor without having to inherit from a specific parent class or having to use a metaclass:

```
class FixedCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()

cust = FixedCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Mersenne'
print(f'After:  {cust.first_name!r} {cust.__dict__}')
```

```
>>>
Before: '' {}
After:  'Mersenne' {'_first_name': 'Mersenne'}
```

Things to Remember

- Metaclasses enable you to modify a class's attributes before the class is fully defined.
- Descriptors and metaclasses make a powerful combination for declarative behavior and runtime introspection.
- Define `__set_name__` on your descriptor classes to allow them to take into account their surrounding class and its property names.
- Avoid memory leaks and the `weakref` built-in module by having descriptors store data they manipulate directly within a class's instance dictionary.

Item 51: Prefer Class Decorators Over Metaclasses for Composable Class Extensions

Although metaclasses allow you to customize class creation in multiple ways (see Item 48: "Validate Subclasses with `__init_subclass__`" and Item 49: "Register Class Existence with `__init_subclass__`"), they still fall short of handling every situation that may arise.

For example, say that I want to decorate all of the methods of a class with a helper that prints arguments, return values, and exceptions raised. Here, I define the debugging decorator (see Item 26: "Define Function Decorators with `functools.wraps`" for background):

```
from functools import wraps

def trace_func(func):
    if hasattr(func, 'tracing'):  # Only decorate once
        return func

    @wraps(func)
    def wrapper(*args, **kwargs):
        result = None
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            result = e
            raise
```

```
        finally:
            print(f'{func.__name__}({args!r}, {kwargs!r}) -> '
                  f'{result!r}')

    wrapper.tracing = True
    return wrapper
```

I can apply this decorator to various special methods in my new `dict` subclass (see Item 43: "Inherit from `collections.abc` for Custom Container Types" for background):

```
class TraceDict(dict):
    @trace_func
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    @trace_func
    def __setitem__(self, *args, **kwargs):
        return super().__setitem__(*args, **kwargs)

    @trace_func
    def __getitem__(self, *args, **kwargs):
        return super().__getitem__(*args, **kwargs)

    ...
```

And I can verify that these methods are decorated by interacting with an instance of the class:

```
trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass  # Expected
```

```
>>>
__init__(({'hi': 1}, [('hi', 1)]), {}) -> None
__setitem__(({'hi': 1, 'there': 2}, 'there', 2), {}) -> None
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➥{}) -> KeyError('does not exist')
```

The problem with this code is that I had to redefine all of the methods that I wanted to decorate with `@trace_func`. This is redundant boilerplate that's hard to read and error prone. Further, if a new method is

later added to the dict superclass, it won't be decorated unless I also define it in TraceDict.

One way to solve this problem is to use a metaclass to automatically decorate all methods of a class. Here, I implement this behavior by wrapping each function or method in the new type with the trace_func decorator:

```
import types

trace_types = (
    types.MethodType,
    types.FunctionType,
    types.BuiltinFunctionType,
    types.BuiltinMethodType,
    types.MethodDescriptorType,
    types.ClassMethodDescriptorType)

class TraceMeta(type):
    def __new__(meta, name, bases, class_dict):
        klass = super().__new__(meta, name, bases, class_dict)

        for key in dir(klass):
            value = getattr(klass, key)
            if isinstance(value, trace_types):
                wrapped = trace_func(value)
                setattr(klass, key, wrapped)

        return klass
```

Now, I can declare my dict subclass by using the TraceMeta metaclass and verify that it works as expected:

```
class TraceDict(dict, metaclass=TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass  # Expected
```

```
>>>
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
```

```
__getitem__({'hi': 1, 'there': 2}, 'does not exist'),
➥{}) -> KeyError('does not exist')
```

This works, and it even prints out a call to `__new__` that was missing from my earlier implementation. What happens if I try to use `TraceMeta` when a superclass already has specified a metaclass?

```
class OtherMeta(type):
    pass

class SimpleDict(dict, metaclass=OtherMeta):
    pass

class TraceDict(SimpleDict, metaclass=TraceMeta):
    pass
```

```
>>>
Traceback ...
TypeError: metaclass conflict: the metaclass of a derived
➥class must be a (non-strict) subclass of the metaclasses
➥of all its bases
```

This fails because `TraceMeta` does not inherit from `OtherMeta`. In theory, I can use metaclass inheritance to solve this problem by having `OtherMeta` inherit from `TraceMeta`:

```
class TraceMeta(type):
    ...

class OtherMeta(TraceMeta):
    pass

class SimpleDict(dict, metaclass=OtherMeta):
    pass

class TraceDict(SimpleDict, metaclass=TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass  # Expected
```

```
>>>
__init_subclass__((), {}) -> None
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➥{}) -> KeyError('does not exist')
```

But this won't work if the metaclass is from a library that I can't modify, or if I want to use multiple utility metaclasses like `TraceMeta` at the same time. The metaclass approach puts too many constraints on the class that's being modified.

To solve this problem, Python supports *class decorators*. Class decorators work just like function decorators: They're applied with the `@` symbol prefixing a function before the class declaration. The function is expected to modify or re-create the class accordingly and then return it:

```
def my_class_decorator(klass):
    klass.extra_param = 'hello'
    return klass

@my_class_decorator
class MyClass:
    pass

print(MyClass)
print(MyClass.extra_param)
```

```
>>>
<class '__main__.MyClass'>
hello
```

I can implement a class decorator to apply `trace_func` to all methods and functions of a class by moving the core of the `TraceMeta.__new__` method above into a stand-alone function. This implementation is much shorter than the metaclass version:

```
def trace(klass):
    for key in dir(klass):
        value = getattr(klass, key)
        if isinstance(value, trace_types):
            wrapped = trace_func(value)
            setattr(klass, key, wrapped)
    return klass
```

I can apply this decorator to my `dict` subclass to get the same behavior as I get by using the metaclass approach above:

```
@trace
class TraceDict(dict):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass  # Expected
```

```
>>>
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➥{}) -> KeyError('does not exist')
```

Class decorators also work when the class being decorated already has a metaclass:

```
class OtherMeta(type):
    pass

@trace
class TraceDict(dict, metaclass=OtherMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass  # Expected
```

```
>>>
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➥{}) -> KeyError('does not exist')
```

When you're looking for composable ways to extend classes, class decorators are the best tool for the job. (See Item 73: "Know How

to Use `heapq` for Priority Queues" for a useful class decorator called `functools.total_ordering`.)

Things to Remember

- A class decorator is a simple function that receives a `class` instance as a parameter and returns either a new class or a modified version of the original class.
- Class decorators are useful when you want to modify every method or attribute of a class with minimal boilerplate.
- Metaclasses can't be composed together easily, while many class decorators can be used to extend the same class without conflicts.

7 Concurrency and Parallelism

Concurrency enables a computer to do many different things *seemingly* at the same time. For example, on a computer with one CPU core, the operating system rapidly changes which program is running on the single processor. In doing so, it interleaves execution of the programs, providing the illusion that the programs are running simultaneously.

Parallelism, in contrast, involves *actually* doing many different things at the same time. A computer with multiple CPU cores can execute multiple programs simultaneously. Each CPU core runs the instructions of a separate program, allowing each program to make forward progress during the same instant.

Within a single program, concurrency is a tool that makes it easier for programmers to solve certain types of problems. Concurrent programs enable many distinct paths of execution, including separate streams of I/O, to make forward progress in a way that seems to be both simultaneous and independent.

The key difference between parallelism and concurrency is *speedup*. When two distinct paths of execution in a program make forward progress in parallel, the time it takes to do the total work is cut in half; the speed of execution is faster by a factor of two. In contrast, concurrent programs may run thousands of separate paths of execution seemingly in parallel but provide no speedup for the total work.

Python makes it easy to write concurrent programs in a variety of styles. Threads support a relatively small amount of concurrency, while coroutines enable vast numbers of concurrent functions. Python can also be used to do parallel work through system calls, subprocesses, and C extensions. But it can be very difficult to make concurrent Python code truly run in parallel. It's important to understand how to best utilize Python in these different situations.