Inheritance and Subtype Polymorphism

Austin Bingham
@austin_bingham
austin@sixty-north.com

Presenter

Robert Smallshire
@robsmallshire
rob@sixty-north.com

pluralsight
hardcore dev and IT training

single inheritance

```
class SubClass(BaseClass)
```

- Subclasses will want to initialize base classes

- Base class initializer will only be called automatically if subclass initializer is undefined

Calling base class initializer

- Other languages automatically call base class initializers
- Python treats `__init__()` like any other method
- Base class `__init__()` is not called if overridden

Calling base class initializer

- Other languages automatically call base class initializers
- Python treats `__init__()` like any other method
- Base class `__init__()` is not called if overridden
- Use `super()` to call base class `__init__()`

Multiple inheritance in Python is not much more complex than single inheritance.

`isinstance()`

determines if an object is of a specified type

issubclass()

determines if one type is a subclass of another

multiple inheritance

defining a class with more than one base class

```
class SubClass(Base1, Base2, . . .)
```

Multiple inheritance

- Subclasses inherit methods of all bases
- Without conflict, names resolve in the obvious way
- Method Resolution Order (MRO) determines name lookup in all cases

If a class

A. has **multiple** base classes

B. defines **no initializer**

then **only** the initializer of the **first** base class is automatically called

__bases__

a tuple of base classes

method resolution order

ordering that determines method name lookup

- Commonly called “MRO”
- Methods may be defined in multiple places
- MRO is an ordering of the inheritance graph

method resolution order

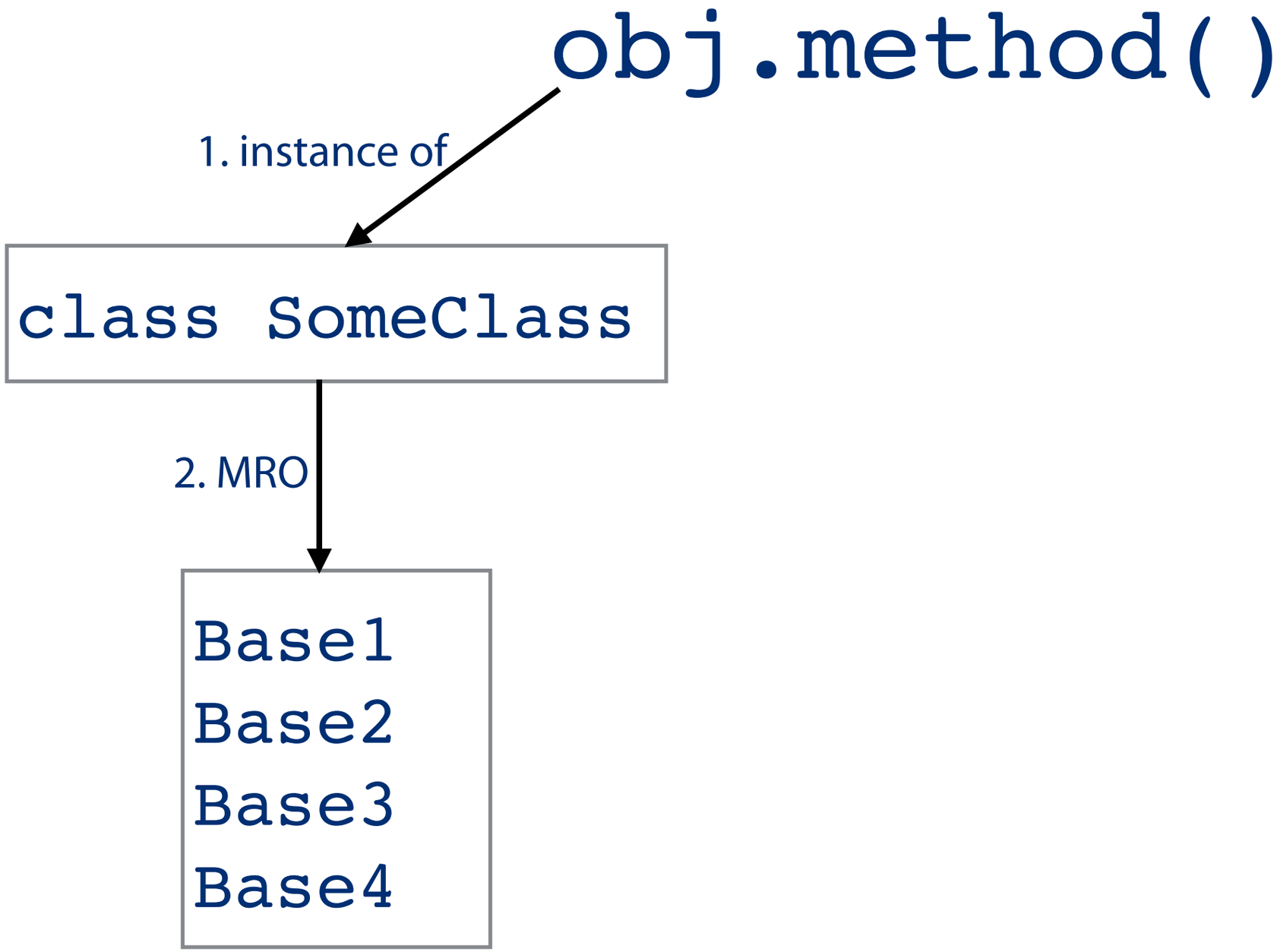
ordering that determines method name lookup

- Commonly called “MRO”
- Methods may be defined in multiple places
- MRO is an ordering of the inheritance graph
- Actually quite simple

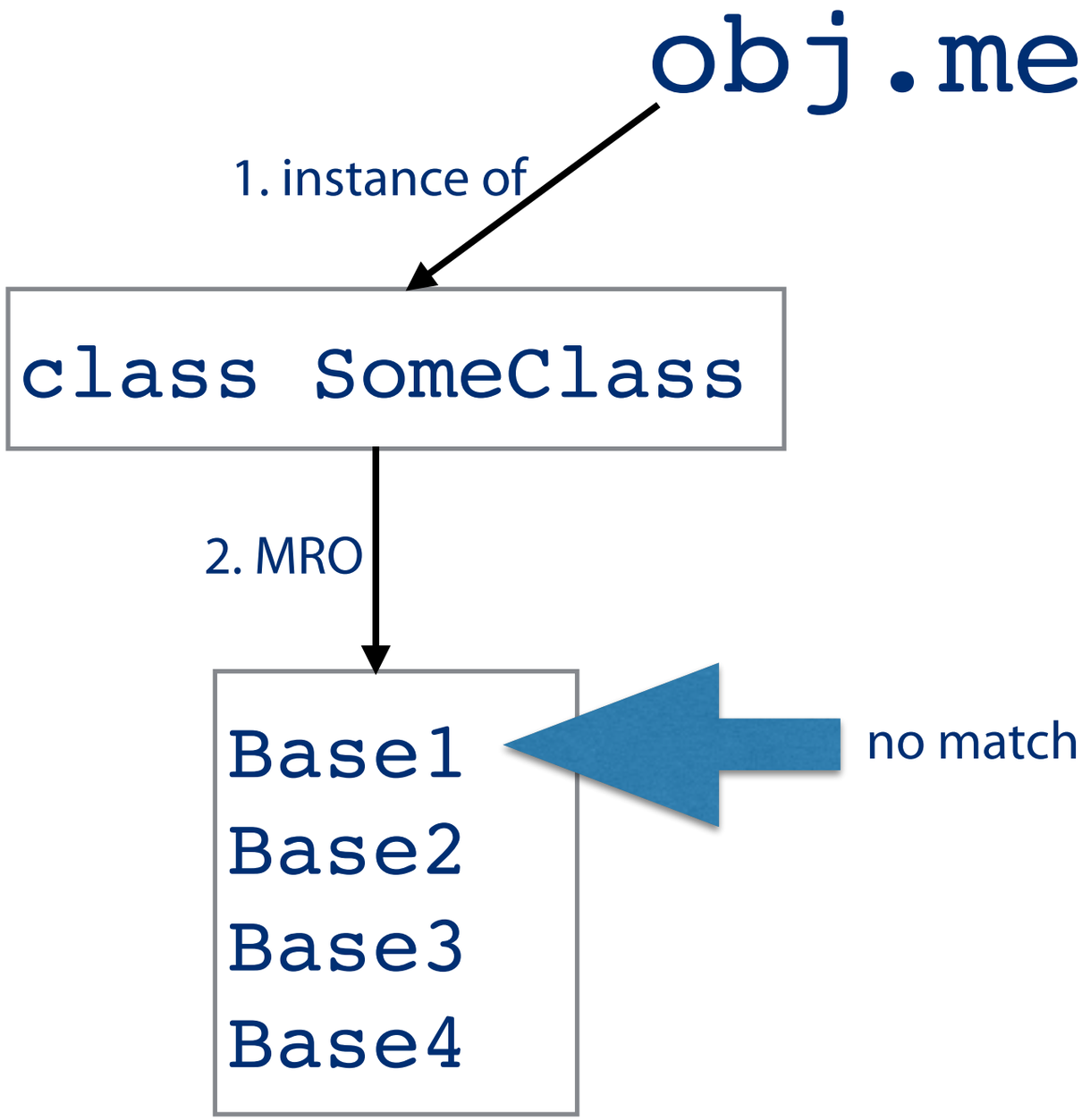
How is MRO used?

```
obj.method()
```

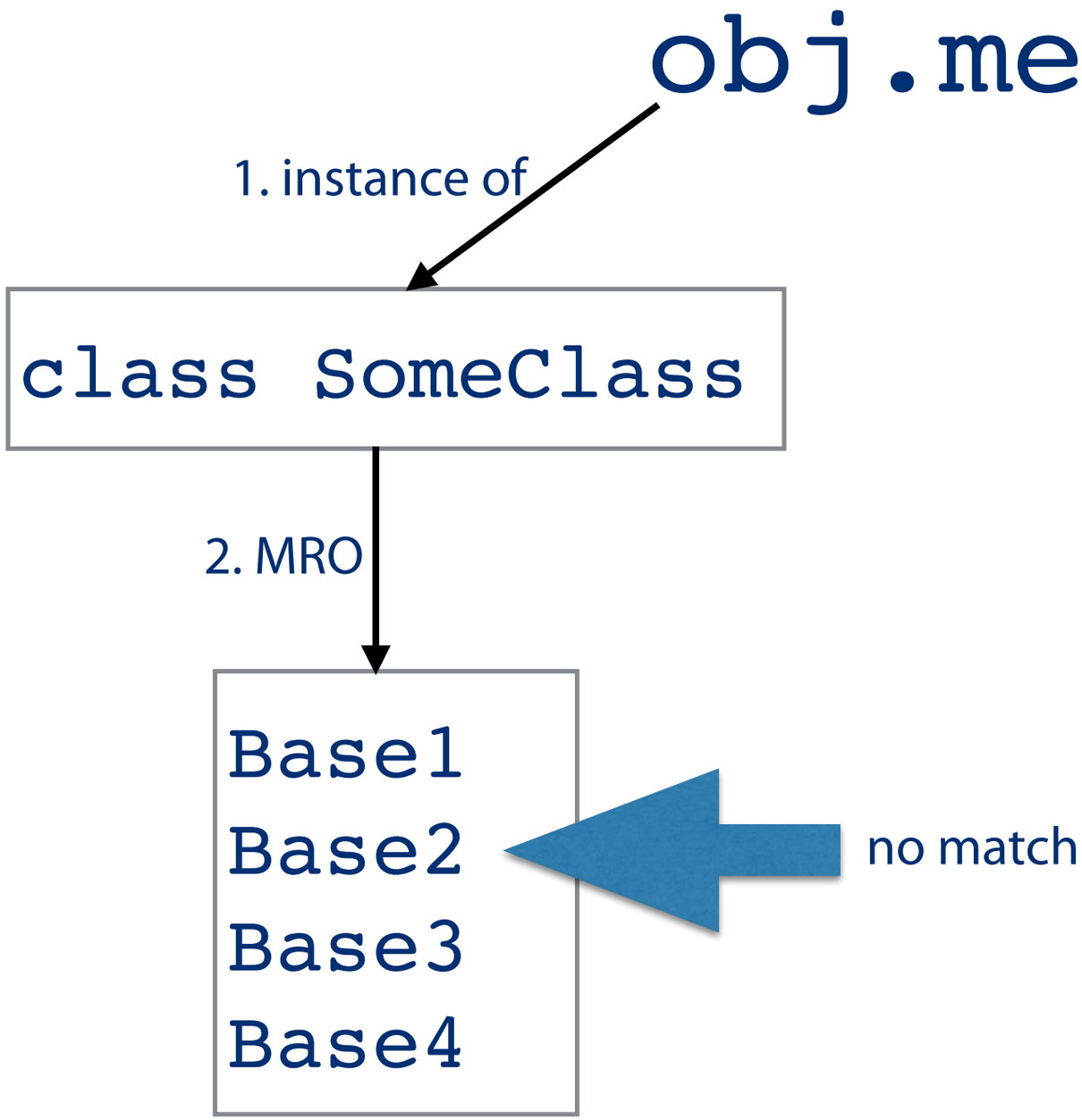
How is MRO used?



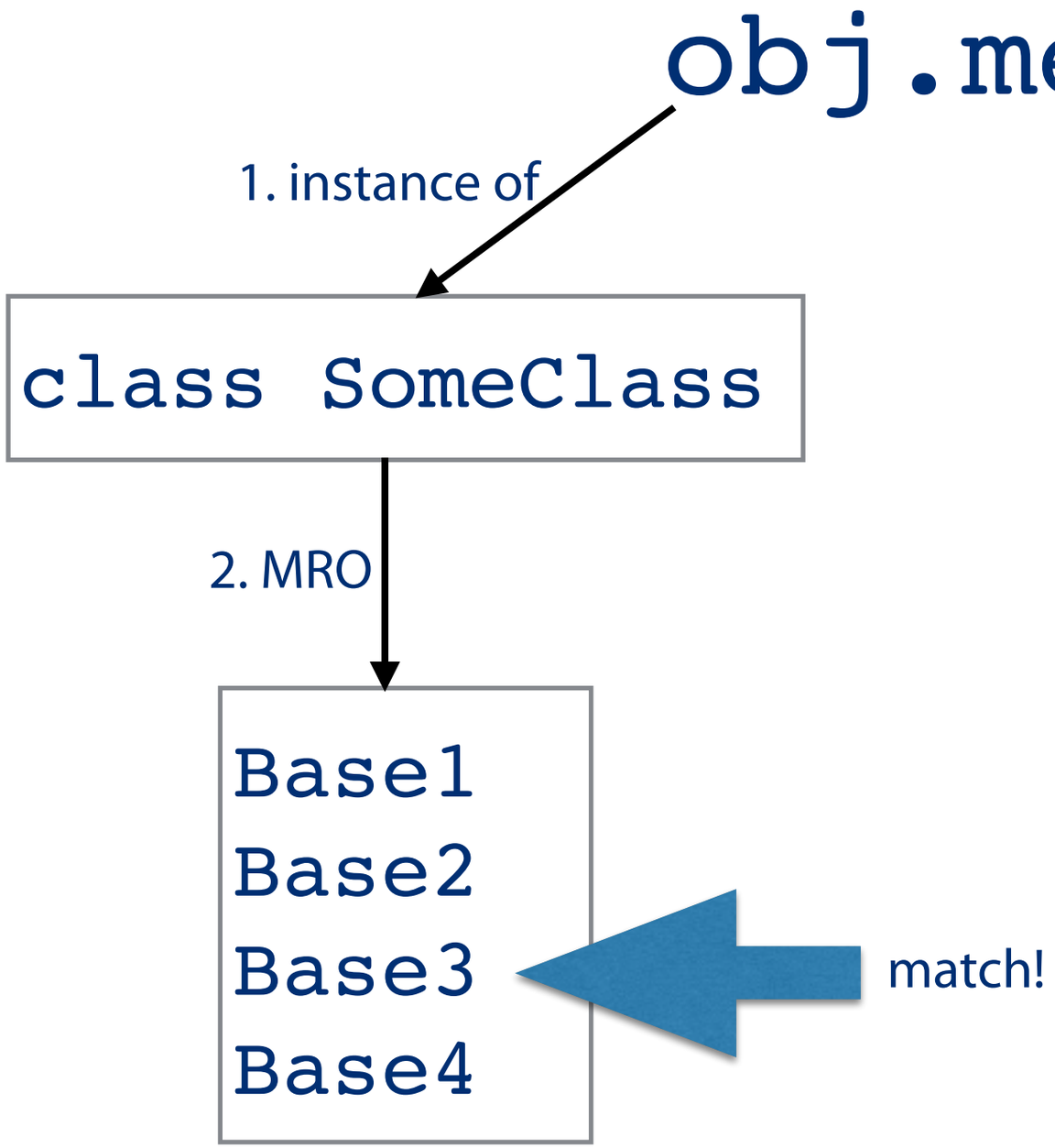
How is MRO used?



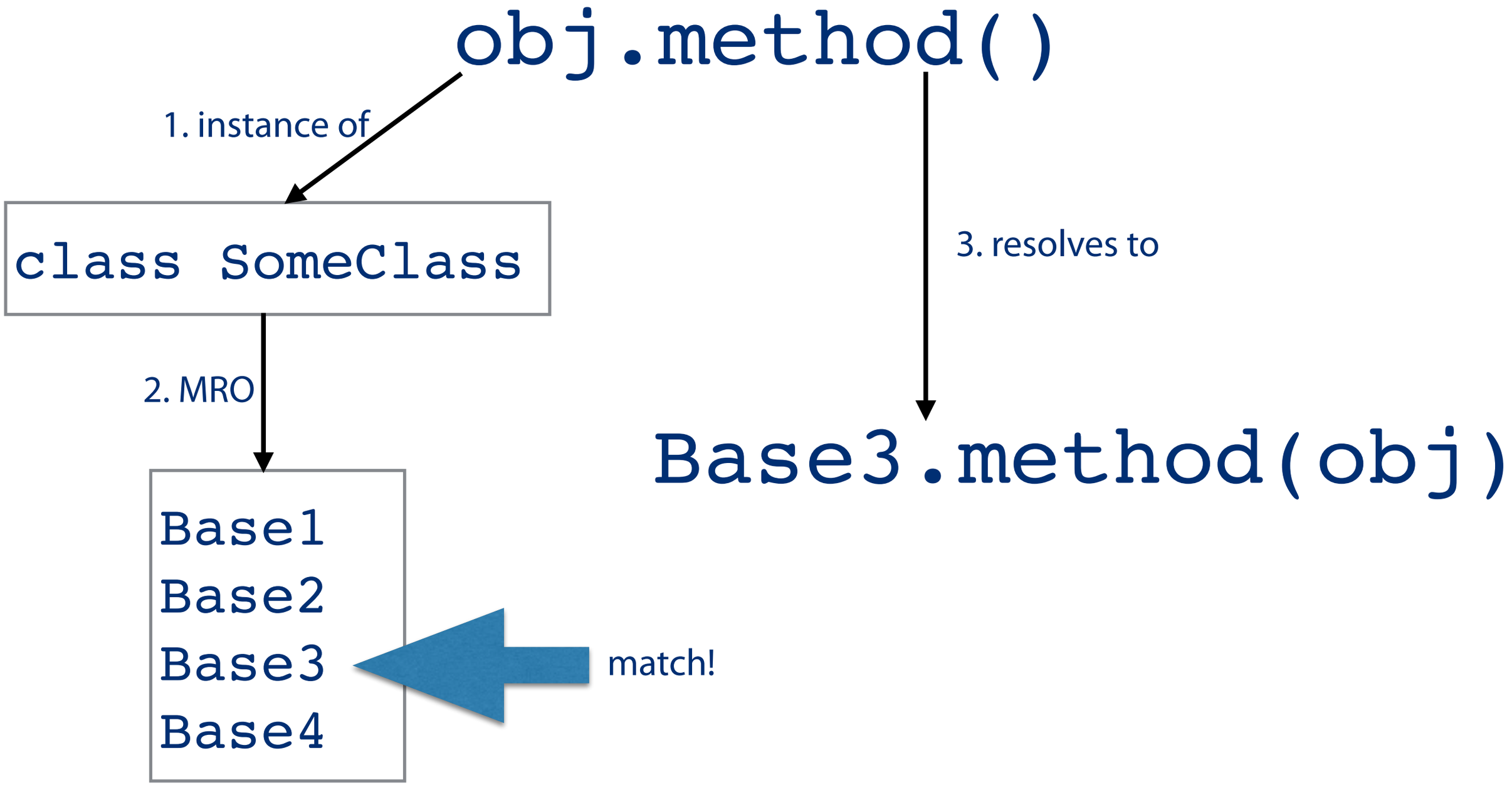
How is MRO used?



How is MRO used?



How is MRO used?



python™

Not all inheritance declarations are **allowed!**

C3

algorithm for calculating MRO in Python

- Subclasses come before base classes
- Base class order from class definition is preserved
- First two qualities are preserved no matter where you start in the inheritance graph

super()

Given a ***method resolution order***
and a class C, `super()` gives you an object which resolves methods using only the part of the ***MRO*** which comes after C.

`super()` returns a **proxy** object which **routes** method calls.

Bound proxy

bound to a specific class or instance

Unbound proxy

not bound to a class or instance

`super()` returns a **proxy** object which **routes** method calls.

Bound proxy

bound to a specific class or instance

Unbound p

not bound to a clas

We'll only discuss **bound proxies** in this course

There are two types of bound proxies:

instance-bound

and

class-bound

Class-bound proxy

subclass of first argument

```
super(base-class, derived-class)
```

class object

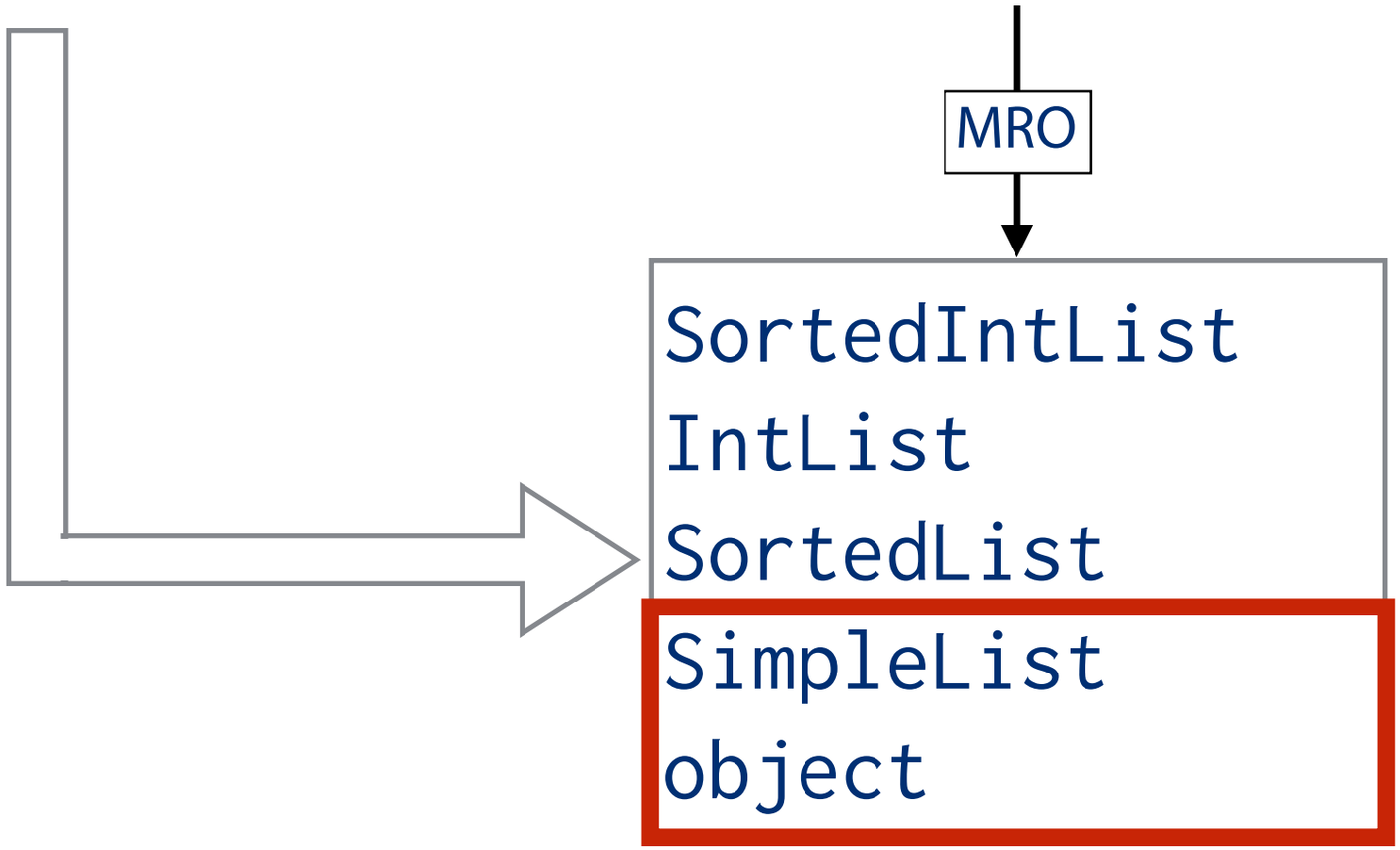
Class-bound proxy

`super(base-class, derived-class)`

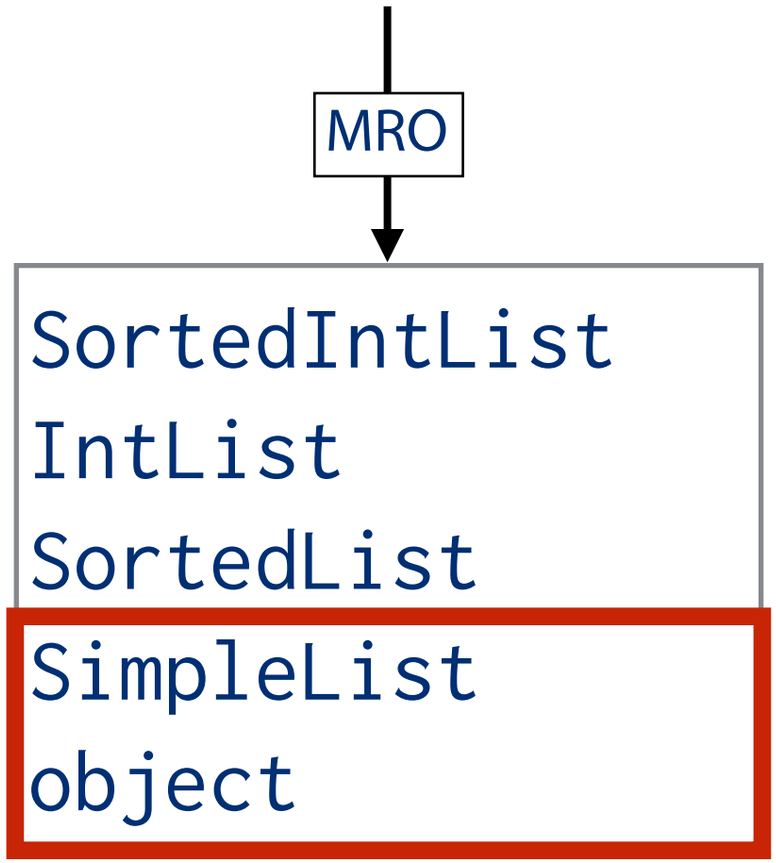
- Python finds MRO for `derived-class`
- It then finds `base-class` in that MRO
- It take everything ***after*** `base-class` in the MRO, and finds the first class in that sequence with a matching method name

python™

```
super(SortedList, SortedIntList)
```

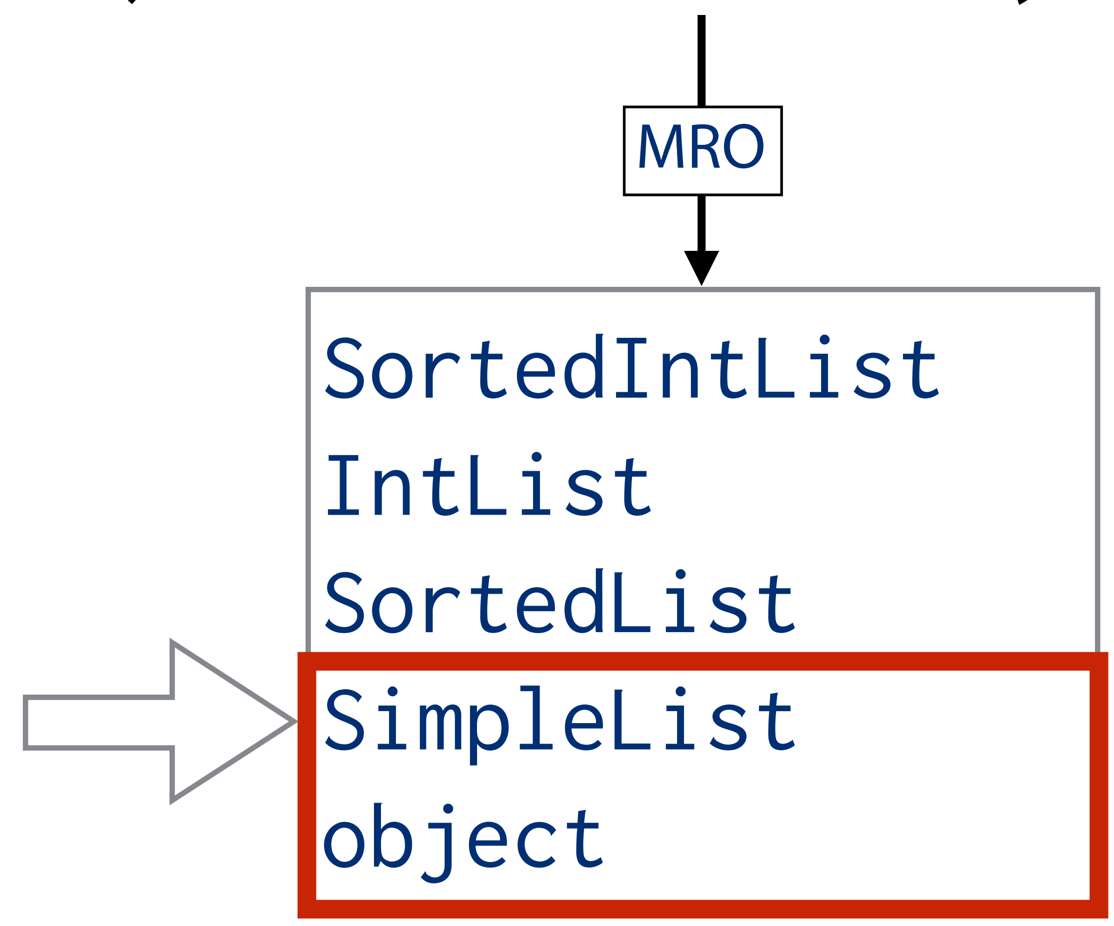


```
super(SortedList, SortedIntList)
```



python™

```
super(SortedList, SortedIntList)
```



Instance-bound proxy

instance of first argument

```
super(class, instance-of-class)
```

class object

Instance-bound proxy

`super(class, instance-of-class)`

- Finds the MRO for the type of the second argument
- Finds the location of the first argument in the MRO
- Uses everything ***after*** that for resolving methods

super() arguments

```
super(base-class, derived-class)
```

```
super(class, instance-of-class)
```

super() arguments

```
super(class-of-method, self)
```

```
super(class-of-method, class)
```

super() arguments

super()

instance method

```
super(class-of-method, self)
```

class method

```
super(class-of-method, class)
```

`super()` **uses everything after a specific class in an MRO to resolve method calls**

So **how** does `SortedIntList` work?

`super()` **uses everything after a specific class in an MRO to resolve method calls**

So **how** does `SortedIntList` work?

Both classes use `super()` instead of **direct** base classes references

sorted_list.py

```
class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

    def __getitem__(self, index):
        return self._items[index]

    def sort(self):
        self._items.sort()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return "SimpleList({!r})".format(self._items)

class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, item):
        super().add(item)
        self.sort()

    def __repr__(self):
        return "SortedList({!r})".format(list(self))

class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items: self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
        super().add(item)

    def __repr__(self):
        return "IntList({!r})".format(list(self))

class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))
```

```
>>> SortedIntList.mro()
[<class 'sorted_list.SortedIntList'>,
 <class 'sorted_list.IntList'>,
 <class 'sorted_list.SortedList'>,
 <class 'sorted_list.SimpleList'>,
 <class 'object'>]

>>> sil = SortedIntList()
>>> sil.add(6)
```

sorted_list.py

```
class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

    def __getitem__(self, index):
        return self._items[index]

    def sort(self):
        self._items.sort()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return "SimpleList({!r})".format(self._items)

class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, item):
        super().add(item)
        self.sort()

    def __repr__(self):
        return "SortedList({!r})".format(list(self))

class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items: self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
        super().add(item)

    def __repr__(self):
        return "IntList({!r})".format(list(self))

class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))
```

```
>>> SortedIntList.mro()
[<class 'sorted_list.SortedIntList'>,
 <class 'sorted_list.IntList'>,
 <class 'sorted_list.SortedList'>,
 <class 'sorted_list.SimpleList'>,
 <class 'object'>]

>>> sil = SortedIntList()
>>> sil.add(6)
```

sorted_list.py

```
class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

    def __getitem__(self, index):
        return self._items[index]

    def sort(self):
        self._items.sort()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return "SimpleList({!r})".format(self._items)

class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, item):
        super().add(item)
        self.sort()

    def __repr__(self):
        return "SortedList({!r})".format(list(self))

class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items: self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
        super().add(item)

    def __repr__(self):
        return "IntList({!r})".format(list(self))

class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))
```

```
>>> SortedIntList.mro()
[<class 'sorted_list.SortedIntList'>,
 <class 'sorted_list.IntList'>,
 <class 'sorted_list.SortedList'>,
 <class 'sorted_list.SimpleList'>,
 <class 'object'>]

>>> sil = SortedIntList()
>>> sil.add(6)
```

sorted_list.py

```
class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

    def __getitem__(self, index):
        return self._items[index]

    def sort(self):
        self._items.sort()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return "SimpleList({!r})".format(self._items)

class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, item):
        super().add(item)
        self.sort()

    def __repr__(self):
        return "SortedList({!r})".format(list(self))

class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items: self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
        super().add(item)

    def __repr__(self):
        return "IntList({!r})".format(list(self))

class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))
```

```
>>> SortedIntList.mro()
[<class 'sorted_list.SortedIntList'>,
 <class 'sorted_list.IntList'>,
 <class 'sorted_list.SortedList'>,
 <class 'sorted_list.SimpleList'>,
 <class 'object'>]

>>> sil = SortedIntList()
>>> sil.add(6)
```

sorted_list.py

```
class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

    def __getitem__(self, index):
        return self._items[index]

    def sort(self):
        self._items.sort()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return "SimpleList({!r})".format(self._items)

class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, item):
        super().add(item)
        self.sort()

    def __repr__(self):
        return "SortedList({!r})".format(list(self))

class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items: self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
        super().add(item)

    def __repr__(self):
        return "IntList({!r})".format(list(self))

class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))
```

```
>>> SortedIntList.mro()
[<class 'sorted_list.SortedIntList'>,
 <class 'sorted_list.IntList'>,
 <class 'sorted_list.SortedList'>,
 <class 'sorted_list.SimpleList'>,
 <class 'object'>]

>>> sil = SortedIntList()
>>> sil.add(6)
```

sorted_list.py

```
class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

    def __getitem__(self, index):
        return self._items[index]

    def sort(self):
        self._items.sort()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return "SimpleList({!r})".format(self._items)

class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, item):
        super().add(item)
        self.sort()

    def __repr__(self):
        return "SortedList({!r})".format(list(self))

class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items: self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
        super().add(item)

    def __repr__(self):
        return "IntList({!r})".format(list(self))

class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))
```

```
>>> SortedIntList.mro()
[<class 'sorted_list.SortedIntList'>,
 <class 'sorted_list.IntList'>,
 <class 'sorted_list.SortedList'>,
 <class 'sorted_list.SimpleList'>,
 <class 'object'>]
```

```
>>> sil = SortedIntList()
>>> sil.add(6)
```

`object`

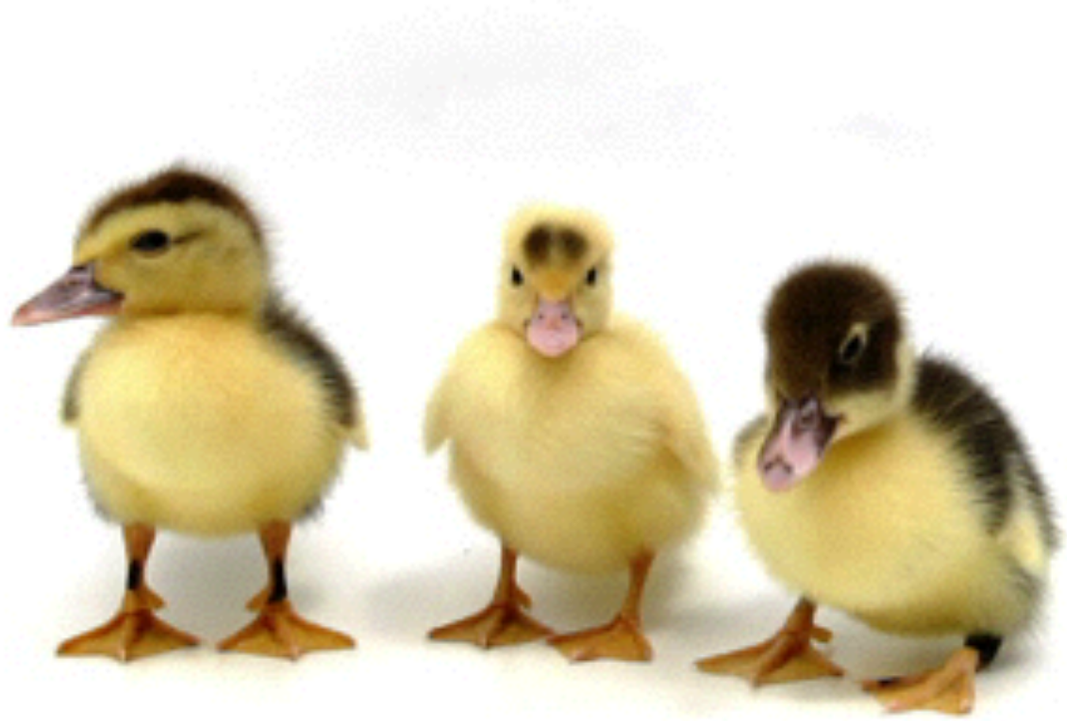
the core of the Python object model

`object` is the ultimate base class of every class

`object` is automatically added as a base class

Duck Tails

Inheritance and Implementation Sharing



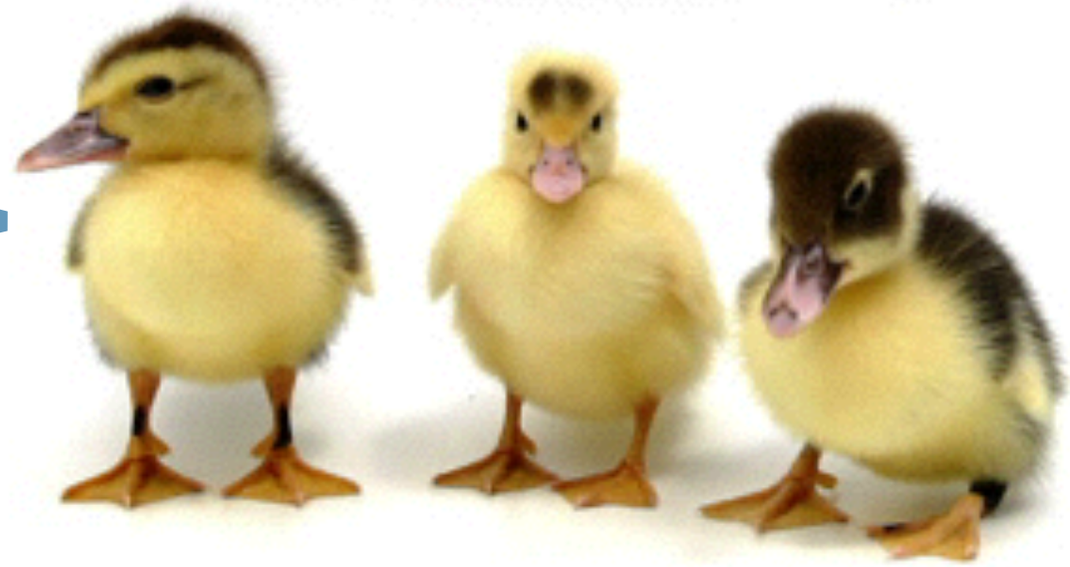
Duck Tails

Unlike nominal typing

Type Management is tiring!

No function argument types

Python uses duck typing

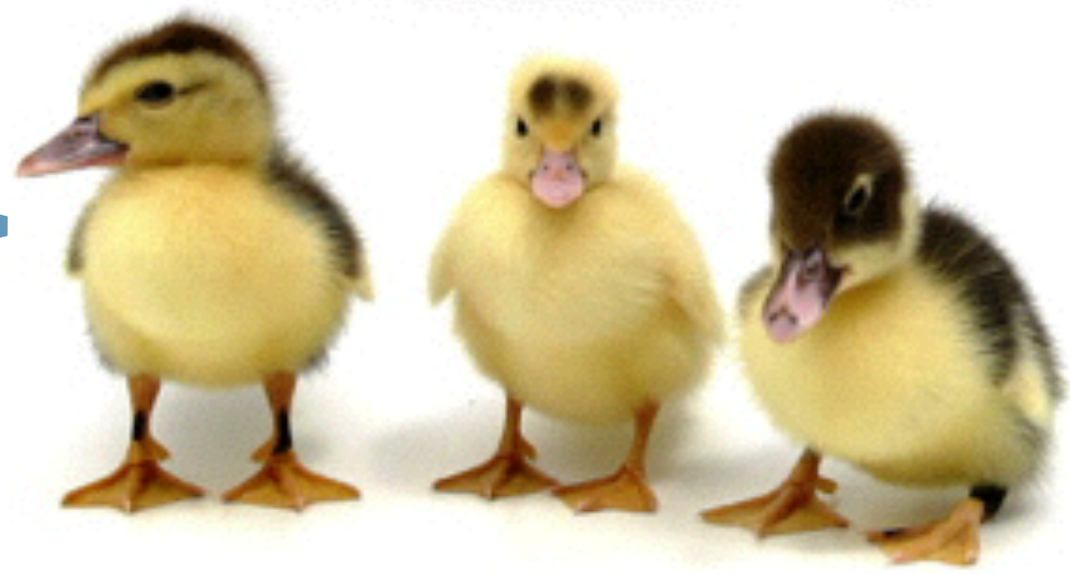


Duck Tails

In Python inheritance is a way to share implement ion

Not satisfy a type system

Python uses duck typing



Summary: Inheritance and Subtype Polymorphism

- Specify single inheritance by putting a base class in parentheses after defining a class's name
- Subclasses have all of the methods of their base class
- It's often best to explicitly call a base class initializer from a subclass's initializer
- If a class with a single base class doesn't define an initializer, the base class's initializer will be called automatically on construction
- `isinstance()` takes an object as its first argument and a type as its second
- `isinstance()` determines if its first argument is an instance of the second argument, or any subclass of the second argument
- `isinstance()` can accept a tuple of types as its second argument, in which it returns `True` if the first argument is of any of those types
- Checking for specific types is rare in Python and is sometimes regarded as bad design

Summary: Inheritance and Subtype Polymorphism

- `isinstance()` determines if its first argument is a direct or indirect subclass of, or the same type as, the second argument
- Multiple inheritance means having more than one direct base class
- You declare multiple base classes with a comma-separated list of class names in parentheses after a class's name in a class definition
- A class can have as many base classes as you want
- Python uses a well-defined "method resolution order" to resolve methods at runtime
- If a multiply-inheriting class defines no initializer, Python will automatically call the initializer of its first base class on construction
- `__bases__` is a tuple of types on a class object which defines the base classes for the class
- `__bases__` is in the same order as in the class definition
- `__bases__` is populated for both single and multiple inheritance
- Method resolution order defines the order in which Python will search an inheritance graph for methods

Summary: Inheritance and Subtype Polymorphism

- MRO is short for Method Resolution Order
- MRO is stored as a tuple of types in the `__mro__` attribute of a class
- The `mro()` method on type objects returns the contents of `__mro__` as a list
- To resolve a method, Python uses the first entry in a class's MRO which has the requested method
- MRO is dependent on base class declaration order
- MRO is calculated by Python using the C3 algorithm
- MRO honors base-class ordering from class definitions
- MRO puts subclasses before base classes
- The relative order of classes in an MRO is consistent across all classes
- It is possible to specify an inconsistent base class ordering, in which case Python will raise a `TypeError` when the class definition is reached
- `super()` operates by using the elements in an MRO that come after some specified type
- `super()` returns a proxy object which forwards calls to the correct objects

Summary: Inheritance and Subtype Polymorphism

- There are two distinct types of `super()` proxies, bound and unbound
- Unbound `super()` proxies are primarily used for implementing other Python features
- Bound proxies can be bound to either class objects or instances
- Calling `super()` with a base-class and derived-class argument returns a proxy bound to a class
- Calling `super()` with a class and an instance of that class returns a proxy bound to an instance
- A `super()` proxy takes the MRO of its second argument (or the type of its second argument), finds the first argument in that MRO, and uses everything after it in the MRO for method resolution
- Since class-bound proxies aren't bound to an instance, you can't directly call instance methods that they resolve for you
- However, `classmethod`s resolved by class-bound proxies can be called directly
- Python will raise a `TypeError` if the second argument is not a subclass or instance of the first argument

Summary: Inheritance and Subtype Polymorphism

- Inappropriate use of `super()` can violate some design constraints
- Calling `super()` with no arguments inside an instance method produces an instance-bound proxy
- Calling `super()` with no arguments inside a `classmethod` produces a class-bound proxy
- In both cases, the no-argument form of `super()` is the same as calling `super()` with the method's class as the first argument and the method's first argument as the second
- Since `super()` works on MROs and not just a class's base classes, class can be designed to cooperate without prior knowledge of one another
- The class `object` is at the core of Python's object model
- `object` is the ultimate base class for all other classes in Python
- If you don't specify a base class for a class, Python automatically uses `object` as the base

Summary: Inheritance and Subtype Polymorphism

- **Because `object` is in every class's inheritance graph, it shows up in every MRO.**
- **`object` provides hooks for Python's comparison operators**
- **`object` provides default `__repr__()` and `__str__()` implementations**
- **`object` implements the core attribute lookup and management functionality in Python**
- **Inheritance in Python is best used as a way to share implementation**