Chapter 06 - Metaclasses and Attributes

May 8, 2021

0.1 Overview

- Metaclasses let you intercept Pythons `class` statement and provide special behavior each time a class is defined
- Pythons built-in features for dynamically customizing attribute access provide wounderful tools to ease the transition from simple classes to complex ones
- Dynamic attributes enable you to override objects and cause unexpected side effects
- metaclasses can create extremely bizarre behaviors that are unapproachable to newcomers

0.2 Item 44 Use Plain Attributes Instead of Setter and Getter Methods

[1]:
```python
class OldResistor:
    def __init__(self, ohms):
        self._ohms = ohms

    def get_ohms(self):
        return self._ohms

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

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

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

- such methods are especially clumsy for operations like incrementing in place

[2]:
```python
r0.set_ohms(r0.get_ohms() - 4e3)
assert r0.get_ohms() == 6e3
```

- these utility methods do, howerver, help the interface for a class, making it easier to encapsulate functionality, validate usage, and define boundaries
- those are important goals when desigdesigning a class to ensure that you don't break callers as the class evolves over time
- in Python, howerver, you never need to implement explicit setter or getter methods

- instead you should always start your implementations with sime public attributes

[3]:
```python
class Resistor:
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0

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

- these attributes make operations like incrementing in place natural and clear

[4]:
```python
r1.ohms += 5e3
```

- later if I decide I need special behavior when an attribute is set, I can migrate to the `@property` decorator and its corresponding `setter` attributes
- below we define a new subclass of Resistor that lets me vary the current by assigning the voltage property
- note in order for this code to work properly, the names of both the setter and the getter methods must match the intended property name

[5]:
```python
class VoltageResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)
        self._voltage = 0

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

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

- now assigning the voltage property will run the voltage setter method, which in turn will update the current attribute of the object to match

[6]:
```python
r2 = VoltageResistance(1e3)
print(f'Before: {r2.current:.2f} amps')
r2.voltage = 10
print(f'After: {r2.current:.2f} amps')
```

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

- specifying a `setter` on a property also enables me to perform type checking and validation on values passed to the class

[7]:
```python
class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

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

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

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-7-dff0c4f8d6eb> in <module>
     14
     15 r3 = BoundedResistance(1e3)
---> 16 r3.ohms = 0

<ipython-input-7-dff0c4f8d6eb> in ohms(self, ohms)
     10     def ohms(self, ohms):
     11         if ohms <= 0:
---> 12             raise ValueError(f'ohms must be > 0; got {ohms}')
     13         self._ohms = ohms
     14

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

- this happens because `BoundedResistance.__init__` calls `Resistor.__init__`, which assigns `self.ohms = -5`
  - the assignment causes the `@ohms.setter` method from `BoundedResistance` to be called and it immediately runs the validation code before objec construction has completed
- we can even use `@property` to make attributes from parent classes immutable

[9]:
```python
class FixedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

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

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

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

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-99062b4be95c> in <module>
     14
     15 r4 = FixedResistance(1e3)
---> 16 r4.ohms = 2e3

<ipython-input-9-99062b4be95c> in ohms(self, ohms)
     10     def ohms(self, ohms):
     11         if hasattr(self, '_ohms'):
---> 12             raise AttributeError("Ohms is immatuble")
     13         self._ohms = ohms
     14

AttributeError: Ohms is immatuble
```

- when you use `@property` methods to implement setters and getter, be sure that the behavior you implement is not surprising
- for example, dont set other attributes in getter property methods

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

- the biggest shortcomming of `@property` is that the methods for an attribute can only be shared by subclasses
- unrelated classes can't share the same implementation
- python does support descriptors that enable reusable property logic and many other use cases

0.2.1 Things to Remeber

- defining new class interfaces using simple public attributes and avoid defining setter and getter methods
- use `@property` to define special behavior when attributes are accessed on your objects, if necessary

- follow the rule of least surprise and avoid odd side effects in your `@peoperty` methods
- ensure that `@property` methods are fast; for slow or complex work- especially involving `I/O` or causing side effect -use normal methods instead

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

- the built-in `@property` decorator makes it easy for simple accesses of an instance's attributes to act smarter
- one advanced but common use of `@property` is transitioning what was one a simple numerical attribute into an on-the-fly calculations
- this is extremely helpful because it lets you migrate all existing usage of a class to have new behaviors without requiring any of the call sites to be rewritten
- `@property` also provides an important stopgap for improving interfaces over time

- lets say we need to implement a leaky buckey quota using plain python objects
- here the `Bucket` class represents how much quota remains and duration for which the quota will be available

[19]:
```
from datetime import datetime, timedelta

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

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

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

[20]:
```
def fill(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
    bucket.quota += amount
```

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

[21]:
```
def deduct(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        return False # Bucket hasn't been filled this period
    if bucket.quota - amount < 0:
        return False # Bucket was filled, but not enough
    bucket.quota -= amount
```

```python
    return True      # Bucket had enough, quota consumed
```

- to use this class, first I fill the bucket up:

[22]:
```python
bucket = Bucket(60)
fill(bucket, 100)
print(bucket)
```

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

- then we deduct the quota the we need

[23]:
```python
if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
print(bucket)
```

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

- eventually we are prevented from making progress because I try to deduct more quota than is available
- in this case, the bucket quota level remains unchanged

[24]:
```python
if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')
print(bucket)
```

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

- the problem with the implementation is that we never know what quota level the bucket started with
- the quota is deducted over the course of the period untill it reaches zero
- at that point, `deduct` will always return `False` untill the bucket is refilled
- when this happend, it would be useful to know whether callers to `deduct` are being blocked because the `Bucket` ran out of quota or because the `Bucket` never had quota during this period in the first place

- to fix this, I can change the class to keep track of the `max_quota` issued in the period and the `quota_consumes` in the period

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

- when the quota attribute is assigned, we take special action to be compatible with the current usage of the class by the `fill` and `deduct` function

[34]:
```python
class NewBucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.max_quota = 0
        self.quota_consumed = 0

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

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

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

- rerunning the demo code from above produces the same results:

[37]:
```python
bucket = NewBucket(60)
print('Initial', bucket)
fill(bucket, 100)
print('Filled', bucket)

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

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

```
print('Still', bucket)
```

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

- best part is that the code using `Bucket.quota` doesent have to change or know that the class has changed
- new usage of `Bucket` can do the right thing and access `max_quots` and `quota_consumed` directly
- `@property` lets you make incremental progress towards a better data model over time
- reading the `Bucket` example above, you may have though that `fill` and `deduct` should have implemented as instance methods in the first place
- the `fill` and `deduct` should have been instance methods but realworld is messy

- `@property` is a toll that helps us address problems in real-world code
- dont overuse it
- when you find yoursel repeatedly extending `@property` methods, its time to refactor your class instead of further paving over your codes poor design

0.3.1 Things To Remember

- use `@property` to give existing instance attributes new functionality
- make incremental progress toward better data models by using `@property`
- consider refactoring a class and all call sitesd when you find yourself using `@property` too heavil

0.4 Item 46: Use Descriptors for Reusable `@property` Methods

- big problem with `@property` built-in is reuse
- the method it decorates can't be reused for multiple attributes of the same class
- they cant also be reused by unrelated classes

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

[1]:
```
class Homework:
    def __init__(self):
        self._grade = 0

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

    @grade.setter
    def grade(self, value):
```

```python
        if not (0 <= value <= 100):
            raise ValueError('Grade must be between 0 and 100')
        self._grade = value


# using @property makes this class easy to use
galileo = Homework()
galileo.grade = 95
```

- say that I also want to give the student a grade for an exam where the exam has multiple subjects each with a seprate grade

[2]:
```python
class Exam:
    def __init__(self):
        self._writing_grade = 0
        self._math_grade = 0

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


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

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

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

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

- this approach is not general
- if we want to resue the precentage validation in other classes beyond homework and exams, we need to write the `@property` boilerplate and `_check_grade` method over and over again

- better way to do this in python is to use a `descriptor`
- the `descriptor protocol` defines how attribute access is interpreted by the language
- a descriptor class can provide `__get__` and `__set__` methods that let you reuse the grade validation behavior

- for this purpose, `descriptors` are better than mix-ins because they let you reuse the same logic for many different attributes in a single class

[4]:
```
class Grade:
    def __get__(self, instance, instance_type):
        pass

    def __set__(self, instance, value):
        pass

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

- it is important to understand what Python will do when such descriptor attributes are accessed on an `Exam` instance

- when we assign a property

[5]:
```
exam = Exam()
exam.writing_grade = 40
```

- it is interpreted as

[7]:
```
Exam.__dict__['writing_grade'].__set__(exam, 40)
```

- when I retrieve a property

[8]:
```
exam.writing_grade
```

- it is interpreted as:

[9]:
```
Exam.__dict__['writing_grade'].__get__(exam, Exam)
```

- what drives this behavior is the `__getattribute__` method of objects
- in short, when an `Exam` instance doesent have an attribute named `writing_grade`, python falls back to the `Exam` class's attribute instead
- if this class attibute is an object that has `__get__` and `__set__` methods, python assumes you want to follow the descriptor protocol

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

[10]:
```
# Incorrect attempt at using descriptors
class Grade:
    def __init__(self):
        self._value = 0

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

```
        return self._value

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

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


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

```
Writing 82
Science 99
```

- accedssing multiple attributes on a single `Exam` instance works as expected as show above
- but accessing these attributes on multiple `Exam` instances causes unexpected behavior

[11]:
```
second_exam = Exam()
second_exam.writing_grade = 75
print(f'Second {second_exam.writing_grade} is right')
print(f'First {first_exam.writing_grade} is wrong; '
 f'should be 82')
```

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

- the problem is that a single `Grade` instance is shared across all `Exam Instances` for the class attribute `writing_grade`
- the `Grade` instance for this attribute is constructed once in the program lifetime,

- to solve this we need the `Grade` class to keep track of its values for each unique `Exam` instance
- I can do this by saving the per-instance state in a dictionary

[12]:
```
class Grade:
    def __init__(self):
        self._values = {}

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

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

- the implementation is simple and works well, but there still one gotcha: it leaks memeory
- the `_value` dictionary holds a refrence to every instance of `Exam` ever passed to `__set__` over the lifetime of the program
- this causes instanes to never have their reference count go to zero, preventing cleanup by the garbage collector

- to fix this, we can use Pythons `weakref` built-in module
- this module provides a special case called `WeakKeyDictionary` that can take the place of the simple dictionary used for `_values`
- the unique behavior of `WeakKeyDictionary` is that is removes `Exam` instances form its set of items when the python runtime knows its holding the instances last remaining refrence in the ptogram
- python does the bookkeeping for us and ensures that the `_value` dictionary will be empty when all `Exam` instances are no longer in use

[21]:
```python
from weakref import WeakKeyDictionary

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

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

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

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


first_exam = Exam()
first_exam.writing_grade = 82
second_exam = Exam()
```

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

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

0.4.1 Things to Remember

- reuse the behavior and validation of `@property` methods by defining your own descriptor classes
- use `WeakKeyDictionary` to ensure that your descriptor classes dont cause memeory leaks
- dont get bogged down trying to understand exactly how `__getattribute__` uses the descriptor protocol for getting and setting attributes

0.5 Item 47: Use `__gettr__`, `__getattribute__`, and `__setattr__` for Lazy Attributes

- pythons object hooks make it easy to write generic code for gluing systems together
- say we want to represent the records in a database as Python Objects
- the database has its schema set already
- the code that uses objects corresponding to these records must also know what the database looks like
- in python the code that connects python objects to the database does not need to explicitly specify the schema of the records: it can be generic

- how is that possible?
- plain instance attributes, `@property` methods and descriptors cant do this because they all need to be defined in advance
- python makes the dynamic behavior possible with the `__getattr__` special method
- if a class defines `__getattr__` that method is called every time an attribute cant be found in an objects dictionary

[22]:
```
class LazyRecord:
    def __init__(self):
        self.exists = 5

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

- here we access the missing property `foo`
- this causes python to call the `__getattr__` method above, which mutates the instance dictionary `__dict___`

[23]:
```
data = LazyRecord()
print('Before', data.__dict__)
print('foo:', data.foo)
```

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

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

- here I add logging to `LazyRecord` to show when `__getattr__` is actually called
- note how we call `super().__getattr__()` to use the super classes implementation of `__getattr__` in order to fetch the real property value and avoid infinite recursion

[25]:
```
class LoggingLazyRecord(LazyRecord):
    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r}), '
            f'populating instance dictionary')
        result = super().__getattr__(name)
        print(f'* Returning {result!r}')
        return result

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

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

- the `exists` attribute is present in the instance dictionary, so `__getattr__` is never called for it
- the `foo` attribute is not in the instance dictionary initially, so `__getattr__` is called for the first time

- but the call to `__getattr__` for `foo` also does a `setattr`, which populates `foo` in the instance dictionary
- this is why the second time I access `foo`, it does not log a call to `__getattr__`

- this behavior is espically helpful for use cases like lazily accessing schemaless data
- `__getattr__` runs once to do the hard work of loading a property: all subsequent accesses to retrieve the existing result

- say that I also want transactions in the database system
- the next time the user accesses a proprty, I want to know whether the corresponding record in the database is still valid and whether the transaction is still open
- the `__getattr__` hook wont let me do this reliably because it will use the objects instance dictionary as the fast path for existing attributes

- to enable this more advanced use case, python has another object called `__getattribute__`
- this special method is called every time an attribute is accessed on an object, even in cases where it does exist in the attribute dictionary

- this enables me to do things like check global transaction state on every property accesses
- it is important to note that such an operation can incur significant overhead and negatively impact performance, but sometimes its work it

- below we define `ValidatingRecord` to log each time `__getattribute__` is called

[27]:
```
class ValidatingRecord:
    def __init__(self):
        self.exists = 5

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

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

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

- in the event that a dynamically accessed property should't exist, I can raise an `AttributeError` to cause pythons standard missing property behavior for both `__getattr__` and `__getattribute__`

[28]:
```
class MissingPropertyRecord:
    def __getattr__(self, name):
        if name == 'bad_name':
            raise AttributeError(f'{name} is missing')

data = MissingPropertyRecord()
data.bad_name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-28-2c3b6b9673c8> in <module>
      5
      6 data = MissingPropertyRecord()
----> 7 data.bad_name

<ipython-input-28-2c3b6b9673c8> in __getattr__(self, name)
      2     def __getattr__(self, name):
      3         if name == 'bad_name':
----> 4             raise AttributeError(f'{name} is missing')
      5
      6 data = MissingPropertyRecord()

AttributeError: bad_name is missing
```

- python code implementing generic functionality often relies on the `hasattr` built-in function to determin when properties exist and the `getattr` built-in function to retrieve property values
- these functions also look in the instance dictionary for an attribute name before calling `__getattr__`

[29]:
```
data = LoggingLazyRecord() # Implements __getattr__
print('Before: ', data.__dict__)
print('Has first foo: ', hasattr(data, 'foo'))
print('After: ', data.__dict__)
print('Has second foo: ', hasattr(data, 'foo'))
```

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

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

[30]:
```
data = ValidatingRecord() # Implements __getattribute__
print('Has first foo: ', hasattr(data, 'foo'))
print('Has second foo: ', hasattr(data, 'foo'))
```

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

```
Has second foo:  True
```

- lets say we want to lazily push data back to the database when values are assigned to my python object
- we can do this with `__setattr__`, a simmilar `object` hook that lets you intercept arbitrary attribute assignments
- unlike when retrieving an attribute when `__getattr__` and `__getattribute__`, there no need for two seprate methods
- the `__setattr__` method is always called every time an attribute is assigned on an instance

[31]:
```python
class SavingRecord:
    def __setattr__(self, name, value):
        # save some data for the record
        # ...
        super().__setattr__(name, value)
```

- here I define a logging subclass of `SavingRecord`
- its `__setattr__` method is always called on each attribute assignment

[32]:
```python
class LoggingSavingRecord(SavingRecord):
    def __setattr__(self, name, value):
        print(f'* Called __setattr__({name!r}, {value!r})')
        super().__setattr__(name, value)

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

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

- the problem with `__getattribute__` and `__setattr__` is that they are called on every attibute access for an object, even when you may not want that to happen

- imagine I want attribute access on my objects to look up keys in an associated dictionary

[33]:
```python
class BrokenDictionaryRecord:
    def __init__(self, data):
        self.data = {}

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

- this requires accessing `self._data` form the `__getattribute__` methods

- howerver if I actually try to do that, python will recure untill it reaches its stack limit and then die

[34]:
```
# following code causes error
# data = BrokenDictionaryRecord({'foo': 3})
# data.foo
```

- the problem is that `__getattribute__` access `self._data` which causes `__getattribute__` to run again, which accesses `self._data` again and so on
- the solution is to use the `super().__getattribute__` method to fetch the values form the instance attribute dictionary to avoid recursion

[35]:
```
class DictionaryRecord:
    def __init__(self, data):
        self._data = data

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

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

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

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

0.5.1 Things to Remember

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

0.6 Item 48: Validate Subclasses with `__init_subclass__`

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

- often a class's validation code runs in the `__init__` method when an object of the class's type is constructed in `runtime`
- using metaclasses for validation can raise errors much earlier, such as when the module containing the class is first imported at program startup

- before we get into how to define a metaclass for validating subclasses, its important to understand the metaclass action for standard objects
- a metaclass is defined by inheriting from `typ`
- in the default class, a metaclass receives the contents of associated `class` statements in its `__new__` method
- below we can inspect and modify the class information before the type is actually constructed

[5]:
```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print(f'* Running {meta}.__new__ for {name}')
        print('Bases', bases)
        print(class_dict)
        return type.__new__(meta, name, bases, class_dict)

class MyClass(metaclass=Meta):
    stuff = 123

    def foo(self):
        pass

print("")

class MySubClass(MyClass):
    other = 567

    def bar(self):
        pass
```

```
* Running <class '__main__.Meta'>.__new__ for MyClass
Bases ()
{'__module__': '__main__', '__qualname__': 'MyClass', 'stuff': 123, 'foo':
<function MyClass.foo at 0x000001D9ABA94CA0>}

* Running <class '__main__.Meta'>.__new__ for MySubClass
Bases (<class '__main__.MyClass'>,)
{'__module__': '__main__', '__qualname__': 'MySubClass', 'other': 567, 'bar':
<function MySubClass.bar at 0x000001D9ABAA43A0>}
```

- the metaclass has access to the names of the classes, the parent classes it inherits from (`bases`), and all the class attributes that were defined in the `class's` body
- all classes inherit from `object`, so its not explicitly listed in the `tuples` base classes

- we can add functionality to the `Meta.__new__` method in order to validate all of the parameters of an associated class before its defined
- say that I want to represent any type of multi-sided polygon
- we can do this by defining a special validating metaclass and using it in the base class of my polygon class hierarchy

[17]:
```python
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Polygon class
        if bases:
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

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

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

class Triangle(Polygon):
    sides = 3

class Rectangle(Polygon):
    sides = 4

class Nonagon(Polygon):
    sides = 9

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

- if I try to define a polygon with fewer than three sides, the validation will cause the `class` statement to fail immediately aftet the `class` statement body
- this means the program will not even be able to start running when I define such a class

[18]:
```python
print('Before class')

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

print('After class')
```

```
Before class
Before sides
After sides
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-18-eaa16278af3b> in <module>
```

```
      1 print('Before class')
      2
----> 3 class Line(Polygon):
      4     print('Before sides')
      5     sides = 2

<ipython-input-17-80258a6a7e92> in __new__(meta, name, bases, class_dict)
      4         if bases:
      5             if class_dict['sides'] < 3:
----> 6                 raise ValueError('Polygons need 3+ sides')
      7         return type.__new__(meta, name, bases, class_dict)
      8

ValueError: Polygons need 3+ sides
```

- all that above seems like alot of machinery in order to get `Python` to accomplish such a basic task
- we can howerver use the syntax `__init__subclass__` special class method- for achieving the special behavior while avoiding metaclasses entirely

[22]:
```
class BetterPolygon:
    sides = None # Must be specified by subclasses
    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.sides < 3:
            raise ValueError('Polygons need 3+ sides')

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

class Hexagon(BetterPolygon):
    sides = 6

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

- we can access the `sides` attribute directly on the `cls` instance in `__init_subclas__` instead of having to go into the class dictionary with `class_dict['sides']`
- if I define an invalid subclass of `BetterPolygon`, the same exception is raise

[23]:
```
print('Before class')

class Point(BetterPolygon):
    sides = 1

print('After class')
```

```
Before class
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-3df430f0fbb3> in <module>
      1 print('Before class')
      2
----> 3 class Point(BetterPolygon):
      4     sides = 1
      5

<ipython-input-22-437ce10fa1df> in __init_subclass__(cls)
      4         super().__init_subclass__()
      5         if cls.sides < 3:
----> 6             raise ValueError('Polygons need 3+ sides')
      7
      8     @classmethod

ValueError: Polygons need 3+ sides
```

- another problem with the standard Python metaclass machinery is that you can only specify a signle metaclass per class definition
- here, we define a second metaclass that I'd like to use for validating the fill color used for a region (not necessarily just polygons)

[26]:
```
class ValidateFilled(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Filled class
        if bases:
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
        return type.__new__(meta, name, bases, class_dict)

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

- when we try to use the `Polygon` metaclass and `Filled` metaclasses together, we get a cryptic `error` message

[27]:
```
class RedPentagon(Filled, Polygon):
    color = 'red'
    sides = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-ddfb321e600a> in <module>
----> 1 class RedPentagon(Filled, Polygon):
      2     color = 'red'
      3     sides = 5
```

```
TypeError: metaclass conflict: the metaclass of a derived class must be a␣
↪(non-strict) subclass of the metaclasses of all its bases
```

- its possible to fix this by creating a complex hierarchy of metaclass `type` definition to layer validation
- but this ruines composability
- if we want to use apply the color validation logic from `ValidateFilledPolygon` to another hierarchy of classes, we have to duplicat eall of the logic again, which reduces code reuse and increases boilerplate

- the `__init_subclass__` special class method can also be used to solve this problem
- it can be defined by multiple levels of a class hierarchy as long as the `super` built-in function is used to call any parent or siblings `__init_subclass__` definitions
- here we define a class to represent region fill color that can be composed with `BetterPolygon` class from before

[31]:
```python
class Filled:
    color = None # Must be specified by subclasses

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

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

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

- if we specify the number of sides incorrectly, we get a validation error

[33]:
```python
print('Before class')

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

print('After class')
```

```
Before class
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-c09f5b0b6f3e> in <module>
      1 print('Before class')
```

```
      2
----> 3 class BlueLine(Filled, Polygon):
      4     color = 'blue'
      5     sides = 2

<ipython-input-17-80258a6a7e92> in __new__(meta, name, bases, class_dict)
      4         if bases:
      5             if class_dict['sides'] < 3:
----> 6                 raise ValueError('Polygons need 3+ sides')
      7         return type.__new__(meta, name, bases, class_dict)
      8

ValueError: Polygons need 3+ sides
```

- if we specify the color incorrectly, we also get a validation error

[35]:
```
print('Before class')

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

print('After class')
```

```
Before class
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-35-8042334559da> in <module>
      1 print('Before class')
      2
----> 3 class BeigeSquare(Filled, Polygon):
      4     color = 'beige'
      5     sides = 4

<ipython-input-17-80258a6a7e92> in __new__(meta, name, bases, class_dict)
      5             if class_dict['sides'] < 3:
      6                 raise ValueError('Polygons need 3+ sides')
----> 7         return type.__new__(meta, name, bases, class_dict)
      8
      9 class Polygon(metaclass=ValidatePolygon):

<ipython-input-31-57af4bbf8b90> in __init_subclass__(cls)
      5         super().__init_subclass__()
      6         if cls.color not in ('red', 'green', 'blue'):
----> 7             raise ValueError('Fills need a valid color')
      8
```

```
      9 class RedTriangle(Filled, Polygon):

ValueError: Fills need a valid color
```

- you can even use `__init_subclass__` in complex cases like diamond inheritance
- below we define a basic diamong hierarchy to show this in action

[36]:
```python
class Top:
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Top for {cls}')

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

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

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

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

- as expected, the `Top.__init_subclass` is called only a single time for each class, even though there are two paths to it for the `Bottom` class its `Left` and `right` parent classes

0.6.1 Things to Remember

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

0.7 Item 50: Annotate Class Attributes with `__set_name__`

- a useful feature enabled by metaclasses is the ability to modify or annotate properties after a class is define but before the class is actually used
- this approach is commonly used with the `descriptors` to give them more introspection into how they're being used within their containing class
- say we want to define a new class that represents a row in a customer database
- we would like to have a corresponding property on the class for each column in the database table

- below we define a descriptor class to connect attributes to column names

[38]:
```
class Field:
    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name

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

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

- With the column name stored in the `Field` descriptor, we can save all of the per-instance state directly in the instance dictionary as protected fields by using the `setattr` built-in function, and later I can load state with `getattr`
- at first, this seems to be much more convient than building `descriptors` with the `weakref` built-in module to avoid memory leaks
- defining the class representing a row requires supplying the database tables column name for each class attribute

[39]:
```
class Customer:
    # Class attributes
    first_name = Field('first_name')
    last_name = Field('last_name')
    prefix = Field('prefix')
    suffix = Field('suffix')
```

- using the class is simple
- here we can see how the `Field` descriptors modify the intance dictionary `__dict__` as expected

[42]:
```
cust = Customer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Euclid'
print(f'After: {cust.first_name!r} {cust.__dict__}')
```

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

- but the class definition seems redundant
- we already declared the name of the field for the class on the left (`field_name =`)
- why do we also have to pass a string containing the same information to the `Field` constructor (`Field('first_name')`) on the right

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

- the problem is that the order of operation in the `Customer` class definition is the opposite of how it reads from left to right
- first the `Field` constructor is called as `Field('first_name')`
- then the return value of that is assigned to `Customer.field_name`
- there is no way for a `Field` instance to know upfront which class attribute it will be assigned to

- to eliminate this redundancy we can use a `metaclass`
- `metaclasses` let you hook into the `class` statement directly and take action as soon as a `class` body is finished
- in this case, we can use the metaclass to assign `Field.name` and `Field.internal_name` on the descriptor automatically instead of manually specifying the field name multiple times

[43]:
```python
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        for key, value in class_dict.items():
            if isinstance(value, Field):
                value.name = key
                value.internal_name = '_' + key
        cls = type.__new__(meta, name, bases, class_dict)
        return cls
```

- here we define a base class that uses the metaclass
- all metaclasses representig batabase rows should inherit from the class to ensure that they use `metaclasses`

[44]:
```python
class DatabaseRow(metaclass=Meta):
    pass
```

- to work with metaclass, the `Field` descriptor is largely unchanged
- the only difference is that it no longer requires arguments to be passed to its constructor
- instead, its attributes are set by the `Meta.__new__` method

[45]:
```python
class Field:
    def __init__(self):
        # these will be assigned by the metaclass
        self.name = None
        self.internal_name = None
```

```
    def __get__(self, instance, instance_tpye):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, "")

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

- by using the metaclasses, the new `DatabaseRow` base class, and the new `Field` descriptor, the class definition for a database row no longer has the redundancy from before

[49]:
```
class BetterCustomer(DatabaseRow):
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()
```

- this behavior of the new class is identical to the behavior of the old one

[51]:
```
cust = BetterCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Euler'
print(f'After: {cust.first_name!r} {cust.__dict__}')
```

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

- the trouble with this approach is that you cant use the `Field` class for properties unless you also inherit from `DatabaseRow`
- the solution to this problem is to use the `__set_name__` special method - this method is called for every descriptor instance when its containing class is defined
- it receives as parameters the owning class that contains the descriptor instance and the attribute name to which the descriptor instance assigned
- here we avoid defining a metaclass entirely and move what the `Meta.__new__` method from above was doing into `__set_name__`:

[52]:
```
class Field:
    def __int__(self):
        self.name = None
        self.internal_name = None

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

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
```

```
        return getattr(instance, self.internal_name, '')

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

- now we get the benefits of the `Field` descriptor without having to inherit from a specific parent class or having to use a metaclass

[54]:
```
class FixedCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()

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

```
Before: '' {}
After: 'Mersenne' {'first_name': 'Mersenne'}
```

0.7.1 Things to Remember

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

0.8 Item 51: Perfer Class Decorators Over Metaclasses for Composable Class Extensions

- although metaclasses allow you to customize class creation in multiple ways, they still falls short of handling every situation that may arise
- for example, if we want to decorate all of the methods of a class with a helper that prints arguments, return values, and exceptions raised

[75]:
```
from functools import wraps
def trace_func(func):
    if hasattr(func, 'tracing'): # Only decorate once
        return func
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = None
        try:
            result = func(*args, **kwargs)
```

```
            return result
        except Exception as e:
            result = e
            raise
        finally:
            print(f'{func.__name__}({args!r}, {kwargs!r}) -> '
                    f'{result!r}')
    wrapper.tracing = True
    return wrapper
```

- it can apply this decorator to various special methods the new `dict` subclass

[76]:
```
class TraceDict(dict):
    @trace_func
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

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

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

- we can verify that these methods are decorated by interacting with an instance of the class

[78]:
```
trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']

try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected
```

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

- the problem with the code is that we had to redefine all the methods that we wanted to decorate with `@trace_func`
- this is redundant boiler-plate that's hard to read and error prone
- further if a new method is later added to the dict superclass, it wont be decorated ubless we also define it in `TraceDict`

- one way to solve this problem is to use a metaclass to automatically decorate all methods of a class

- here we implement this behavior by wrapping each function or method in the new type with the `trace_func` decorator

[88]:
```python
import types

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

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

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

- now I can declare my `dict` subclass by using the `TraceMeta` metaclass and verify that it works as expected

[85]:
```python
class TraceDict(dict, metaclass=TraceMeta):
    pass


trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']

try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected
```

- this works and it even prints out a call to `__new__` that was missing rom my earlier implementation
- what happens if we try to use `TraceMeta` when a superclass already has specified a metaclass?

[86]:
```python
class OtherMeta(type):
    pass

class SimpleDict(dict, metaclass=OtherMeta):
```

```
    pass

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-f9cd7b9c5140> in <module>
      5     pass
      6
----> 7 class TraceDict(SimpleDict, metaclass=TraceMeta):
      8     pass

TypeError: metaclass conflict: the metaclass of a derived class must be a␣
 ↪(non-strict) subclass of the metaclasses of all its bases
```

- it fails because `TraceMeta` does not inherit from `OtherMeta`
- we could use `metclass` inheritance to solve the problem by having `OtherMeta` inherit from `TraceMeta`

[89]:
```
class TraceMeta(type):
    def __new__(meta, name, bases, class_dict):
        klass = super().__new__(meta, name, bases, class_dict)

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

class OtherMeta(TraceMeta):
    pass

class SimpleDict(dict, metaclass=OtherMeta):
    pass

class TraceDict(SimpleDict, metaclass=TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']

try:
    trace_dict['does not exist']
```

```
except KeyError:
    pass # Expected
```

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

- the problem is that it wont work if the metaclass is from a library that we cant modify or if we want to use multiple utility metaclasses like `TraceMeta` at the same time
- the metaclass approach puts too many constraints on the class that being modified

- we can use class `decorators`
- class `decorators` work just like function decorators
- the function is expected to modfiy or `re-create` the class accordingly and then return it

[90]:
```
def my_class_decorator(klass):
    klass.extra_param = 'hello'
    return klass

@my_class_decorator
class MyClass:
    pass

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

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

- we can implement a class decorator to apply `trace_func` to all methods and functions of a class by moving the core of the `TraceMeta.__new__` method aboce into a stand-alone function

[91]:
```
def trace(klass):
    for key in dir(klass):
        value = getattr(klass, key)
        if isinstance(value, trace_types):
            wrapped = trace_func(value)
            setattr(klass, key, wrapped)
    return klass
```

- we can apply this decorator to my `dict` subclass to get the sme behavior as I get by using the metaclass approach

[93]:
```
@trace
class TraceDict(dict):
    pass

trace_dict = TraceDict([('hi', 1)])
```

```python
trace_dict['there'] = 2
trace_dict['hi']

try:
    trace_dict['does not exist']
except:
    pass # Excepted
```

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

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

[94]:
```python
class OtherMeta(type):
    pass

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

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']

try:
    trace_dict['does not exist']
except:
    pass # Excepted
```

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

- when your looking for composable ways to extend classes, class decorators are the best tool for the job

0.8.1 Things to Remeber

- a class decorator is a simple function that receives a `class` instance as a parameter and returns either a new class or modified version of the original class
- class decorators are useful when you want to modify every method or attribute of a class with minimal boilerplate
- metaclasses cant be composed together easily, while many class decorators can be used to extend the same class without conflicts