Chapter 05 - Classes and Interfaces

May 8, 2021

0.1 Item 37: Compose Classes Instead of Nesting Many Levels of Built-in Types

- pythons built-in dictionary type is wounderful for maintaining dynamic internal state over the lifetime of an object
- by `dynamic` we mean situations where you need to do bookkeeping for unexpected set of identifiers
- imagine you need to redord the grades of a set of students whose names arent known in advance

- **Nesting Many Levels of Built-in Types Means dont put a dictionary in another dictionary or dont put a tuple in a dictionary inside of a dictionary**

- dictionaries and their related built-in types are easy to use that theres a danger of overextending them to write britle code
- for example, say I want to extend the student/grade class feature to keep a `list` of grades by subject, not just overall
- I can do this by chainging the `_grade` dictionary to map student names (its keys) to yet another dictionary (its values)
- the inner most dictionary will map subjects (its keys) to a `list` of grades (its values)
- we will use a `defaultdict` instance for the inner dictionary to handle missing subjects

- imagine that the requirements change agian
- we also want tot track the weight of each score toward the overall grade in the class so that midterm and final exams are more important than pop quizes
- 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`

[8]:
```python
from collections import defaultdict

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))
```

```
    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

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
```

- as you can imagine the complexity is just going to get unmanageable
- this is when we need to make the leap from built-in types like `dictionaries`, `tuples`, `sets` and lists to a hierarchy of classes
- when you realize that the class is getting complicated, break it all out into classes
- you can then provide well-defined interfaces that better encapsulate your data
- this lets you create a layer of abstraction between your interfacesand your concrete implementations

0.1.1 Refactoring to Classes

- we 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
- below I use the tuple of (score, weight) to track grades in a list

[9]:
```
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
```

- note the _ is used in loops where we want to ignore the unused variable
- the problem with the code above is that the `tuple` instances are postitional
- if we want to associate more information with a grade, such as a set of notes from the teacher, we need to rewrite every uage of the two-tuple to be aware that there are now three items present instead
- this means we now have to use multiple _ in the code

[10]:
```
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
```

- the 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
- as soon as you see yourself going longer than a two tuple, its time to reconsider your approach

- the `namedtuple` type in the `collections` built-in module does exactly what we need in this case
- it lets us easily define tiny, immutable data classes

[11]:
```
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 we need to say, support mutability or behaviors in the simple data containers

Limitations of `namedtuple`: - you cant specify default argument values for `namedtuple` class - 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 `dataclass` module may be a better choice - the attribute values of `namedtuple` instances are still accessible using numerical indexes and iterations - especially in externalized APIs, this can lead to unintentional usage that makes it harfer to mvoe to a real `class` later - if your not in control of all of the usages of your `namedtuple` instances, its better to explicitly define a new class

- Next, we write a class to represent a single subject that contains a `set` of grades

[12]:
```
class Subject:
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
```

```python
            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 we write a class to represent a set of subjects that are being studied by a single student

[13]:
```python
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 we write a container for all the students, keyed dynamically by their names

[14]:
```python
class Gradebook:
    def __init__(self):
        self._students = defaultdict(Student)

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

- the line count of these classes in almost dbouble the previous implementation size, but this code is much easier to read

[15]:
```python
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
```

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

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

- many of pythons 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 thats used to determine each index's value for sorting
- Here we sort a `list` of names based on their lenghts by providing the `len` built-in function as the `key` hook

[16]:
```
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 function 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 refrenced like any other value in the language

- for example, say I that I want to customize the behavior of the `defaultdict` class
- 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

[17]:
```
def log_missing():
    print('key added')
    return 0
```

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

[19]:
```
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 the `APIs` easy to build and test because it separates side effects from deterministic behavior
- for example, say I 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 (function inside of function)
- we define a helper function that uses such a closure as the default vale hook

[20]:
```
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 the function produces the expected result `(2)`, even though the **`defaultdict`** has no idea that the `missing` hook maintains state
- antother benefit of accepting simple functions for interfaces is that it's easy to add functionality later by hiding state in a closure

[21]:
```
result, count = increment_with_report(current, increments)
assert count == 2
```

- the problem with defining a closure for stateful hooks is that its harder to read than a statless function
- another approach is to define a small class that encapsulates the state you want to track

[22]:
```
class CountMissing:
    def __init__(self):
        self.added = 0
```

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

- in other languages, you might expect that now `defaultdict` woule have to be modified to accommodate the interface of `CountMissing`
- but in Python, thanks to first-class functions, you can reference the `CountMissing.missing` directly on an object and pass it to `defaultdict` as the default value hook

[23]:
```python
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 clearner then using the `increment_with_report` function
- in isolation howerver, its still not clear what the purpose of the `CountMissing` class is
- who constructs the `CountMissing` object, who calls the `missing` method
- unless you see its usage with `defaultdict`, the class is a mystery

- to celarify the 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 instances, just like a normal function or method
- all objects that can be executed in this manner are reffered to as `callables`

[24]:
```python
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 we use a `BetterCountMissing` instance as the default value hook for a `defaultdict` to track the number of missing keys that were added

[25]:
```python
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 suitible (like API hooks)
- it directs new readers of the code to the entry point thats 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 going on when you use `__call__`
- all that `defaultdict` requires is a function for the default value hook
- python provides different ways to satisfy a simple function interface and you can choose the one that works best for what you need to accomplish

- instead of defining and instantiating classes, you can oftern simply use functions for simple interfaces betweeen components in python
- refrences to functions and methods in python are first class, meaning they can be used in expressions
- the `__call__` special method enables instances of a class to be called like plain python function
- when you need a function to maintain state, consider defining a class that provides the `__cal__` method instead of defining a stateful closure

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

- in python not only do objects support polymorphims, but classes do as well
- 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 providiing different functionality
- for example, samy that I'm writting 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

[65]:
```
class InputData:
    def __init__(self, data):
        self.data = data

    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()
```

- you 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 the network, decompress data transparently, and so on
- I'd want a similar abstract interface for `MapReduce` worker that consumes the input data in a standard way

[15]:
```
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
```

- below I define a concrete subclass of Worker to implement the specific `MapReduce` function I want to apply- a simple newline counter

[16]:
```
class LineCountWorker(Worker):
    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')

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

- one issue we have is that all of the classes are good but what conencts?
- the classes are only useful once the objects are constructed
- what is responsible for building the objects and orchestrating the `MapReduce`?

- the simplest approach is to manually build and connect the objects with some helper functions
- below we list the contents of a directory and construct a `PAthInputData` instance for each file it contains

[17]:
```
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`

[18]:
```
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

- we execute these `Worker` instances by fanning out the `map` step to multiple threads
- then we call `reduce` repeatedly to combine the results into one final value

[56]:
```python
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 we conenct all the pieces together in a function to run each step

[57]:
```python
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 as expected

[58]:
```python
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' + str(random.randint(1,101))
# write_test_files(tmpdir)

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

Problem with the code above: - the huge issue is that the `mapreduce` function is not generic at all - If we wanted to write another `InputData` or `Worker` subclass, I would also have to rewrite the `generate_input`, `create_workers` and `mapreduce` functions to match

- the problem boils donw to needing a generic way to construc objects
- in other programming languages, you could use `constructor polymorphism`
  - you would require that each `InputData` subclass provides a special constructor that can be used generically by the helper methods that orchestrate the `MapReduce` (simmilar to the factory pattern)
- problem in python is that it only allows for a single constructor method `__init__`
- its unreasonable to require every `InputData` subclass to have a compatible constructor

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

- lets apply this idea to the `MapReduce` classes
- we can extend the `InputData class` with a generic `@classmethod` that's responsible for creating new `InputData` instance using a common interface

[59]:
```
class GenericInputData:
    def read(self):
        raise NotImplementedError

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

- we have `generate_inputs` take a dictionary with a set of configuration parameters that the `GenericInputData` concrete subclass needs to interpret
- below I use the `config` to find the directory to list for input files

[60]:
```
class PathInputData(GenericInputData):
    def __init__(self, path):
        super().__init__()
        self.path = path

    def read(self):
        with open(self.path) as f:
            return f.read()
    @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 we can make the `create_workers` helper part of the `GeenricWorker` class
- we use the `input_classs` 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

[61]:
```
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 the `input_class.generate_inputs` above is the class polymorphism that were using to show
- you can see how `create_workers` calling `cls()` provides an alternative way to construct `GenericWorkers` object besides using the `__init__` method directly

- the effect on my concrete `GenericWorker` subclass is nothing more than changing its parent class

[62]:
```
class LineCountWorker(GenericWorker):
    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')

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

finally we can rewrite the `mapreduce` function to be completely generic by calling `create_workers`

[63]:
```
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 `mapreduce` function requires more parameters so that it can operate generically

[64]:
```
config = {'data_dir': tmpdir}
result = mapreduce(LineCountWorker, PathInputData, config)
print(f'There are {result} lines')
```

```
There are 4910 lines
```

- 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 conenct many concrete subclasses

0.4 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

[3]:
```python
class MyBaseClass:
    def __init__(self, value):
        self.value = value

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

- the code works well for basic class hierarchies but brakes in many cases
- if a class is affected by multiple inheritance (something to avoid in general), 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

- below, we define two parent classes that operate on the instance's `value` field

[6]:
```python
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)

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

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

- we can define another class that defines the same parent classes but in a different ordering (`PlusFive` second in the `__init__`)
- we still kept the `TimesTwo.__init__` and `PlusFive.__init__` in the same order
- the classe's behavior doesnot match the order of the parent classes
- the conflict here between the inheritance base classes and the `__inti__` calls is hard to spot, which makes this espically difficult for new readers of the code to understand

[7]:
```python
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)

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 seprate 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

[18]:
```
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 we define a child class that inherits from both of these classes, making `MyBaseClass` the top of the diamond

[19]:
```
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 on 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
- the `MRO` define sthe ordering in which superclasses are initialized, following an algorithm called `C3 linearization`

- below we create a diamond-shaped class hierarchy again, but this time we use `super` to initialize the parent class

[20]:
```
class TimesSevenCorrect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
```

```
        self.value *= 7

class PlusNineCorrect(MyBaseClass):
    def __inti__(self, 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

[24]:
```
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 35
```

- this order may seem backwars ar first meaning, `TimesSevenCorrect.__init__` should have run first
- the result should be `(5 * 7) + 9 == 44` but it is not
- the ordering matched what the `MRO` defines foer the class
- The `MRO` ordering is avaliable on a class method called `mro`

[25]:
```
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 we 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 oppsire order from how their `__init__` functions were called

- Besides making multiple inheritance robust, the call to `super().__init__` is also much more maintainable than calling `MyBaseClass.__init__` directly in 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 whos `MRO` parent view youre trying to access and then the instance on which to access that view
- using these optional parameters within the constructor looks like:

[26]:
```
class ExplicitTrisect(MyBaseClass):
    def __init__(self, value):
        super(ExplicitTrisect, self).__init__(value)
```

```python
        self.value /= 3
```

- those parameters are not needed as pythons complier can do it for you

[27]:
```python
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
```

[28]:
```python
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 specifc functionality of a superclass's implementation from a child class (to wrap or reuse functionality)

- 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

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

- python is an object-oriented language with built-in facilities for marking multiple inheritance tractable (`super`)
- howerver its better to avoid multiple inheritance altogether
- if you want the convenience and encapsulation that comes 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 dont define their own instance attributes nor require their `__init__` constructor to be called

- writing `min-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

- lets 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 that I can use it with all my classes

- below we define an example `mixin` that accomplishes this with a new public method that's added to any class that inherits from it

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

[40]:
```
class ToDictMixin:
    def to_dict(self):
        return self._traverse_dict(self.__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
```

- below we define an example class that uses the `mix-in` to make a dictionary representation of a binary tree

[41]:
```
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 obejcts into a dictionary becomes easy

[42]:
```
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 behavior can be overriden when required
- for example, below I define a subclass of `BinaryTree` that hold a refrence to its parent

- this circular refrence would cause the default implementation of `ToDictMixin.to_dict` to loop forever
- the solution is to override the `BinaryTreeWithParent._traverse` method to only process values that matter, preventing cycles encountered by the `mix-in`
- the `_traverse` override inserts the parents numerical value and otherwise defers to the `mix-ins` default implementation by using the `super` built-functionality

[47]:
```python
class BinaryTreeWithParent(BinaryTree):
    def __init__(self, value, left=None,
                 right=None, parent=None):
        super().__init__(value, left=left, right=right)
        self.parent = parent

    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 refrencing properties arent followed

[48]:
```python
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 the `BinaryTreeWithParent._traverse`, I've also enabled any class that has an attribute type `BinaryTreeWithParent` to automatically work with the `ToDictMixin`

[49]:
```python
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
- lets say I want a mixin 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

[56]:
```python
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 bot 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

- the `mix-in` makes it simple to create hierarchies fo 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

[57]:
```python
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
```

- seralizing these classes to and from JSON is simple
- here we can verify that data is able to be sent round-trip through serializing and deserializing

[58]:
```python
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, its fine if the class you apply `JsonMixin` to already inherit from `JsonMixin` higher up the class hierarchy
- the resulting class will behave the same way, thanks to the behavior of `super`

0.5.1 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 with `mix-in` classes may require
- `mix-ins` can include instance methods or class methods, depending on your needs
- compose `mix-ins` to create complex functionality from simple behaviors

0.6 Item 42: Perfer Public Attributes Over Private Onces

- there are only two types of visibility for a class's attribute: `public` or `private`

[1]:
```
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

[2]:
```
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

[3]:
```
assert foo.get_private_field() == 10
```

- howerver directly accessing private fields from outside the class raises an exception:

[4]:
```
foo.__private_field
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-a888a87e4048> in <module>
----> 1 foo.__private_field

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

[6]:
```python
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

[9]:
```python
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()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-38ca44b74faf> in <module>
      8
      9 baz = MyChildObject()
---> 10 baz.get_private_field()

<ipython-input-9-38ca44b74faf> in get_private_field(self)
      5 class MyChildObject(MyParentObject):
      6     def get_private_field(self):
----> 7         return self.__private_field
      8
      9 baz = MyChildObject()

AttributeError: 'MyChildObject' object has no attribute
 '_MyChildObject__private_field'
```

- accessing the parents private attribute from the child class fails simply because the transformed attribute name doesn't exist
- knowing the schema, you can easily access the private attributes of any class- from a subclass or externally- without asking for permission

[11]:
```python
assert baz._MyParentObject__private_field == 71
```

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

[12]:
```python
print(baz.__dict__)
```

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

- python does not enforce not using private attributes because it belives that "we are all concenting adults here"
- `PEP8` specifes that fields prefixed by a single underscore, `_protected_field` are protected by convention
- new programmers use `__` but should not because when someone want to use the private attribute they will find a way
- document each protected field and explain which fields are internal APIS available to subclasses and which should be left out entirely

[13]:
```python
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

[16]:
```python
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'

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

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

- this is primarly a concenn 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 espically possible with attribute names that are very common (like `value`)

- to reduce the risk of this issue occuring, you can use a private attribute in the parent class to ensure that there are no attribute names that overlap with child classes

[17]:
```
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'

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

```
5 and hello are different
```

0.6.1 Things to Remember

- private attributes arent rigorously enforced by the Python complier
- Plan fromt he 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 contorl

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

- much of programming 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 memebers

[18]:
```
class FrequencyList(list):
    def __init__(self, members):
        super().__init__(members)

    def frequency(self):
```

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

- by subclassing `list`, I get all of `lists` standard functionality and preservice the semantics familiar to Python programmers
- we can define additional methods to provide any custom behaviors that I need

[22]:
```python
foo = FrequencyList(['a', 'b', 'c', 'b', 'a', 'd'])
print('Lenght is', len(foo))

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

```
Lenght is 6
After pop: ['a', 'b', 'c', 'b', 'a']
Frequency: {'a': 2, 'b': 2, 'c': 1}
```

- now imagine that I want to provide an object that feels like a `list` and allows indexing but is not a `list` subclass
- for example, say that I want to provide sequence semantics (like `list` or `tuple` for a binary tree class

[24]:
```python
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

[25]:
```python
bar = [1, 2, 3]
bar[0]
```

[25]: 1

- 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__` that traverses the object tree depth first

[32]:
```python
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')

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

[33]:
```
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 implementing `__getitem__` isent enough to provide all of the sequence semantics you'd expect from a `list` instance
- the `len` built-in function requires another special method named `__len__`, that must have an implementation for a custom sequence type

[34]:
```
class SequenceNode(IndexableNode):
    def __len__(self):
        for count, _ in enumerate(self._traverse(), 1):
            pass
        return count
```

[35]:
```python
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 isent 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 type 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 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

[36]:
```python
from collections.abc import Sequence

class BadType(Sequence):
    pass

foo = BadType()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-997f61651b50> in <module>
      4     pass
      5
----> 6 foo = BadType()

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 done with the `SequenceNode`, it provides all of the additional methods like `index`, and `count` for free

[38]:
```python
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 `MutableMaping`, 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 comparision and sortingm which may be provided by container classes and non-container classes
- 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 interface and behavior