The Core Python Object Model

The Python Object Model

Objects

- Python is an object oriented language
- Everything is an object: lists, None, exceptions, classes, numbers, etc…
- Objects combine data with functions for operating on the data – their "methods"

```
a = 'Hello World'           # A string object
b = a.upper()               # A method applied to the string

items = [1,2,3]             # A list object
items.append(4)             # A method applied to the list
```

Methods as Functions

A helpful way to think about methods is as a function that takes the object as the first argument.

```
>>> string = 'Hello world'
>>> str.upper(string)
'HELLO WORLD'
>>> a = [1, 2, 3]
>>> list.append(a, 4)
>>> a
[1, 2, 3, 4]
```

And as we'll find out, this is in fact what is happening under the hood. Objects combine data and functions for working on the data (and some interesting rules about how they operate).

The Class Statement

- We can define custom classes
- Type and class are synonyms

```python
class Frobulator:

    def __init__(self, data):
        self.data = data
        self.version = 1.0

    def update(self, version):
        self.version = version

    def frobulate(self):
        return self.data
```

- The body of the class is the collection of methods

Methods and Explicit Self

- The class definition/class object is the type
- Individual objects are "instances" of the type
- When we call a method the instance is passed in (automatically) as the first argument – called "self"
- Methods are functions that operate on instances

```
>>> frob = Frobulator(data)
>>> frob.update(2.0)


def update(self, version):
    self.version = version
```

- In other languages self is called "this" and isn't declared explicitly in the method definition

Instantiation

- Instances are created by calling the class
- Every instance can be created with its own data
- Everything that lives on the class (like methods) are shared by all the instances

```
>>> frob1 = Frobulator(79)
>>> frob2 = Frobulator(130)
>>> frob1.data
79
>>> frob2.data
130
>>> frob2.update(2.0)
>>> frob1.version
1.0
>>> frob2.version
2.0
```

The Initialiser

- __init__ is the initialiser, it initialises the instance
- One of Python's protocol (magic/dunder) methods
- Part of the "two phase object creation protocol"
- Similar to a "constructor" in other languages
- __init__ is called automatically on instantiation
- Typically they set attributes, the instance data

```
>>> frob1 = Frobulator(108)


def __init__(self, data):
    self.data = data
    self.version = 1.0

>>> frob1.data
108
```

Attributes

- Data and methods are all object attributes
- There are three attribute operations on objects

```
x = obj.attr              # Get an attribute
obj.attr = value          # Set an attribute
del obj.attr              # Delete an attribute
```

- Objects are "open by default" – new attributes can be created and existing ones modified/deleted

```
>>> del frob1.version
>>> frob1.supersonic = True
```

Note: attribute deletion is rare

Attribute Access Functions

- Four functions give us access to object attributes

```
getattr(obj, 'attribute')         # Fetch an attribute
setattr(obj, 'attribute', value) # Set an attribute
delattr(obj, 'attribute')         # Delete an attribute
hasattr(obj, 'attribute')         # Test if an attribute
exists

attributes = [ 'name', 'shares', 'price']
for attr in attributes:
    print(attr, '=', getattr(stock, attr))
```

- getattr has an optional third argument – a default value for non-existent attributes (useful for duck typing)

```
getattr(stock, date, 'today')
```

Bound Methods

- Calling a method is two step, first the lookup and then the call
- Looking up a method returns a "bound method" – self is already bound in as the first argument

```
>>> frob
<__main__.Frobulator object at 0xc7c40>
>>> method = frob.frobulate
>>> method
<bound method Frobulator.frobulate of <__main__.Frobulator object at 0xc7c40>>
>>> method()
79
```

The instance is bound into the method

More on Classes

- Everything that lives on the class is shared by <u>all instances</u>
- For example the update method on Frobulator
- A function defined on the class becomes a method on all instances
- Defined once, used in many places
- So far we've only looked at methods but there are other things we can put in the class definition

Class Level Attributes

- As well as methods classes can have attributes
- Class level attributes or variables

```
class DataConnection:

    active = True

    def __init__(self, address, port):
        self.socket_address = address, port
        ...
```

- Class attributes are shared by all instances
- Accessible via the class *and* via the instances

Class Level Attributes

```
>>> conn1 = DataConnection(addr1, port1)
>>> conn2 = DataConnection(addr2, port2)
>>> DataConnection.active
True
>>> conn1.active
True
>>> conn2.active
True
>>> DataConnection.active = False
>>> conn1.active
False
>>> conn2.active = True
>>> conn1.active
False
>>> DataConnection.active
False
>>> conn2.active
True
```

- Setting the attribute on an instance *shadows* (does not change) the class level attribute

Using Class Attributes

- Class attributes are useful for anything shared by all instances
  - Default settings
  - Configuration
  - Caches
  - With inheritance for customisation

Note: Class attributes for default settings can be a performance optimisation – the initialiser doesn't need to set default values for attributes as the defaults live on the class. Instances are free to override (shadow) the class level default.

Class Attributes and Inheritance

- An example for customisation via inheritance

```
class Date:

    datefmt = '{year}-{month}-{day}'

    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __str__(self):
        return self.datefmt.format(year=self.year,
                                   month=self.month,
                                   day=self.day)

class USDate(Date):
    datefmt = '{month}/{day}/{year}'
```

Problem: Simple Attributes

- Objects, classes and instances are open by default
- We can create new attributes and set any value
- This is powerful and useful, but what if we want to prevent users setting the wrong type

```
>>> stock = Stock('IBM', 300, 479.25)
>>> stock.shares
300
>>> stock.shares = '450'
```

Setting shares to a string is going to cause a crash later in our code!

Private Attributes

- Some languages solve this by using getter and setter methods and private attributes
- In Python a private attribute is signalled with a leading underscore (see next slide)
- This is a convention only, but widely understood, and really useful for testing (etc)

Note: Languages that have data privacy tend to also provide reflection (or have pointers) that can circumvent privacy – privacy is also a convention in those languages.

Properties

- Properties wrap getter [and optionally] setter methods but are used just like attributes

```
class Stock:

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property           # A "get only" property
    def cost(self):
        return self.shares * self.price

>>> stock = Stock('GOOG', 100, 490.10)
>>> stock.cost
49010.0
```

Properties

Wrapping a setter as well to make shares a property:

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    @property
    def shares(self):
        return self._shares

    @shares.setter
    def shares(self, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        self._shares = value
```

Note: we don't need to change any existing code, *including* this call in `__init__`.

Setting "shares" triggers the property setter and we now get type checking in the initialiser for free.

```
>>> stock = Stock('GOOG', 100, 490.10)
>>> stock.shares = '50'
TypeError: Expected an int
```

The property prevents this

__slots__ Attribute

- __slots__ on a class limits the available attributes
- Attempting to set other attributes will error
- This is a memory optimisation!

```
class Stock:
    __slots__ = ('name', 'shares', 'price')
    ...

>>> stock = Stock('GOOG', 100, 490.10)
>>> stock.foo = 33
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Stock' object has no attribute 'foo'
```

Warning: creates "non-standard" Python objects!

Inheritance

- A tool for specialising existing objects
- A tool for code reuse!
- Without an explicit base class classes inherit from object

```
class Parent:
    ...

class Child(Parent):
    ...
```

- The parent – also superclass or base class
- The child – also subclass or derived type
- Inheritance can modify and extend the parent

Adding New Methods

- Extend an existing class by adding new methods
- The child has all the behaviour/features of the parent *plus* the new method

```
class MyStock(Stock):
    def panic(self):
        self.sell(self.shares)

>>> stock = MyStock('GOOG', 100, 490.10)
>>> stock.shares
100
>>>
>>> stock.panic()
>>> stock.shares
0
```

Overriding Methods

- Modify classes by replacing (overriding or shadowing) existing methods
- Everything else behaves as the parent

```
class MyStock(Stock):
    def cost(self):
        return 1.25 * self.shares * self.price

>>> stock = MyStock('GOOG', 100, 490.10)
>>> stock.cost()
61262.5
```

Calling the Parent

- It's common for a child class to modify the parent behaviour without replacing it altogether
- Use super() to call up to the parent

```
class Stock:
    ...
    def cost(self):
        return self.shares * self.price
    ...

class MyStock(Stock):
    def cost(self):
        real_cost = super().cost()
        return real_cost * 1.25
```

Inheritance and __init__

- With inheritance you must initialise your parents

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price


class MyStock(Stock):
    def __init__(self, name, shares, price, factor):
        super().__init__(name, shares, price)
        self.factor = factor

    def cost(self):
        return self.factor * super().cost()
```

Inheritance

- Inheritance establishes an "is a" relationship
- Both Object Oriented theory and in the type system

```
class MyStock(Stock):
    ...

>>> stock = MyStock('GOOG', 100, 490.10)
>>> type(stock)
<class '__main__.MyStock'>
>>> isinstance(stock, Stock)
True
>>> isinstance(stock, object)
True
>>> issubclass(Stock, object)
True
```

Composition Over Inheritance

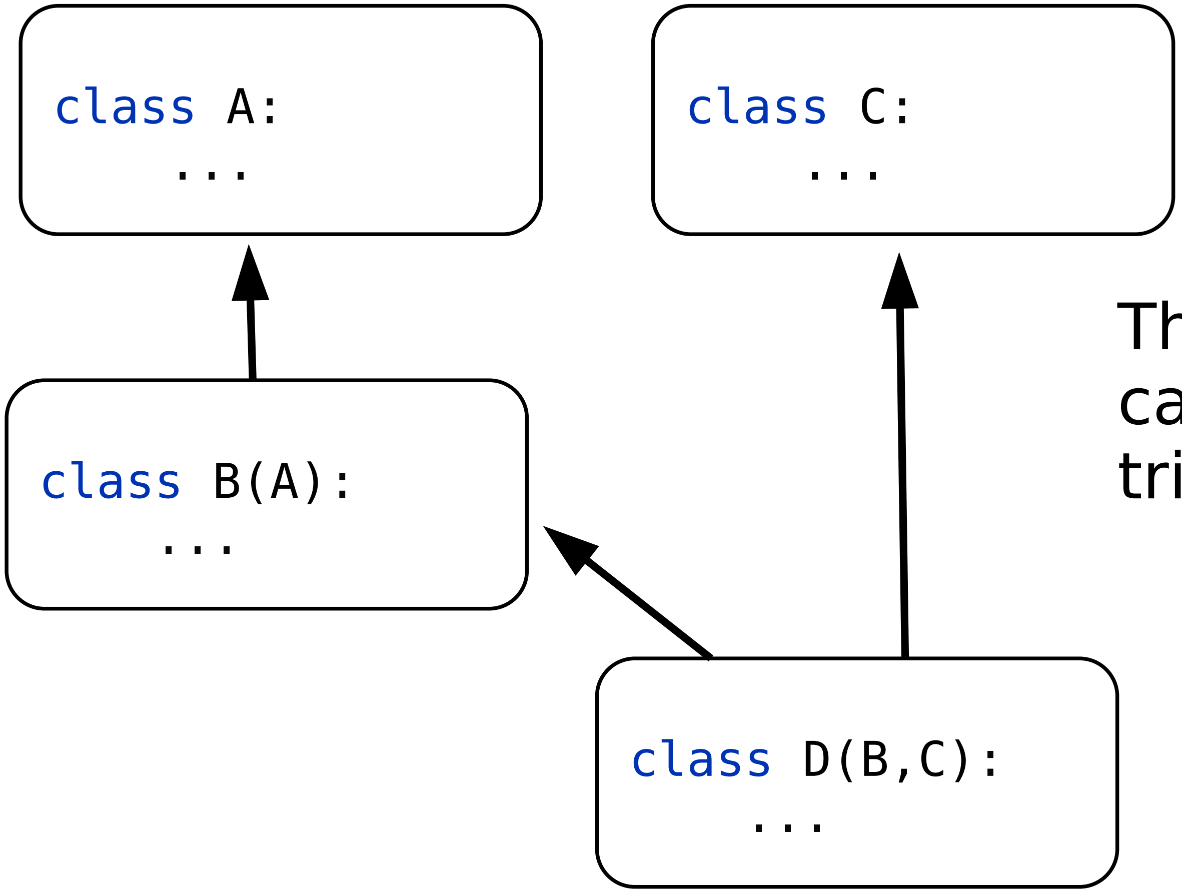
- Inheritance exposes all the features of the parent
- Object composition limits what is exposed – more control at the cost of more work
- Composition is a "has a" relationship
- The composed object is exposed by explicit "delegation"

```
class Holding:
    def __init__(self, name, shares, price):
        self._stock = Stock(name, shares, price)
    def cost(self):
        return self._stock.cost()
    @property
    def name(self):
        return self._stock.name
```

Note: If we used "dependency injection" as well as composition the Stock would be created outside the Holding and passed into the initialiser.

Multiple Inheritance

- Classes can have multiple parents



The practical details can be somewhat tricky!

Cooperative Inheritance

- Python has "cooperative multiple inheritance"
- Child classes can arrange their parents to cooperate
- So the order of the parents is significant
- Attribute lookup can jump from parent to parent

```python
class Child(Parent1, Parent2, Parent3):
    ...
```

Cooperative Inheritance

```
class Parent:
    def method(self):
        print('Parent')
```

```
class A(Parent):
    def method(self):
        print('A')
        super().method()
```

```
class B(Parent):
    def method(self):
        print('B')
        super().method()
```

- See what happens when we use it:

```
class C(A, B):
    pass
```

```
>>> c = C()
>>> c.method()
A
B
Parent
```

It's gone sideways!

Cooperative Inheritance

- Using cooperative inheritance we can create and combine classes as components to build things

An Interesting Code Reuse

```python
class Dog:
    def noise(self):
        return 'Woof'
    def chase(self):
        return 'Chasing!'

class LoudDog(Dog):
    def noise(self):
        return super()\
         .noise().upper()
```

```python
class Bike:
    def noise(self):
        return 'Ding!'
    def pedal(self):
        return 'Pedaling!'

class LoudBike(Bike):
    def noise(self):
        return super()\
         .noise().upper()
```

- Two unrelated pieces of code
- Yet there is code duplicated between them
- But nowhere obvious to put the common code

Mixin Classes

- Mixin classes add new behavior to existing classes
- We "mix-in" a slice of functionality, like aspect oriented programming
- This is a great tool for reducing code duplication
- Individual bits of functionality can be provided as reusable components in the form of mixin classes

```
# An example from Flask
from flask_login import UserMixin

class User(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True)
    ...
```

Mixin Classes

- To add a loud() method to unrelated classes like Dog and Bike we can write a mixin class that looks like this:

```
class Loud:
    def noise(self):
        return super().noise().upper()
```

- Note that it can't be used by itself!
- We use it by mixing it into other classes
- The mixin class always goes first

```
class LoudDog(Loud, Dog):
    pass

class LoudBike(Loud, Bike):
    pass
```

How it Works

- Example:

```
class LoudDog(Loud, Dog):
    pass
```

super()

```
>>> dog = LoudDog()
>>> dog.noise()
'WOOF'
```

- super moves from one class to the next
- This allows us to use our mixin class with *any* compatible class!
- Details to be explained shortly

Inside Python Objects

- A dictionary with string keys can be seen as a collection of "named" objects

```
stock = {
    "name": 'GOOG',
    "shares": 100,
    "price": 490.10
}
```

- Dictionaries are one of the basic data structures
- They are used internally everywhere in Python
- "Names" (dictionary keys) mapping to objects provides a "namespace", a fundamental idea in Python

Dicts and Objects

- Dictionaries are used inside classes we create for:
  - Instance data
  - Class members
- In fact the entire object system is mostly just an extra layer that's put on top of dictionaries
- Let's take a look...

Dicts and Instances

- A dictionary holds the instance data (__dict__)

```
>>> stock = Stock('GOOG', 100, 490.10)
>>> stock.__dict__
{'name': 'GOOG', 'shares': 100, 'price': 490.1}
```

- When we assign to self we populate this dictionary

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

self.__dict__
```

```
{
    "name": 'GOOG',
    "shares": 100,
    "price": 490.10
}
```

Dicts and Instances

- Because every instance has its own dictionary they can each store separate data

```
s = Stock('GOOG',100,490.10)
t = Stock('AAPL',50,123.45)
```

```
{
    "name": 'GOOG',
    "shares": 100,
    "price": 490.10
}
```

```
{
    "name": 'AAPL',
    "shares": 50,
    "price": 123.45
}
```

Dicts and Classes

- The class members are also stored in a dictionary

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

    def cost(self):
        return self.shares * self.price

    def sell(self, nshares):
        self.shares -= nshares
```

methods

```
Stock.__dict__   ──────────▶   {
                                   "__init__": <function>,
                                   "cost": <function>,
                                   "sell": <function>
                               }
```

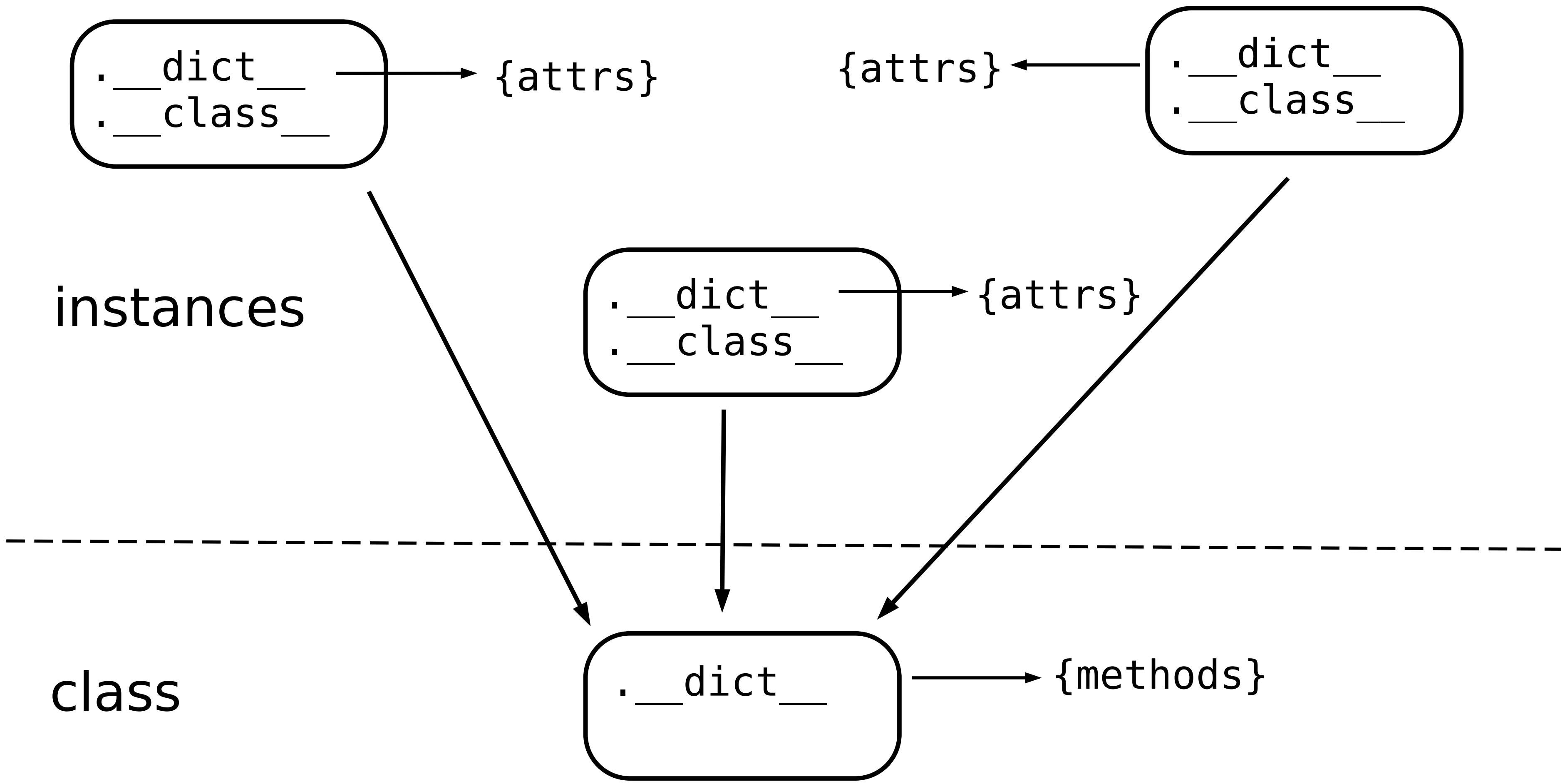
Instances and Classes

- Instances and classes are linked together
- The __class__ attribute links instances back to the class

```
>>> stock = Stock('GOOG', 100, 490.10)
>>> stock.__dict__
{'name': 'GOOG', 'shares': 100, 'price': 490.1}
>>> stock.__class__
<class '__main__.Stock'>
```

- The instance dictionary holds the data which is separate for each instance whereas the class dictionary holds the data which is shared by all the instances

Instances and Classes



Attribute Access

- When we work with objects we access their methods and data using the dot operator (.), the attribute lookup operator

```
x = obj.name            # Getting
obj.name = value    # Setting
del obj.name            # Deleting
```

- Attribute operations correspond directly to operations on the underlying dictionary
- This can potentially be customised by the attribute lookup protocol

Modifying Instances

- Operations that modify an object update the underlying dictionary

```
>>> stock = Stock('GOOG', 100, 490.10)
>>> stock.__dict__
{'name': 'GOOG', 'shares': 100, 'price': 490.1}
>>> stock.shares = 50
>>> stock.date = '2023-03-23'
>>> stock.__dict__
{'name': 'GOOG', 'shares': 50, 'price': 490.1, 'date': '2023-03-23'}
>>> del stock.shares
>>> stock.__dict__
{'name': 'GOOG', 'price': 490.1, 'date': '2023-03-23'}
```

Fetching Attributes

- Suppose you want to fetch an attribute from the object

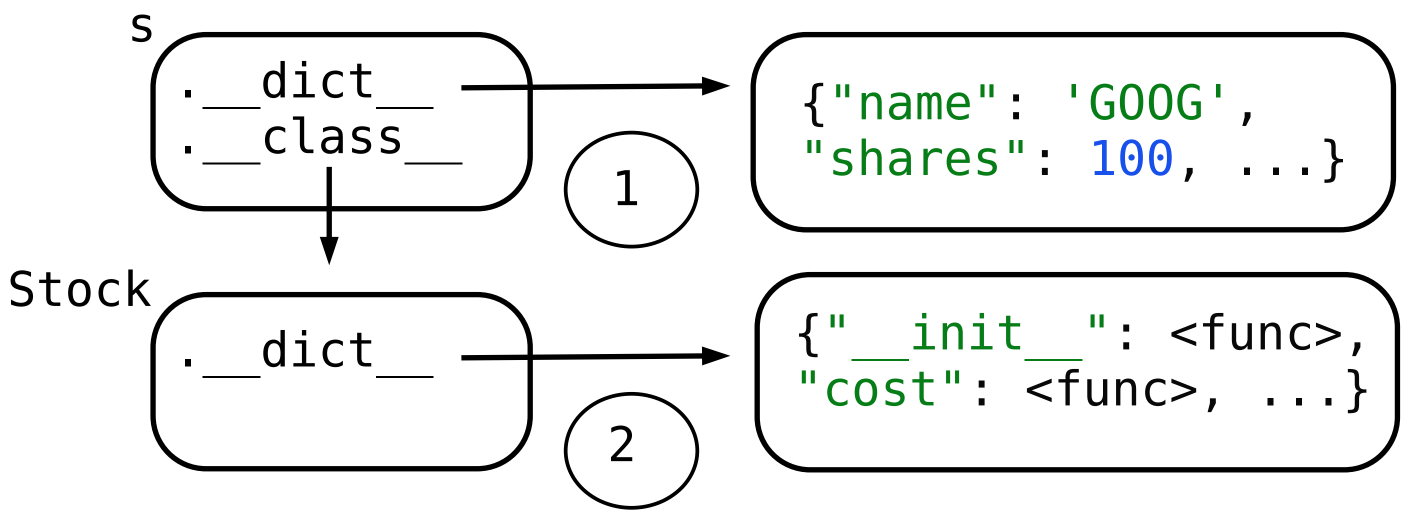
```
x = obj.name
```

- The attribute may exist in two places
  - The local instance attribute dictionary
  - The class member dictionary
- So both dictionaries have to be checked

Fetching Attributes

- First check in __dict__ of the instance
- If the attribute isn't found look in __dict__ of the class

```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
```



- This lookup scheme is how class members are shared by all instances of the class

How Inheritance Works

- Classes may inherit from other classes

```
class A(B, C):
    ...
```

- The bases of the class are stored as a tuple on the class

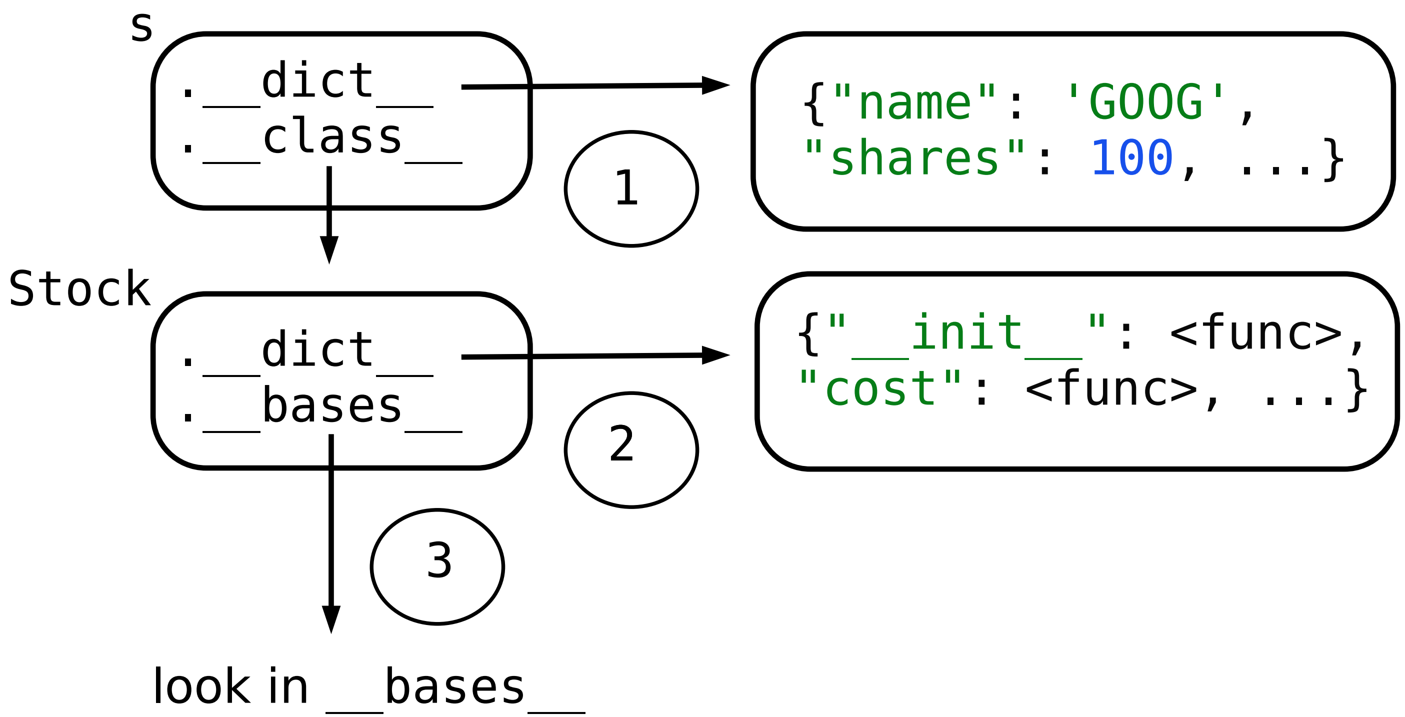
```
>>> C.__bases__
(<class '__main__.A'>, <class '__main__.B'>)
```

- This provides a link from the class to the parents
- This link is used to extend the search for attribute lookups

Fetching Attributes

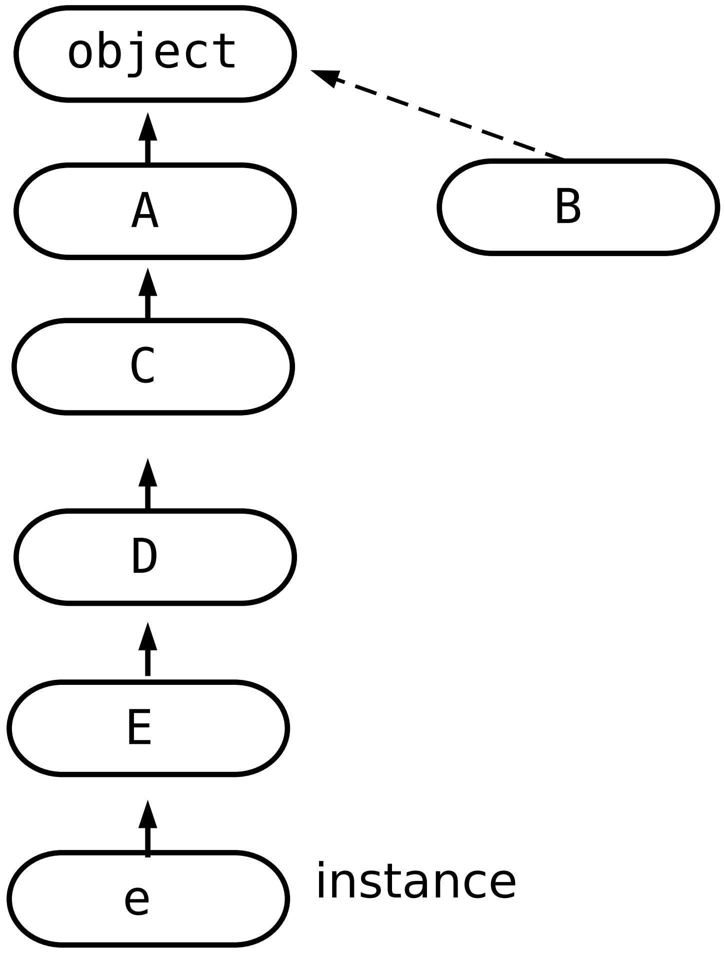
- First check in __dict__ of the instance
- If the attribute isn't found look in __dict__ of the class

```
>>> s = Stock(...)
>>> s.name
'GOOG'
>>> s.cost()
49010.0
```



Single Inheritance

- Where we have an inheritance hierarchy with several classes, attribute lookup walks up the inheritance tree



- With single inheritance, there is a single path to the top
- Python stops when it finds the first match

MRO

- The inheritance chain is computed when the class is created and stored as an "MRO" attribute on the class

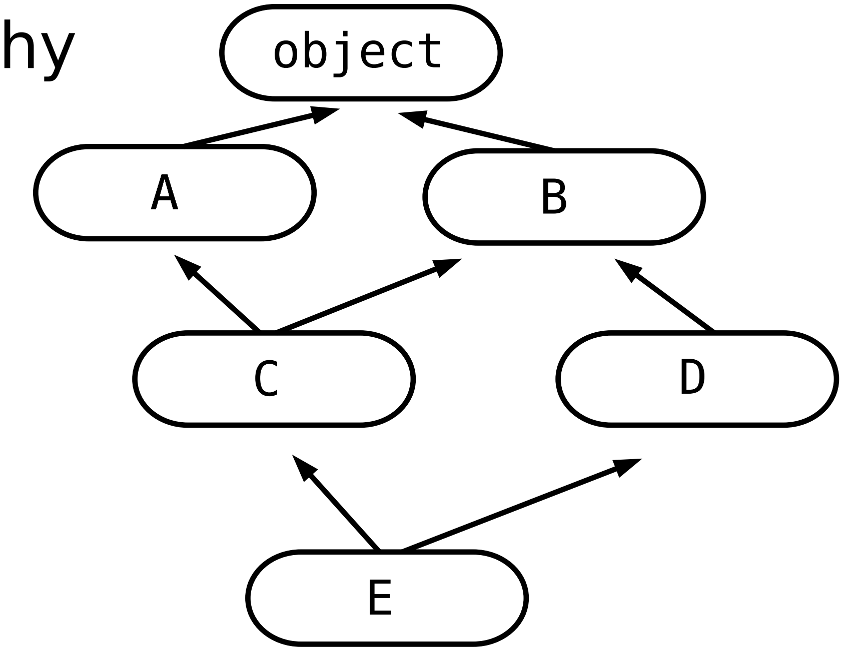
```
>>> E.__mro__
(<class '__main__.E'>, <class '__main__.D'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

- The "method resolution order"
- To find an attribute Python walks the MRO
- First match wins

Multiple Inheritance

- Consider this inheritance hierarchy

```
class A: pass
class B: pass
class C(A, B): pass
class D(B): pass
class E(C, D): pass
```



- What happens here?

```
e = E()
e.attr
```

- A similar search process is carried out, but is more complex as there are several potential search paths

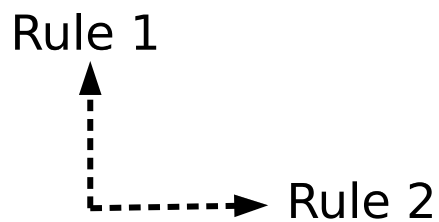
Multiple Inheritance

- Python uses "cooperative multiple inheritance"
- There are two rules to decide the order of the MRO:

<u>Rule 1</u>: Children go before parents

<u>Rule 2</u>: Parents go in order

- Inheritance goes in two directions (up the hierarchy and across the order of the parents)




```python
class C(A, B):
    ...
```

Multiple Inheritance

- The multiple inheritance hierarchy is flattened, linearised

```
>>> E.__mro__
(<class '__main__.E'>, <class '__main__.C'>, <class
'__main__.A'>, <class '__main__.D'>, <class '__main__.B'>,
<class 'object'>)
```

- The order of the MRO is calculated using the C3 linearization algorithm
- A constrained merge sort of parents
- It produces an ordering based on "the rules"

Multiple Inheritance

- Consider classes with a common parent

```python
class Base:
    ...

class A(Base):
    ...

class B(Base):
    ...

class C(Base):
    ...
```

- All children of a common base go first

```python
class D(A, B, C):
    ...
```

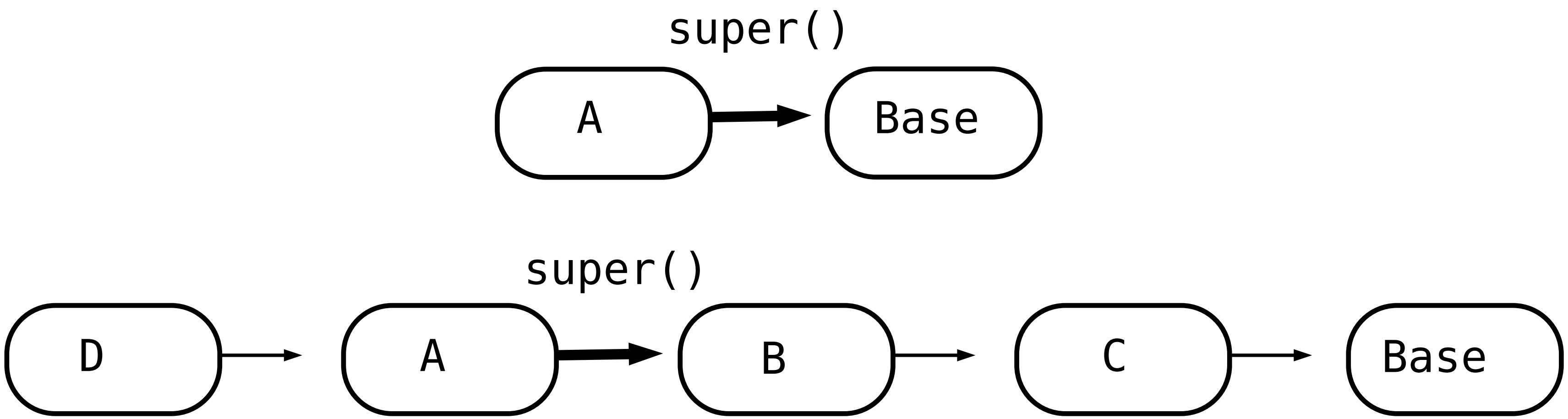
MRO: D → A → B → C → Base

Why super()?

- Always use super when overriding methods

```python
class A(Base):
    def method(self):
        ...
        return super().method()
```

- super() jumps to the *next* class in the MRO
- The next class isn't always directly a parent



super Explained

- super() is one of Python's most poorly understood features

```python
class A(Base):
    def method(self):
        ...
        return Base.method()
```

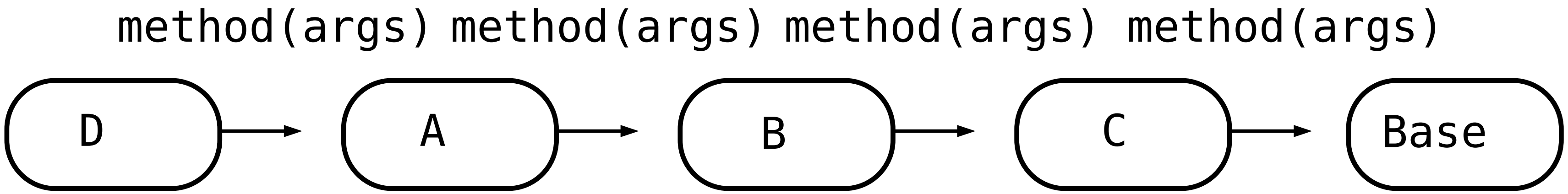
vs.

```python
class A(Base):
    def method(self):
        ...
        return super().method()
```

- The classes above are not equivalent
- super() jumps to the *next* class is in the MRO, which is not always the direct parent
- So hardcoding the base will break if anyone uses that class with multiple inheritance

Designing for Inheritance

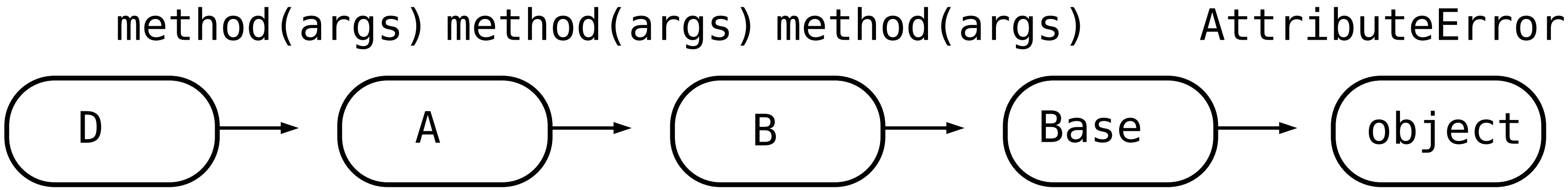
- Rule 1: Compatible method signatures



- Overriden methods must have a compatible signature across the whole hierarchy
- Remember: super() might not go an immediate parent
- Tip: If there are varying signatures you can use keyword arguments (but must avoid passing them on)

Designing for Inheritance

- Rule 2: method chains must terminate



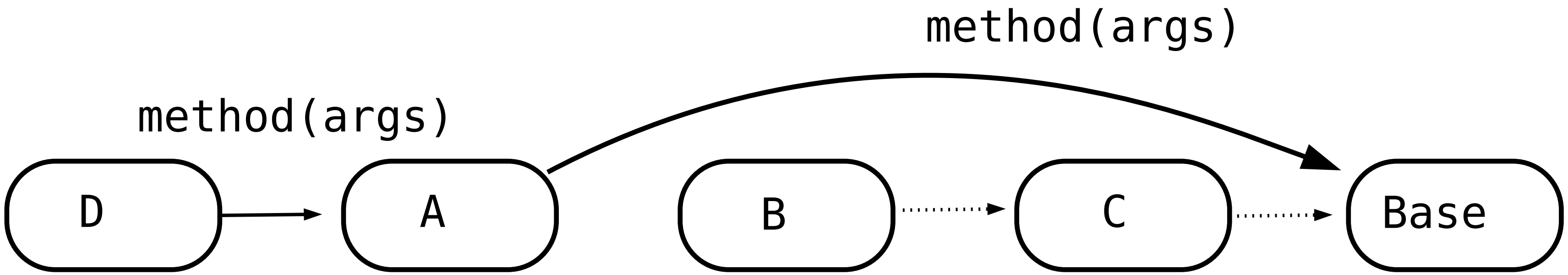
- If every class calls super at some point it will reach object and fail – some class has to terminate the chain

```
class Base:
    def method(self):
        pass
```

- Typically the role of a common base class

Designing for Inheritance

- Rule 3: use super() everywhere



- Direct parent calls cause unexpected and broken behaviour with multiple inheritance

```
class A(Base):
    def method(self):
        ...
        return Base.method(self)   # NO!
```

Operator Overloading

- Classes can customise almost every aspect of their behaviour
- This is done through special methods, the protocol methods

```
class Point:
    def __init__(self):
        ...
    def __str__(self):
        ...
```

- There are dozens of protocols and hundreds of methods
- We'll look at some of the essential protocol methods

The String Conversion Protocol

- Objects have two string representations

```
>>> from datetime import date
>>> d = date(2023, 3, 23)
>>> print(d)
2023-03-23
>>> d
datetime.date(2023, 3, 23)
```

- str(x) – Printable output (for humans)

```
>>> str(d)
'2023-03-23'
```

- repr(x) – For programmers

```
>>> repr(d)
'datetime.date(2023, 3, 23)'
```

String Conversions

```python
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __str__(self):
        return '{}-{}-{}'.format(self.year,
            self.month, self.day)

    def __repr__(self):
        return 'Date({}, {}, {})'. format(self.year,
            self.month, self.day)
```

Note: the convention is for __repr__() to return a string that if fed back to eval() recreates the original object. It's not always possible.

'`{self.__class__.__name__}(...)`' can be used to avoid hardcoding class names in the repr.

The Container Protocol

- Methods for implementing containers

```
len(x)          x.__len__()
x[a]            x.__getitem__(a)
x[a] = v        x.__setitem__(a)
del x[a]        x.__delitem__(a)
a in x          x.__contains(a)
```

- Definitions in a class

```python
class Container:
    def __len__(self):
        ...
    def __getitem__(self, a):
        ...
    def __setitem__(self, a, v):
        ...
    def __delitem__(self, a):
        ...
    def __contains__(self, a):
        ...
```

The Numeric Protocol

- Mathematical operations (see reference for the gory details including right hand variants)

`a + b`	`a.__add__(b)`	Addition
`a - b`	`a.__sub__(b)`	Subtraction
`a * b`	`a.__mul__(b)`	Multiplication
`a / b`	`a.__div__(b)`	Division
`a // b`	`a.__floordiv__(b)`	Floor division
`a % b`	`a.__mod__(b)`	Modulo
`a << b`	`a.__lshift__(b)`	Left shift
`a >> b`	`a.__rshift__(b)`	Right shift
`a & b`	`a.__and__(b)`	Bitwise and
`a | b`	`a.__or__(b)`	Bitwise or
`a ^ b`	`a.__xor__(b)`	Bitwise xor
`a ** b`	`a.__pow__(b)`	Power
`-a`	`a.__neg__()`	Unary negative
`~a`	`a.__invert__()`	Bitwise not
`abs(a)`	`a.__abs__()`	Absolute value
`a @ b`	`a.__matmul__(b)`	Matrix multiplication