```
    def average_grade(self, name):
        grades = self._grades[name]
        return sum(grades) / len(grades)
```

Using the class is simple:

```
book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
book.report_grade('Isaac Newton', 95)
book.report_grade('Isaac Newton', 85)

print(book.average_grade('Isaac Newton'))
```

```
>>>
90.0
```

Dictionaries and their related built-in types are so easy to use that there's a danger of overextending them to write brittle code. For example, say that I want to extend the `SimpleGradebook` class to keep a `list` of grades by subject, not just overall. I can do this by changing the `_grades` dictionary to map student names (its keys) to yet another dictionary (its values). The innermost dictionary will map subjects (its keys) to a `list` of grades (its values). Here, I do this by using a `defaultdict` instance for the inner dictionary to handle missing subjects (see Item 17: "Prefer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State" for background):

```
from collections import defaultdict

class BySubjectGradebook:
    def __init__(self):
        self._grades = {}                       # Outer dict

    def add_student(self, name):
        self._grades[name] = defaultdict(list)  # Inner dict
```

This seems straightforward enough. The `report_grade` and `average_grade` methods gain quite a bit of complexity to deal with the multilevel dictionary, but it's seemingly manageable:

```
    def report_grade(self, name, subject, grade):
        by_subject = self._grades[name]
        grade_list = by_subject[subject]
        grade_list.append(grade)

    def average_grade(self, name):
        by_subject = self._grades[name]
```

```
        total, count = 0, 0
        for grades in by_subject.values():
            total += sum(grades)
            count += len(grades)
        return total / count
```

Using the class remains simple:

```
book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
print(book.average_grade('Albert Einstein'))
```

```
>>>
81.25
```

Now, imagine that the requirements change again. I also want to track the weight of each score toward the overall grade in the class so that midterm and final exams are more important than pop quizzes. One way to implement this feature is to change the innermost dictionary; instead of mapping subjects (its keys) to a `list` of grades (its values), I can use the `tuple` of `(score, weight)` in the values `list`:

```
class WeightedGradebook:
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = defaultdict(list)

    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]
        grade_list = by_subject[subject]
        grade_list.append((score, weight))
```

Although the changes to `report_grade` seem simple—just make the grade list store `tuple` instances—the `average_grade` method now has a loop within a loop and is difficult to read:

```
    def average_grade(self, name):
        by_subject = self._grades[name]

        score_sum, score_count = 0, 0
        for subject, scores in by_subject.items():
            subject_avg, total_weight = 0, 0
```

```
            for score, weight in scores:
                subject_avg += score * weight
                total_weight += weight

            score_sum += subject_avg / total_weight
            score_count += 1

        return score_sum / score_count
```

Using the class has also gotten more difficult. It's unclear what all of the numbers in the positional arguments mean:

```
book = WeightedGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75, 0.05)
book.report_grade('Albert Einstein', 'Math', 65, 0.15)
book.report_grade('Albert Einstein', 'Math', 70, 0.80)
book.report_grade('Albert Einstein', 'Gym', 100, 0.40)
book.report_grade('Albert Einstein', 'Gym', 85, 0.60)
print(book.average_grade('Albert Einstein'))
```

```
>>>
80.25
```

When you see complexity like this, it's time to make the leap from built-in types like dictionaries, tuples, sets, and lists to a hierarchy of classes.

In the grades example, at first I didn't know I'd need to support weighted grades, so the complexity of creating classes seemed unwarranted. Python's built-in dictionary and `tuple` types made it easy to keep going, adding layer after layer to the internal bookkeeping. But you should avoid doing this for more than one level of nesting; using dictionaries that contain dictionaries makes your code hard to read by other programmers and sets you up for a maintenance nightmare.

As soon as you realize that your bookkeeping is getting complicated, break it all out into classes. You can then provide well-defined interfaces that better encapsulate your data. This approach also enables you to create a layer of abstraction between your interfaces and your concrete implementations.

### Refactoring to Classes

There are many approaches to refactoring (see Item 89: "Consider `warnings` to Refactor and Migrate Usage" for another). In this case,

I can start moving to classes at the bottom of the dependency tree: a single grade. A class seems too heavyweight for such simple information. A `tuple`, though, seems appropriate because grades are immutable. Here, I use the `tuple` of `(score, weight)` to track grades in a list:

```
grades = []
grades.append((95, 0.45))
grades.append((85, 0.55))
total = sum(score * weight for score, weight in grades)
total_weight = sum(weight for _, weight in grades)
average_grade = total / total_weight
```

I used `_` (the underscore variable name, a Python convention for unused variables) to capture the first entry in each grade's `tuple` and ignore it when calculating the `total_weight`.

The problem with this code is that `tuple` instances are positional. For example, if I want to associate more information with a grade, such as a set of notes from the teacher, I need to rewrite every usage of the two-tuple to be aware that there are now three items present instead of two, which means I need to use `_` further to ignore certain indexes:

```
grades = []
grades.append((95, 0.45, 'Great job'))
grades.append((85, 0.55, 'Better next time'))
total = sum(score * weight for score, weight, _ in grades)
total_weight = sum(weight for _, weight, _ in grades)
average_grade = total / total_weight
```

This pattern of extending tuples longer and longer is similar to deepening layers of dictionaries. As soon as you find yourself going longer than a two-tuple, it's time to consider another approach.

The `namedtuple` type in the `collections` built-in module does exactly what I need in this case: It lets me easily define tiny, immutable data classes:

```
from collections import namedtuple


Grade = namedtuple('Grade', ('score', 'weight'))
```

These classes can be constructed with positional or keyword arguments. The fields are accessible with named attributes. Having named attributes makes it easy to move from a `namedtuple` to a class later if the requirements change again and I need to, say, support mutability or behaviors in the simple data containers.

### Limitations of `namedtuple`

Although `namedtuple` is useful in many circumstances, it's important to understand when it can do more harm than good:

- You can't specify default argument values for `namedtuple` classes. This makes them unwieldy when your data may have many optional properties. If you find yourself using more than a handful of attributes, using the built-in `dataclasses` module may be a better choice.
- The attribute values of `namedtuple` instances are still accessible using numerical indexes and iteration. Especially in externalized APIs, this can lead to unintentional usage that makes it harder to move to a real class later. If you're not in control of all of the usage of your `namedtuple` instances, it's better to explicitly define a new class.

Next, I can write a class to represent a single subject that contains a set of grades:

```
class Subject:
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

Then, I write a class to represent a set of subjects that are being studied by a single student:

```
class Student:
    def __init__(self):
        self._subjects = defaultdict(Subject)

    def get_subject(self, name):
        return self._subjects[name]
```

```
    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count
```

Finally, I'd write a container for all of the students, keyed dynamically by their names:

```
class Gradebook:
    def __init__(self):
        self._students = defaultdict(Student)

    def get_student(self, name):
        return self._students[name]
```

The line count of these classes is almost double the previous implementation's size. But this code is much easier to read. The example driving the classes is also more clear and extensible:

```
book = Gradebook()
albert = book.get_student('Albert Einstein')
math = albert.get_subject('Math')
math.report_grade(75, 0.05)
math.report_grade(65, 0.15)
math.report_grade(70, 0.80)
gym = albert.get_subject('Gym')
gym.report_grade(100, 0.40)
gym.report_grade(85, 0.60)
print(albert.average_grade())
```

```
>>>
80.25
```

It would also be possible to write backward-compatible methods to help migrate usage of the old API style to the new hierarchy of objects.

## Things to Remember

- Avoid making dictionaries with values that are dictionaries, long tuples, or complex nestings of other built-in types.
- Use `namedtuple` for lightweight, immutable data containers before you need the flexibility of a full class.
- Move your bookkeeping code to using multiple classes when your internal state dictionaries get complicated.

## Item 38: Accept Functions Instead of Classes for Simple Interfaces

Many of Python's built-in APIs allow you to customize behavior by passing in a function. These *hooks* are used by APIs to call back your code while they execute. For example, the `list` type's `sort` method takes an optional `key` argument that's used to determine each index's value for sorting (see Item 14: "Sort by Complex Criteria Using the `key` Parameter" for details). Here, I sort a `list` of names based on their lengths by providing the `len` built-in function as the `key` hook:

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=len)
print(names)
```

```
>>>
['Plato', 'Socrates', 'Aristotle', 'Archimedes']
```

In other languages, you might expect hooks to be defined by an abstract class. In Python, many hooks are just stateless functions with well-defined arguments and return values. Functions are ideal for hooks because they are easier to describe and simpler to define than classes. Functions work as hooks because Python has *first-class* functions: Functions and methods can be passed around and referenced like any other value in the language.

For example, say that I want to customize the behavior of the `defaultdict` class (see Item 17: "Prefer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State" for background). This data structure allows you to supply a function that will be called with no arguments each time a missing key is accessed. The function must return the default value that the missing key should have in the dictionary. Here, I define a hook that logs each time a key is missing and returns `0` for the default value:

```
def log_missing():
    print('Key added')
    return 0
```

Given an initial dictionary and a set of desired increments, I can cause the `log_missing` function to run and print twice (for `'red'` and `'orange'`):

```
from collections import defaultdict

current = {'green': 12, 'blue': 3}
increments = [
```

```
    ('red', 5),
    ('blue', 17),
    ('orange', 9),
]
result = defaultdict(log_missing, current)
print('Before:', dict(result))
for key, amount in increments:
    result[key] += amount
print('After: ', dict(result))
```

```
>>>
Before: {'green': 12, 'blue': 3}
Key added
Key added
After:  {'green': 12, 'blue': 20, 'red': 5, 'orange': 9}
```

Supplying functions like `log_missing` makes APIs easy to build and test because it separates side effects from deterministic behavior. For example, say I now want the default value hook passed to `defaultdict` to count the total number of keys that were missing. One way to achieve this is by using a stateful closure (see Item 21: "Know How Closures Interact with Variable Scope" for details). Here, I define a helper function that uses such a closure as the default value hook:

```
def increment_with_report(current, increments):
    added_count = 0

    def missing():
        nonlocal added_count  # Stateful closure
        added_count += 1
        return 0

    result = defaultdict(missing, current)
    for key, amount in increments:
        result[key] += amount

    return result, added_count
```

Running this function produces the expected result (2), even though the `defaultdict` has no idea that the `missing` hook maintains state. Another benefit of accepting simple functions for interfaces is that it's easy to add functionality later by hiding state in a closure:

```
result, count = increment_with_report(current, increments)
assert count == 2
```

The problem with defining a closure for stateful hooks is that it's harder to read than the stateless function example. Another approach is to define a small class that encapsulates the state you want to track:

```
class CountMissing:
    def __init__(self):
        self.added = 0

    def missing(self):
        self.added += 1
        return 0
```

In other languages, you might expect that now `defaultdict` would have to be modified to accommodate the interface of `CountMissing`. But in Python, thanks to first-class functions, you can reference the `CountMissing.missing` method directly on an object and pass it to `defaultdict` as the default value hook. It's trivial to have an object instance's method satisfy a function interface:

```
counter = CountMissing()
result = defaultdict(counter.missing, current)  # Method ref
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

Using a helper class like this to provide the behavior of a stateful closure is clearer than using the `increment_with_report` function, as above. However, in isolation, it's still not immediately obvious what the purpose of the `CountMissing` class is. Who constructs a `CountMissing` object? Who calls the `missing` method? Will the class need other public methods to be added in the future? Until you see its usage with `defaultdict`, the class is a mystery.

To clarify this situation, Python allows classes to define the `__call__` special method. `__call__` allows an `object` to be called just like a function. It also causes the `callable` built-in function to return `True` for such an instance, just like a normal function or method. All objects that can be executed in this manner are referred to as *callables*:

```
class BetterCountMissing:
    def __init__(self):
        self.added = 0

    def __call__(self):
        self.added += 1
        return 0
```

```
counter = BetterCountMissing()
assert counter() == 0
assert callable(counter)
```

Here, I use a `BetterCountMissing` instance as the default value hook for a `defaultdict` to track the number of missing keys that were added:

```
counter = BetterCountMissing()
result = defaultdict(counter, current)  # Relies on __call__
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

This is much clearer than the `CountMissing.missing` example. The `__call__` method indicates that a class's instances will be used somewhere a function argument would also be suitable (like API hooks). It directs new readers of the code to the entry point that's responsible for the class's primary behavior. It provides a strong hint that the goal of the class is to act as a stateful closure.

Best of all, `defaultdict` still has no view into what's going on when you use `__call__`. All that `defaultdict` requires is a function for the default value hook. Python provides many different ways to satisfy a simple function interface, and you can choose the one that works best for what you need to accomplish.

### Things to Remember

- Instead of defining and instantiating classes, you can often simply use functions for simple interfaces between components in Python.
- References to functions and methods in Python are first class, meaning they can be used in expressions (like any other type).
- The `__call__` special method enables instances of a class to be called like plain Python functions.
- When you need a function to maintain state, consider defining a class that provides the `__call__` method instead of defining a stateful closure.

## Item 39: Use `@classmethod` Polymorphism to Construct Objects Generically

In Python, not only do objects support polymorphism, but classes do as well. What does that mean, and what is it good for?

Polymorphism enables multiple classes in a hierarchy to implement their own unique versions of a method. This means that many classes

can fulfill the same interface or abstract base class while providing different functionality (see Item 43: "Inherit from `collections.abc` for Custom Container Types").

For example, say that I'm writing a MapReduce implementation, and I want a common class to represent the input data. Here, I define such a class with a `read` method that must be defined by subclasses:

```
class InputData:
    def read(self):
        raise NotImplementedError
```

I also have a concrete subclass of `InputData` that reads data from a file on disk:

```
class PathInputData(InputData):
    def __init__(self, path):
        super().__init__()
        self.path = path

    def read(self):
        with open(self.path) as f:
            return f.read()
```

I could have any number of `InputData` subclasses, like `PathInputData`, and each of them could implement the standard interface for `read` to return the data to process. Other `InputData` subclasses could read from the network, decompress data transparently, and so on.

I'd want a similar abstract interface for the MapReduce worker that consumes the input data in a standard way:

```
class Worker:
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError
```

Here, I define a concrete subclass of `Worker` to implement the specific MapReduce function I want to apply—a simple newline counter:

```
class LineCountWorker(Worker):
    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')
```

```
    def reduce(self, other):
        self.result += other.result
```

It may look like this implementation is going great, but I've reached the biggest hurdle in all of this. What connects all of these pieces? I have a nice set of classes with reasonable interfaces and abstractions, but that's only useful once the objects are constructed. What's responsible for building the objects and orchestrating the MapReduce?

The simplest approach is to manually build and connect the objects with some helper functions. Here, I list the contents of a directory and construct a `PathInputData` instance for each file it contains:

```
import os

def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))
```

Next, I create the `LineCountWorker` instances by using the `InputData` instances returned by `generate_inputs`:

```
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

I execute these `Worker` instances by fanning out the `map` step to multiple threads (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism" for background). Then, I call `reduce` repeatedly to combine the results into one final value:

```
from threading import Thread

def execute(workers):
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads: thread.start()
    for thread in threads: thread.join()

    first, *rest = workers
    for worker in rest:
        first.reduce(worker)
    return first.result
```

Finally, I connect all the pieces together in a function to run each step:

```
def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

Running this function on a set of test input files works great:

```
import os
import random

def write_test_files(tmpdir):
    os.makedirs(tmpdir)
    for i in range(100):
        with open(os.path.join(tmpdir, str(i)), 'w') as f:
            f.write('\n' * random.randint(0, 100))

tmpdir = 'test_inputs'
write_test_files(tmpdir)

result = mapreduce(tmpdir)
print(f'There are {result} lines')
```

```
>>>
There are 4360 lines
```

What's the problem? The huge issue is that the `mapreduce` function is not generic at all. If I wanted to write another `InputData` or `Worker` subclass, I would also have to rewrite the `generate_inputs`, `create_workers`, and `mapreduce` functions to match.

This problem boils down to needing a generic way to construct objects. In other languages, you'd solve this problem with constructor polymorphism, requiring that each `InputData` subclass provides a special constructor that can be used generically by the helper methods that orchestrate the MapReduce (similar to the factory pattern). The trouble is that Python only allows for the single constructor method `__init__`. It's unreasonable to require every `InputData` subclass to have a compatible constructor.

The best way to solve this problem is with *class method* polymorphism. This is exactly like the instance method polymorphism I used for `InputData.read`, except that it's for whole classes instead of their constructed objects.

Let me apply this idea to the MapReduce classes. Here, I extend the InputData class with a generic @classmethod that's responsible for creating new InputData instances using a common interface:

```
class GenericInputData:
    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

I have generate_inputs take a dictionary with a set of configuration parameters that the GenericInputData concrete subclass needs to interpret. Here, I use the config to find the directory to list for input files:

```
class PathInputData(GenericInputData):
    ...

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```

Similarly, I can make the create_workers helper part of the GenericWorker class. Here, I use the input_class parameter, which must be a subclass of GenericInputData, to generate the necessary inputs. I construct instances of the GenericWorker concrete subclass by using cls() as a generic constructor:

```
class GenericWorker:
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
            workers.append(cls(input_data))
        return workers
```

Note that the call to `input_class.generate_inputs` above is the class polymorphism that I'm trying to show. You can also see how `create_workers` calling `cls()` provides an alternative way to construct `GenericWorker` objects besides using the `__init__` method directly.

The effect on my concrete `GenericWorker` subclass is nothing more than changing its parent class:

```
class LineCountWorker(GenericWorker):
    ...
```

Finally, I can rewrite the `mapreduce` function to be completely generic by calling `create_workers`:

```
def mapreduce(worker_class, input_class, config):
    workers = worker_class.create_workers(input_class, config)
    return execute(workers)
```

Running the new worker on a set of test files produces the same result as the old implementation. The difference is that the `mapreduce` function requires more parameters so that it can operate generically:

```
config = {'data_dir': tmpdir}
result = mapreduce(LineCountWorker, PathInputData, config)
print(f'There are {result} lines')
```

```
>>>
There are 4360 lines
```

Now, I can write other `GenericInputData` and `GenericWorker` subclasses as I wish, without having to rewrite any of the glue code.

### Things to Remember

- Python only supports a single constructor per class: the `__init__` method.
- Use `@classmethod` to define alternative constructors for your classes.
- Use class method polymorphism to provide generic ways to build and connect many concrete subclasses.

## Item 40: Initialize Parent Classes with `super`

The old, simple way to initialize a parent class from a child class is to directly call the parent class's `__init__` method with the child instance:

```
class MyBaseClass:
    def __init__(self, value):
        self.value = value
```

```
class MyChildClass(MyBaseClass):
    def __init__(self):
        MyBaseClass.__init__(self, 5)
```

This approach works fine for basic class hierarchies but breaks in many cases.

If a class is affected by multiple inheritance (something to avoid in general; see Item 41: "Consider Composing Functionality with Mix-in Classes"), calling the superclasses' `__init__` methods directly can lead to unpredictable behavior.

One problem is that the `__init__` call order isn't specified across all subclasses. For example, here I define two parent classes that operate on the instance's `value` field:

```
class TimesTwo:
    def __init__(self):
        self.value *= 2

class PlusFive:
    def __init__(self):
        self.value += 5
```

This class defines its parent classes in one ordering:

```
class OneWay(MyBaseClass, TimesTwo, PlusFive):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

And constructing it produces a result that matches the parent class ordering:

```
foo = OneWay(5)
print('First ordering value is (5 * 2) + 5 =', foo.value)
```

```
>>>
First ordering value is (5 * 2) + 5 = 15
```

Here's another class that defines the same parent classes but in a different ordering (`PlusFive` followed by `TimesTwo` instead of the other way around):

```
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

However, I left the calls to the parent class constructors—`PlusFive.__init__` and `TimesTwo.__init__`—in the same order as before, which means this class's behavior doesn't match the order of the parent classes in its definition. The conflict here between the inheritance base classes and the `__init__` calls is hard to spot, which makes this especially difficult for new readers of the code to understand:

```
bar = AnotherWay(5)
print('Second ordering value is', bar.value)
```

```
>>>
Second ordering value is 15
```

Another problem occurs with diamond inheritance. Diamond inheritance happens when a subclass inherits from two separate classes that have the same superclass somewhere in the hierarchy. Diamond inheritance causes the common superclass's `__init__` method to run multiple times, causing unexpected behavior. For example, here I define two child classes that inherit from `MyBaseClass`:

```
class TimesSeven(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value *= 7

class PlusNine(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value += 9
```

Then, I define a child class that inherits from both of these classes, making `MyBaseClass` the top of the diamond:

```
class ThisWay(TimesSeven, PlusNine):
    def __init__(self, value):
        TimesSeven.__init__(self, value)
        PlusNine.__init__(self, value)

foo = ThisWay(5)
print('Should be (5 * 7) + 9 = 44 but is', foo.value)
```

```
>>>
Should be (5 * 7) + 9 = 44 but is 14
```

The call to the second parent class's constructor, `PlusNine.__init__`, causes `self.value` to be reset back to `5` when `MyBaseClass.__init__` gets called a second time. That results in the calculation of `self.value` to be `5 + 9 = 14`, completely ignoring the effect of the `TimesSeven.__init__`

constructor. This behavior is surprising and can be very difficult to debug in more complex cases.

To solve these problems, Python has the `super` built-in function and standard method resolution order (MRO). `super` ensures that common superclasses in diamond hierarchies are run only once (for another example, see Item 48: "Validate Subclasses with `__init_subclass__`"). The MRO defines the ordering in which superclasses are initialized, following an algorithm called *C3 linearization*.

Here, I create a diamond-shaped class hierarchy again, but this time I use `super` to initialize the parent class:

```
class TimesSevenCorrect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
        self.value *= 7

class PlusNineCorrect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
        self.value += 9
```

Now, the top part of the diamond, `MyBaseClass.__init__`, is run only a single time. The other parent classes are run in the order specified in the `class` statement:

```
class GoodWay(TimesSevenCorrect, PlusNineCorrect):
    def __init__(self, value):
        super().__init__(value)

foo = GoodWay(5)
print('Should be 7 * (5 + 9) = 98 and is', foo.value)
```

```
>>>
Should be 7 * (5 + 9) = 98 and is 98
```

This order may seem backward at first. Shouldn't `TimesSevenCorrect.__init__` have run first? Shouldn't the result be `(5 * 7) + 9 = 44`? The answer is no. This ordering matches what the MRO defines for this class. The MRO ordering is available on a class method called `mro`:

```
mro_str = '\n'.join(repr(cls) for cls in GoodWay.mro())
print(mro_str)
```

```
>>>
<class '__main__.GoodWay'>
<class '__main__.TimesSevenCorrect'>
```

```
<class '__main__.PlusNineCorrect'>
<class '__main__.MyBaseClass'>
<class 'object'>
```

When I call `GoodWay(5)`, it in turn calls `TimesSevenCorrect.__init__`, which calls `PlusNineCorrect.__init__`, which calls `MyBaseClass.__init__`. Once this reaches the top of the diamond, all of the initialization methods actually do their work in the opposite order from how their `__init__` functions were called. `MyBaseClass.__init__` assigns `value` to 5. `PlusNineCorrect.__init__` adds 9 to make `value` equal 14. `TimesSevenCorrect.__init__` multiplies it by 7 to make `value` equal 98.

Besides making multiple inheritance robust, the call to `super().__init__` is also much more maintainable than calling `MyBaseClass.__init__` directly from within the subclasses. I could later rename `MyBaseClass` to something else or have `TimesSevenCorrect` and `PlusNineCorrect` inherit from another superclass without having to update their `__init__` methods to match.

The `super` function can also be called with two parameters: first the type of the class whose MRO parent view you're trying to access, and then the instance on which to access that view. Using these optional parameters within the constructor looks like this:

```
class ExplicitTrisect(MyBaseClass):
    def __init__(self, value):
        super(ExplicitTrisect, self).__init__(value)
        self.value /= 3
```

However, these parameters are not required for `object` instance initialization. Python's compiler automatically provides the correct parameters (`__class__` and `self`) for you when `super` is called with zero arguments within a class definition. This means all three of these usages are equivalent:

```
class AutomaticTrisect(MyBaseClass):
    def __init__(self, value):
        super(__class__, self).__init__(value)
        self.value /= 3

class ImplicitTrisect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
        self.value /= 3

assert ExplicitTrisect(9).value == 3
assert AutomaticTrisect(9).value == 3
assert ImplicitTrisect(9).value == 3
```

The only time you should provide parameters to `super` is in situations where you need to access the specific functionality of a superclass's implementation from a child class (e.g., to wrap or reuse functionality).

### Things to Remember

- Python's standard method resolution order (MRO) solves the problems of superclass initialization order and diamond inheritance.
- Use the `super` built-in function with zero arguments to initialize parent classes.

## Item 41: Consider Composing Functionality with Mix-in Classes

Python is an object-oriented language with built-in facilities for making multiple inheritance tractable (see Item 40: "Initialize Parent Classes with `super`"). However, it's better to avoid multiple inheritance altogether.

If you find yourself desiring the convenience and encapsulation that come with multiple inheritance, but want to avoid the potential headaches, consider writing a *mix-in* instead. A mix-in is a class that defines only a small set of additional methods for its child classes to provide. Mix-in classes don't define their own instance attributes nor require their `__init__` constructor to be called.

Writing mix-ins is easy because Python makes it trivial to inspect the current state of any object, regardless of its type. Dynamic inspection means you can write generic functionality just once, in a mix-in, and it can then be applied to many other classes. Mix-ins can be composed and layered to minimize repetitive code and maximize reuse.

For example, say I want the ability to convert a Python object from its in-memory representation to a dictionary that's ready for serialization. Why not write this functionality generically so I can use it with all my classes?

Here, I define an example mix-in that accomplishes this with a new public method that's added to any class that inherits from it:

```
class ToDictMixin:
    def to_dict(self):
        return self._traverse_dict(self.__dict__)
```

The implementation details are straightforward and rely on dynamic attribute access using `hasattr`, dynamic type inspection with `isinstance`, and accessing the instance dictionary `__dict__`:

```
    def _traverse_dict(self, instance_dict):
        output = {}
        for key, value in instance_dict.items():
            output[key] = self._traverse(key, value)
        return output

    def _traverse(self, key, value):
        if isinstance(value, ToDictMixin):
            return value.to_dict()
        elif isinstance(value, dict):
            return self._traverse_dict(value)
        elif isinstance(value, list):
            return [self._traverse(key, i) for i in value]
        elif hasattr(value, '__dict__'):
            return self._traverse_dict(value.__dict__)
        else:
            return value
```

Here, I define an example class that uses the mix-in to make a dictionary representation of a binary tree:

```
class BinaryTree(ToDictMixin):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

Translating a large number of related Python objects into a dictionary becomes easy:

```
tree = BinaryTree(10,
    left=BinaryTree(7, right=BinaryTree(9)),
    right=BinaryTree(13, left=BinaryTree(11)))
print(tree.to_dict())
```

```
>>>
{'value': 10,
 'left': {'value': 7,
          'left': None,
          'right': {'value': 9, 'left': None, 'right': None}},
 'right': {'value': 13,
           'left': {'value': 11, 'left': None, 'right': None},
           'right': None}}
```

The best part about mix-ins is that you can make their generic functionality pluggable so behaviors can be overridden when required. For example, here I define a subclass of `BinaryTree` that holds a reference to its parent. This circular reference would cause the default implementation of `ToDictMixin.to_dict` to loop forever:

```
class BinaryTreeWithParent(BinaryTree):
    def __init__(self, value, left=None,
                 right=None, parent=None):
        super().__init__(value, left=left, right=right)
        self.parent = parent
```

The solution is to override the `BinaryTreeWithParent._traverse` method to only process values that matter, preventing cycles encountered by the mix-in. Here, the `_traverse` override inserts the parent's numerical value and otherwise defers to the mix-in's default implementation by using the `super` built-in function:

```
    def _traverse(self, key, value):
        if (isinstance(value, BinaryTreeWithParent) and
                key == 'parent'):
            return value.value  # Prevent cycles
        else:
            return super()._traverse(key, value)
```

Calling `BinaryTreeWithParent.to_dict` works without issue because the circular referencing properties aren't followed:

```
root = BinaryTreeWithParent(10)
root.left = BinaryTreeWithParent(7, parent=root)
root.left.right = BinaryTreeWithParent(9, parent=root.left)
print(root.to_dict())
```

```
>>>
{'value': 10,
 'left': {'value': 7,
          'left': None,
          'right': {'value': 9,
                    'left': None,
                    'right': None,
                    'parent': 7},
          'parent': 10},
 'right': None,
 'parent': None}
```

By defining BinaryTreeWithParent._traverse, I've also enabled any class that has an attribute of type BinaryTreeWithParent to automatically work with the ToDictMixin:

```
class NamedSubTree(ToDictMixin):
    def __init__(self, name, tree_with_parent):
        self.name = name
        self.tree_with_parent = tree_with_parent

my_tree = NamedSubTree('foobar', root.left.right)
print(my_tree.to_dict())  # No infinite loop
```

```
>>>
{'name': 'foobar',
 'tree_with_parent': {'value': 9,
                      'left': None,
                      'right': None,
                      'parent': 7}}
```

Mix-ins can also be composed together. For example, say I want a mix-in that provides generic JSON serialization for any class. I can do this by assuming that a class provides a to_dict method (which may or may not be provided by the ToDictMixin class):

```
import json

class JsonMixin:
    @classmethod
    def from_json(cls, data):
        kwargs = json.loads(data)
        return cls(**kwargs)

    def to_json(self):
        return json.dumps(self.to_dict())
```

Note how the JsonMixin class defines both instance methods and class methods. Mix-ins let you add either kind of behavior to subclasses. In this example, the only requirements of a JsonMixin subclass are providing a to_dict method and taking keyword arguments for the __init__ method (see Item 23: "Provide Optional Behavior with Keyword Arguments" for background).

This mix-in makes it simple to create hierarchies of utility classes that can be serialized to and from JSON with little boilerplate. For example, here I have a hierarchy of data classes representing parts of a datacenter topology:

```
class DatacenterRack(ToDictMixin, JsonMixin):
    def __init__(self, switch=None, machines=None):
        self.switch = Switch(**switch)
        self.machines = [
            Machine(**kwargs) for kwargs in machines]

class Switch(ToDictMixin, JsonMixin):
    def __init__(self, ports=None, speed=None):
        self.ports = ports
        self.speed = speed

class Machine(ToDictMixin, JsonMixin):
    def __init__(self, cores=None, ram=None, disk=None):
        self.cores = cores
        self.ram = ram
        self.disk = disk
```

Serializing these classes to and from JSON is simple. Here, I verify that the data is able to be sent round-trip through serializing and deserializing:

```
serialized = """{
    "switch": {"ports": 5, "speed": 1e9},
    "machines": [
        {"cores": 8, "ram": 32e9, "disk": 5e12},
        {"cores": 4, "ram": 16e9, "disk": 1e12},
        {"cores": 2, "ram": 4e9, "disk": 500e9}
    ]
}"""

deserialized = DatacenterRack.from_json(serialized)
roundtrip = deserialized.to_json()
assert json.loads(serialized) == json.loads(roundtrip)
```

When you use mix-ins like this, it's fine if the class you apply `JsonMixin` to already inherits from `JsonMixin` higher up in the class hierarchy. The resulting class will behave the same way, thanks to the behavior of `super`.

### Things to Remember

- Avoid using multiple inheritance with instance attributes and `__init__` if mix-in classes can achieve the same outcome.
- Use pluggable behaviors at the instance level to provide per-class customization when mix-in classes may require it.

- Mix-ins can include instance methods or class methods, depending on your needs.
- Compose mix-ins to create complex functionality from simple behaviors.

## Item 42: Prefer Public Attributes Over Private Ones

In Python, there are only two types of visibility for a class's attributes: *public* and *private*:

```
class MyObject:
    def __init__(self):
        self.public_field = 5
        self.__private_field = 10

    def get_private_field(self):
        return self.__private_field
```

Public attributes can be accessed by anyone using the dot operator on the object:

```
foo = MyObject()
assert foo.public_field == 5
```

Private fields are specified by prefixing an attribute's name with a double underscore. They can be accessed directly by methods of the containing class:

```
assert foo.get_private_field() == 10
```

However, directly accessing private fields from outside the class raises an exception:

```
foo.__private_field
```

```
>>>
Traceback ...
AttributeError: 'MyObject' object has no attribute
➥'__private_field'
```

Class methods also have access to private attributes because they are declared within the surrounding `class` block:

```
class MyOtherObject:
    def __init__(self):
        self.__private_field = 71

    @classmethod
    def get_private_field_of_instance(cls, instance):
        return instance.__private_field
```

```
bar = MyOtherObject()
assert MyOtherObject.get_private_field_of_instance(bar) == 71
```

As you'd expect with private fields, a subclass can't access its parent class's private fields:

```
class MyParentObject:
    def __init__(self):
        self.__private_field = 71

class MyChildObject(MyParentObject):
    def get_private_field(self):
        return self.__private_field

baz = MyChildObject()
baz.get_private_field()
```

```
>>>
Traceback ...
AttributeError: 'MyChildObject' object has no attribute
➥'_MyChildObject__private_field'
```

The private attribute behavior is implemented with a simple transformation of the attribute name. When the Python compiler sees private attribute access in methods like `MyChildObject.get_private_field`, it translates the `__private_field` attribute access to use the name `_MyChildObject__private_field` instead. In the example above, `__private_field` is only defined in `MyParentObject.__init__`, which means the private attribute's real name is `_MyParentObject__private_field`. Accessing the parent's private attribute from the child class fails simply because the transformed attribute name doesn't exist (`_MyChildObject__private_field` instead of `_MyParentObject__private_field`).

Knowing this scheme, you can easily access the private attributes of any class—from a subclass or externally—without asking for permission:

```
assert baz._MyParentObject__private_field == 71
```

If you look in the object's attribute dictionary, you can see that private attributes are actually stored with the names as they appear after the transformation:

```
print(baz.__dict__)
```

```
>>>
{'_MyParentObject__private_field': 71}
```

Why doesn't the syntax for private attributes actually enforce strict visibility? The simplest answer is one often-quoted motto of Python: "We are all consenting adults here." What this means is that we don't need the language to prevent us from doing what we want to do. It's our individual choice to extend functionality as we wish and to take responsibility for the consequences of such a risk. Python programmers believe that the benefits of being open—permitting unplanned extension of classes by default—outweigh the downsides.

Beyond that, having the ability to hook language features like attribute access (see Item 47: "Use `__getattr__`, `__getattribute__`, and `__setattr__` for Lazy Attributes") enables you to mess around with the internals of objects whenever you wish. If you can do that, what is the value of Python trying to prevent private attribute access otherwise?

To minimize damage from accessing internals unknowingly, Python programmers follow a naming convention defined in the style guide (see Item 2: "Follow the PEP 8 Style Guide"). Fields prefixed by a single underscore (like `_protected_field`) are *protected* by convention, meaning external users of the class should proceed with caution.

However, many programmers who are new to Python use private fields to indicate an internal API that shouldn't be accessed by subclasses or externally:

```
class MyStringClass:
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return str(self.__value)

foo = MyStringClass(5)
assert foo.get_value() == '5'
```

This is the wrong approach. Inevitably someone—maybe even you—will want to subclass your class to add new behavior or to work around deficiencies in existing methods (e.g., the way that `MyStringClass.get_value` always returns a string). By choosing private attributes, you're only making subclass overrides and extensions cumbersome and brittle. Your potential subclassers will still access the private fields when they absolutely need to do so:

```
class MyIntegerSubclass(MyStringClass):
    def get_value(self):
        return int(self._MyStringClass__value)
```

```
foo = MyIntegerSubclass('5')
assert foo.get_value() == 5
```

But if the class hierarchy changes beneath you, these classes will break because the private attribute references are no longer valid. Here, the `MyIntegerSubclass` class's immediate parent, `MyStringClass`, has had another parent class added, called `MyBaseClass`:

```
class MyBaseClass:
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return self.__value

class MyStringClass(MyBaseClass):
    def get_value(self):
        return str(super().get_value())         # Updated

class MyIntegerSubclass(MyStringClass):
    def get_value(self):
        return int(self._MyStringClass__value)  # Not updated
```

The `__value` attribute is now assigned in the `MyBaseClass` parent class, not the `MyStringClass` parent. This causes the private variable reference `self._MyStringClass__value` to break in `MyIntegerSubclass`:

```
foo = MyIntegerSubclass(5)
foo.get_value()
```

```
>>>
Traceback ...
AttributeError: 'MyIntegerSubclass' object has no attribute
➥'_MyStringClass__value'
```

In general, it's better to err on the side of allowing subclasses to do more by using protected attributes. Document each protected field and explain which fields are internal APIs available to subclasses and which should be left alone entirely. This is as much advice to other programmers as it is guidance for your future self on how to extend your own code safely:

```
class MyStringClass:
    def __init__(self, value):
        # This stores the user-supplied value for the object.
        # It should be coercible to a string. Once assigned in
        # the object it should be treated as immutable.
```

```
        self._value = value

    ...
```

The only time to seriously consider using private attributes is when you're worried about naming conflicts with subclasses. This problem occurs when a child class unwittingly defines an attribute that was already defined by its parent class:

```
class ApiClass:
    def __init__(self):
        self._value = 5

    def get(self):
        return self._value

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello'  # Conflicts

a = Child()
print(f'{a.get()} and {a._value} should be different')
```

```
>>>
hello and hello should be different
```

This is primarily a concern with classes that are part of a public API; the subclasses are out of your control, so you can't refactor to fix the problem. Such a conflict is especially possible with attribute names that are very common (like `value`). To reduce the risk of this issue occurring, you can use a private attribute in the parent class to ensure that there are no attribute names that overlap with child classes:

```
class ApiClass:
    def __init__(self):
        self.__value = 5       # Double underscore

    def get(self):
        return self.__value    # Double underscore

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello'  # OK!
```

```
a = Child()
print(f'{a.get()} and {a._value} are different')
```

```
>>>
5 and hello are different
```

### Things to Remember

- Private attributes aren't rigorously enforced by the Python compiler.
- Plan from the beginning to allow subclasses to do more with your internal APIs and attributes instead of choosing to lock them out.
- Use documentation of protected fields to guide subclasses instead of trying to force access control with private attributes.
- Only consider using private attributes to avoid naming conflicts with subclasses that are out of your control.

## Item 43: Inherit from collections.abc for Custom Container Types

Much of programming in Python is defining classes that contain data and describing how such objects relate to each other. Every Python class is a container of some kind, encapsulating attributes and functionality together. Python also provides built-in container types for managing data: lists, tuples, sets, and dictionaries.

When you're designing classes for simple use cases like sequences, it's natural to want to subclass Python's built-in `list` type directly. For example, say I want to create my own custom `list` type that has additional methods for counting the frequency of its members:

```
class FrequencyList(list):
    def __init__(self, members):
        super().__init__(members)

    def frequency(self):
        counts = {}
        for item in self:
            counts[item] = counts.get(item, 0) + 1
        return counts
```

By subclassing `list`, I get all of `list`'s standard functionality and preserve the semantics familiar to all Python programmers. I can define additional methods to provide any custom behaviors that I need:

```
foo = FrequencyList(['a', 'b', 'a', 'c', 'b', 'a', 'd'])
print('Length is', len(foo))
```

```
foo.pop()
print('After pop:', repr(foo))
print('Frequency:', foo.frequency())
```

```
>>>
Length is 7
After pop: ['a', 'b', 'a', 'c', 'b', 'a']
Frequency: {'a': 3, 'b': 2, 'c': 1}
```

Now, imagine that I want to provide an object that feels like a `list` and allows indexing but isn't a `list` subclass. For example, say that I want to provide sequence semantics (like `list` or `tuple`) for a binary tree class:

```
class BinaryNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

How do you make this class act like a sequence type? Python implements its container behaviors with instance methods that have special names. When you access a sequence item by index:

```
bar = [1, 2, 3]
bar[0]
```

it will be interpreted as:

```
bar.__getitem__(0)
```

To make the `BinaryNode` class act like a sequence, you can provide a custom implementation of `__getitem__` (often pronounced "dunder getitem" as an abbreviation for "double underscore getitem") that traverses the object tree depth first:

```
class IndexableNode(BinaryNode):
    def _traverse(self):
        if self.left is not None:
            yield from self.left._traverse()
        yield self
        if self.right is not None:
            yield from self.right._traverse()

    def __getitem__(self, index):
        for i, item in enumerate(self._traverse()):
            if i == index:
                return item.value
        raise IndexError(f'Index {index} is out of range')
```

You can construct your binary tree as usual:

```
tree = IndexableNode(
    10,
    left=IndexableNode(
        5,
        left=IndexableNode(2),
        right=IndexableNode(
            6,
            right=IndexableNode(7))),
    right=IndexableNode(
        15,
        left=IndexableNode(11)))
```

But you can also access it like a `list` in addition to being able to traverse the tree with the `left` and `right` attributes:

```
print('LRR is', tree.left.right.right.value)
print('Index 0 is', tree[0])
print('Index 1 is', tree[1])
print('11 in the tree?', 11 in tree)
print('17 in the tree?', 17 in tree)
print('Tree is', list(tree))
```

```
>>>
LRR is 7
Index 0 is 2
Index 1 is 5
11 in the tree? True
17 in the tree? False
Tree is [2, 5, 6, 7, 10, 11, 15]
```

The problem is that implementing `__getitem__` isn't enough to provide all of the sequence semantics you'd expect from a `list` instance:

```
len(tree)
```

```
>>>
Traceback ...
TypeError: object of type 'IndexableNode' has no len()
```

The `len` built-in function requires another special method, named `__len__`, that must have an implementation for a custom sequence type:

```
class SequenceNode(IndexableNode):
    def __len__(self):
        for count, _ in enumerate(self._traverse(), 1):
            pass
        return count
```

```
tree = SequenceNode(
    10,
    left=SequenceNode(
        5,
        left=SequenceNode(2),
        right=SequenceNode(
            6,
            right=SequenceNode(7))),
    right=SequenceNode(
        15,
        left=SequenceNode(11))
)

print('Tree length is', len(tree))
```

```
>>>
Tree length is 7
```

Unfortunately, this still isn't enough for the class to fully be a valid sequence. Also missing are the `count` and `index` methods that a Python programmer would expect to see on a sequence like `list` or `tuple`. It turns out that defining your own container types is much harder than it seems.

To avoid this difficulty throughout the Python universe, the built-in `collections.abc` module defines a set of abstract base classes that provide all of the typical methods for each container type. When you subclass from these abstract base classes and forget to implement required methods, the module tells you something is wrong:

```
from collections.abc import Sequence

class BadType(Sequence):
    pass

foo = BadType()
```

```
>>>
Traceback ...
TypeError: Can't instantiate abstract class BadType with
➥abstract methods __getitem__, __len__
```

When you do implement all the methods required by an abstract base class from `collections.abc`, as I did above with `SequenceNode`, it provides all of the additional methods, like `index` and `count`, for free:

```
class BetterNode(SequenceNode, Sequence):
    pass
```

```
tree = BetterNode(
    10,
    left=BetterNode(
        5,
        left=BetterNode(2),
        right=BetterNode(
            6,
            right=BetterNode(7))),
    right=BetterNode(
        15,
        left=BetterNode(11))
)

print('Index of 7 is', tree.index(7))
print('Count of 10 is', tree.count(10))
```

```
>>>
Index of 7 is 3
Count of 10 is 1
```

The benefit of using these abstract base classes is even greater for more complex container types such as `Set` and `MutableMapping`, which have a large number of special methods that need to be implemented to match Python conventions.

Beyond the `collections.abc` module, Python uses a variety of special methods for object comparisons and sorting, which may be provided by container classes and non-container classes alike (see Item 73: "Know How to Use `heapq` for Priority Queues" for an example).

### Things to Remember

- Inherit directly from Python's container types (like `list` or `dict`) for simple use cases.
- Beware of the large number of methods required to implement custom container types correctly.
- Have your custom container types inherit from the interfaces defined in `collections.abc` to ensure that your classes match required interfaces and behaviors.

*This page intentionally left blank*

# Metaclasses and Attributes

Metaclasses are often mentioned in lists of Python's features, but few understand what they accomplish in practice. The name *metaclass* vaguely implies a concept above and beyond a class. Simply put, metaclasses let you intercept Python's `class` statement and provide special behavior each time a class is defined.

Similarly mysterious and powerful are Python's built-in features for dynamically customizing attribute accesses. Along with Python's object-oriented constructs, these facilities provide wonderful tools to ease the transition from simple classes to complex ones.

However, with these powers come many pitfalls. Dynamic attributes enable you to override objects and cause unexpected side effects. Metaclasses can create extremely bizarre behaviors that are unapproachable to newcomers. It's important that you follow the *rule of least surprise* and only use these mechanisms to implement well-understood idioms.

## Item 44: Use Plain Attributes Instead of Setter and Getter Methods

Programmers coming to Python from other languages may naturally try to implement explicit getter and setter methods in their classes:

```
class OldResistor:
    def __init__(self, ohms):
        self._ohms = ohms

    def get_ohms(self):
        return self._ohms

    def set_ohms(self, ohms):
        self._ohms = ohms
```