Multiple Inheritance

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Overview

Review of single inheritance

Basic type inspection tools

Multiple inheritance

A small mystery involving class interaction

Resolve mystery by understanding inheritance

Overview

Method resolution order

How `super()` works

The `object` class

Single Inheritance

```
class SubClass(BaseClass):
    . . .
```

Inherits all attributes

May override methods

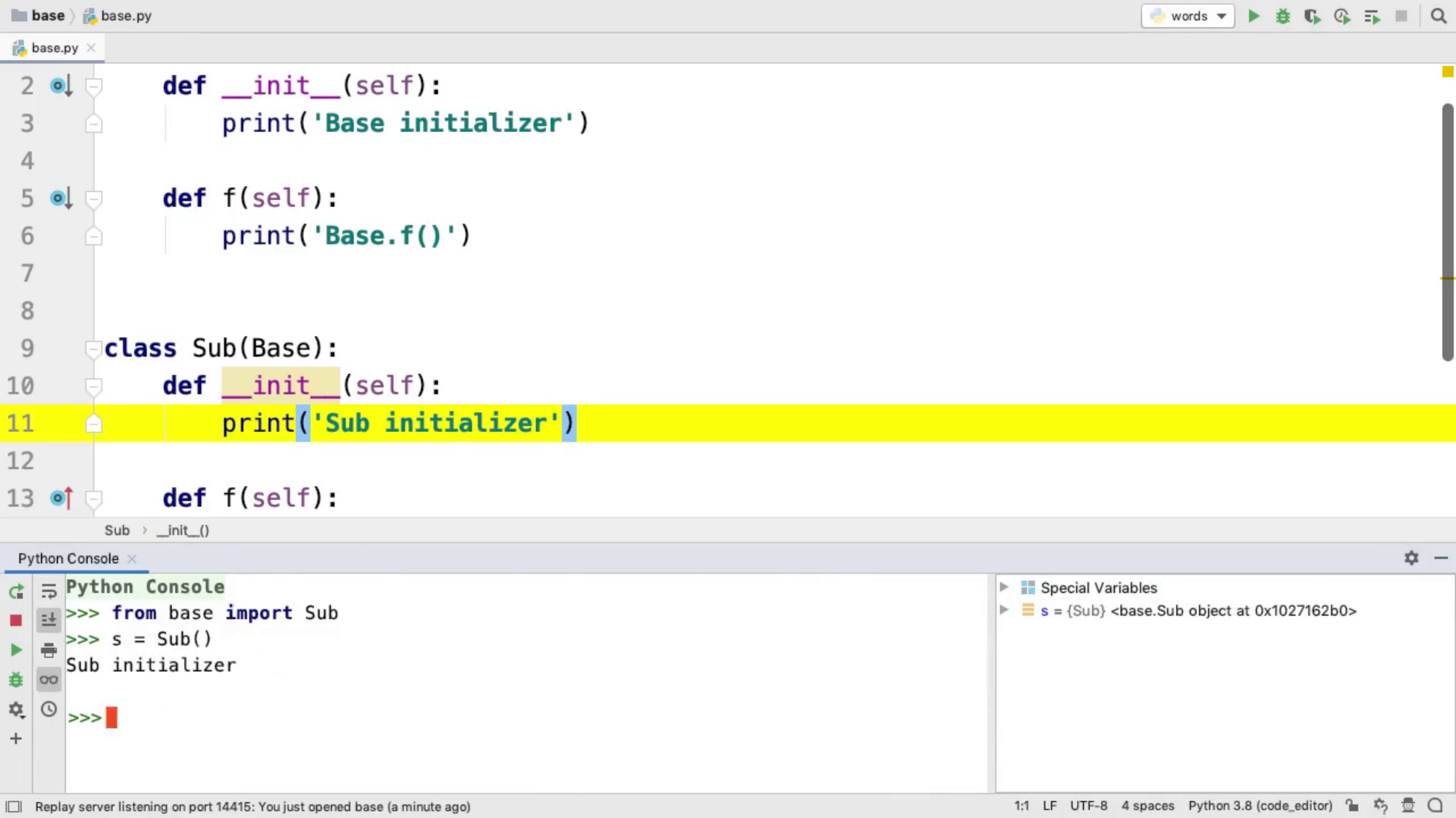
Base Class Initializers

Correct initialization

It's generally necessary to call base class initializers to ensure proper object initialization.

Without override

If a subclass doesn't define an initializer, then the base class initializer is called during construction.


base
base.py
words
base.py
2 def __init__(self):
3 print('Base initializer')
4
5 def f(self):
6 print('Base.f()')
7
8
9 class Sub(Base):
10 def __init__(self):
11 print('Sub initializer')
12
13 def f(self):
Sub › __init__()
Python Console
Python Console
>>> from base import Sub
>>> s = Sub()
Sub initializer
>>>
Special Variables
s = {Sub} <base.Sub object at 0x1027162b0>
Replay server listening on port 14415: You just opened base (a minute ago)
1:1 LF UTF-8 4 spaces Python 3.8 (code_editor)

Base Class Initializers

Unlike C++ and Java, Python doesn't automatically call base class initializers.

`__init__` is treated just like any other method.

If a subclass defines `__init__` , it must explicitly call the base class implementation for it to be run.

words

base.py

```
1  class Base:
2      def __init__(self):
3          print('Base initializer')
4
5      def f(self):
6          print('Base.f()')
7
8
9  class Sub(Base):
10     def __init__(self):
11         super().__init__()
12         print('Sub initializer')
```

Sub › __init__()

Python Console

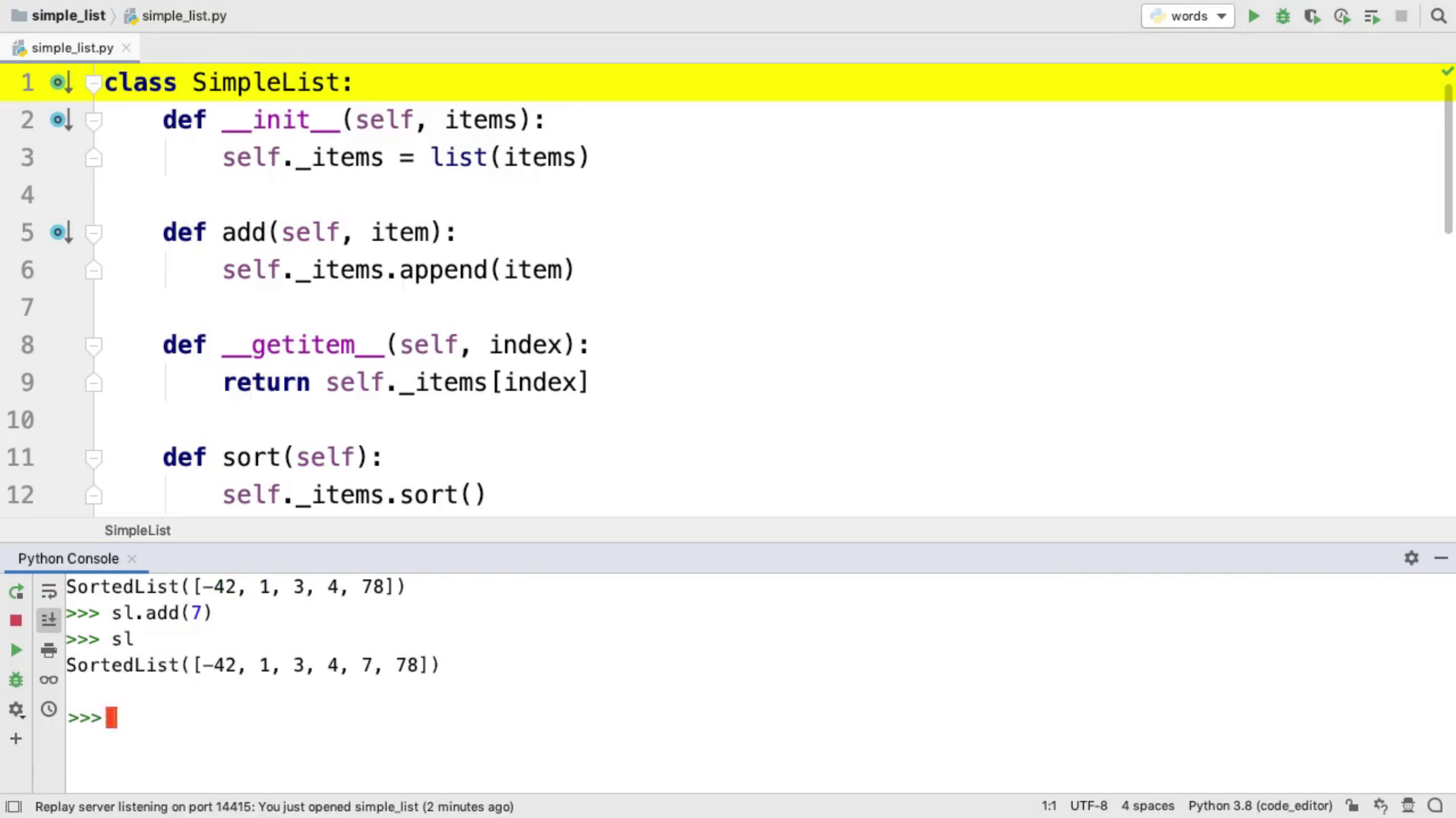
```
>>> from base import Sub
>>> s = Sub()
Base initializer
Sub initializer

>>>
```

Core Python: Getting Started

on

PLURALSIGHT


words
simple_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()
SimpleList
Python Console
SortedList([-42, 1, 3, 4, 78])
>>> sl.add(7)
>>> sl
SortedList([-42, 1, 3, 4, 7, 78])
>>>

Type Inspection

Multiple inheritance is not much more complex than single inheritance.

Both rely on a single underlying model.

A new subclass of `SimpleList` called `IntList`

Constrained to containing only integers

`isinstance()`

Determines if an object is an instance of type.

Takes an object as its first arguments and a type as its second.

Returns `True` of the first argument is an instance of the second.

isinstance()

```
>>> isinstance(3, int)
True
>>> isinstance('hello!', str)
True
>>> isinstance(4.567, bytes)
False
>>> from simple_list import *
>>> sl = SortedList([3, 2, 1])
>>> isinstance(sl, SortedList)
True
>>> isinstance(sl, SimpleList)
True
>>> x = []
>>> isinstance(x, (float, dict, list))
True
>>>
```

Checking Multiple Types

```
isinstance(obj, (type_a, type_b, type_c))
```

simple_list.py

```
31  class IntList(SimpleList):
32      def __init__(self, items=()):
33          for x in items: self._validate(x)
34          super().__init__(items)
35
36      @staticmethod
37      def _validate(x):
38          if not isinstance(x, int):
39              raise TypeError('IntList only supports integer values.')
40
41      def add(self, item):
42          self._validate(item)
```

SimpleList

Python Console

```
    self._validate(item)
  File "/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpn_5rnhx0/build/simple_list/simple_list.py", line 39, in _valid
    raise TypeError('IntList only supports integer values.')
TypeError: IntList only supports integer values.

>>>
```

Type Checks in Python

`isinstance()` can be used for type checking in Python.

Some people consider type checking a sign of poor design.

Sometimes they're the easiest way to solve a problem.

`issubclass()`

Operates on types to check for sub/superclass relationships.

Determines if one class is a subclass of another.

Takes two arguments, both of which must be types.

Returns `True` if the first argument is a subclass of the second.

issubclass()

```
>>> from simple_list import *
>>> issubclass(IntList, SimpleList)
True
>>> issubclass(SortedList, SimpleList)
True
>>> issubclass(SortedList, IntList)
False
>>> class MyInt(int): pass
...
>>> class MyVerySpecialInt(MyInt): pass
...
>>> issubclass(MyVerySpecialInt, int)
True
>>>
```

Core Python: Introspection

on

PLURALSIGHT

Multiple Inheritance

Multiple Inheritance

Defining a class with more than one direct base class

Not universal among object-oriented languages

Can lead to certain complexities

Python has a relatively simple system for dealing with them

Multiple Inheritance Syntax

```
class SubClass(Base1, Base2, Base3):
    . . .
```

Name Resolution with Multiple Base Classes

Classes inherit all methods from all of their bases

If there's no method name overlap, names resolve to the obvious method

In the case of overlap, Python uses a well-defined "method resolution order" to decide which to use

simple_list.py

```
48
49
```

Python Console

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpqx3a5r4o/build/simple_list/simple_list.py", line 33, in __init
    for x in items: self._validate(x)
  File "/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpqx3a5r4o/build/simple_list/simple_list.py", line 39, in _valid
    raise TypeError('IntList only supports integer values.')
TypeError: IntList only supports integer values.
>>> sil.add(-1234)
>>> sil
SortedIntList([-1234, 3, 23, 42])
>>> sil.add("The smallest uninteresting number")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpqx3a5r4o/build/simple_list/simple_list.py", line 42, in add
    self._validate(item)
  File "/var/folders/0k/58g36_tx22xcxqd9mwqzg_h00000gp/T/tmpqx3a5r4o/build/simple_list/simple_list.py", line 39, in _valid
    raise TypeError('IntList only supports integer values.')
TypeError: IntList only supports integer values.

>>>
```

How does `SortedIntList` work?

How does Python know which version of `add()` to call?

How does it seem to know to call **both** of them?

The answer involves "method resolution order" and `super()` .

Base class initialization

If a class uses multiple inheritance and defines no initializer, only the initializer of the first base class is automatically called.

Base Class Initialization

```
>>> class Base1:
...     def __init__(self):
...         print('Base1.__init__')
...
>>> class Base2:
...     def __init__(self):
...         print('Base2.__init__')
...
>>> class Sub(Base1, Base2):
...     pass
...
>>> s = Sub()
Base1.__init__
>>>
```

__bases__

```
>>> from simple_list import SortedIntList, IntList
>>> SortedIntList.__bases__
(<class 'simple_list.IntList'>, <class 'simple_list.SortedList'>)
>>> IntList.__bases__
(<class 'simple_list.SimpleList'>,)
>>>
```

Method Resolution Order

Method Resolution Order

MRO

Ordering of an inheritance graph that determines which implementation to use when invoking a method

Method implementation may be found in any class in an inheritance graph

Determines the order in which the graph is searched when looking for an implementation

MRO is actually quite straightforward.

Method Resolution Order

```
>>> from simple_list import *
>>> SortedIntList.__mro__
(<class 'simple_list.SortedIntList'>, <class 'simple_list.IntList'>, <class 'sim
ple_list.SortedList'>, <class 'simple_list.SimpleList'>, <class 'object'>)
>>>
```

How Is MRO Used?

Python finds the MRO for the type of the object on which a method is invoked

Python checks each class in the MRO in order to find one that implements the method

The first implementation found is used


words
diamond.py
8 return 'B.func'
9
10
11 class C(A):
12 def func(self):
13 return 'C.func'
14
15
16 class D(C, B):
17 pass
18
Python Console
(<class 'diamond.D'>, <class 'diamond.C'>, <class 'diamond.B'>, <class 'diamond.A'>, <class 'object'>)
>>> d = D()
>>> d.func()
'C.func'
>>>

`object`

The ultimate base class for every class in Python.

Method Resolution Order

```
>>> from simple_list import *
>>> SortedIntList.__mro__
(<class 'simple_list.SortedIntList'>, <class 'simple_list.IntList'>, <class 'sim
ple_list.SortedList'>, <class 'simple_list.SimpleList'>, <class 'object'>)
>>>
```

The definition of `IntList` makes no mention of `SortedList`

`SortedIntList` maintains both the sorting and type constraints

How is `IntList.add()` delegating to `SortedList.add()` ?

Calculating Method Resolution Order

C3

Algorithm used to calculate method resolution orders

Ensures that:

- **Subclasses come before base classes**
- **Base class order from class definition is preserved**
- **The first two qualities are preserved for all MROs in a program**

C3 prohibits some inheritance declarations in Python.

Invalid Inheritance Graphs

```
>>> class A: pass
...
>>> class B(A): pass
...
>>> class C(A): pass
...
>>> class D(B, A, C): pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, C
>>>
```

super()

```
class SortedList(SimpleList):

    def add(self, item):

        super().add(item)

        self.sort()
```

◀ Access the "base class part" of an object

super()

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

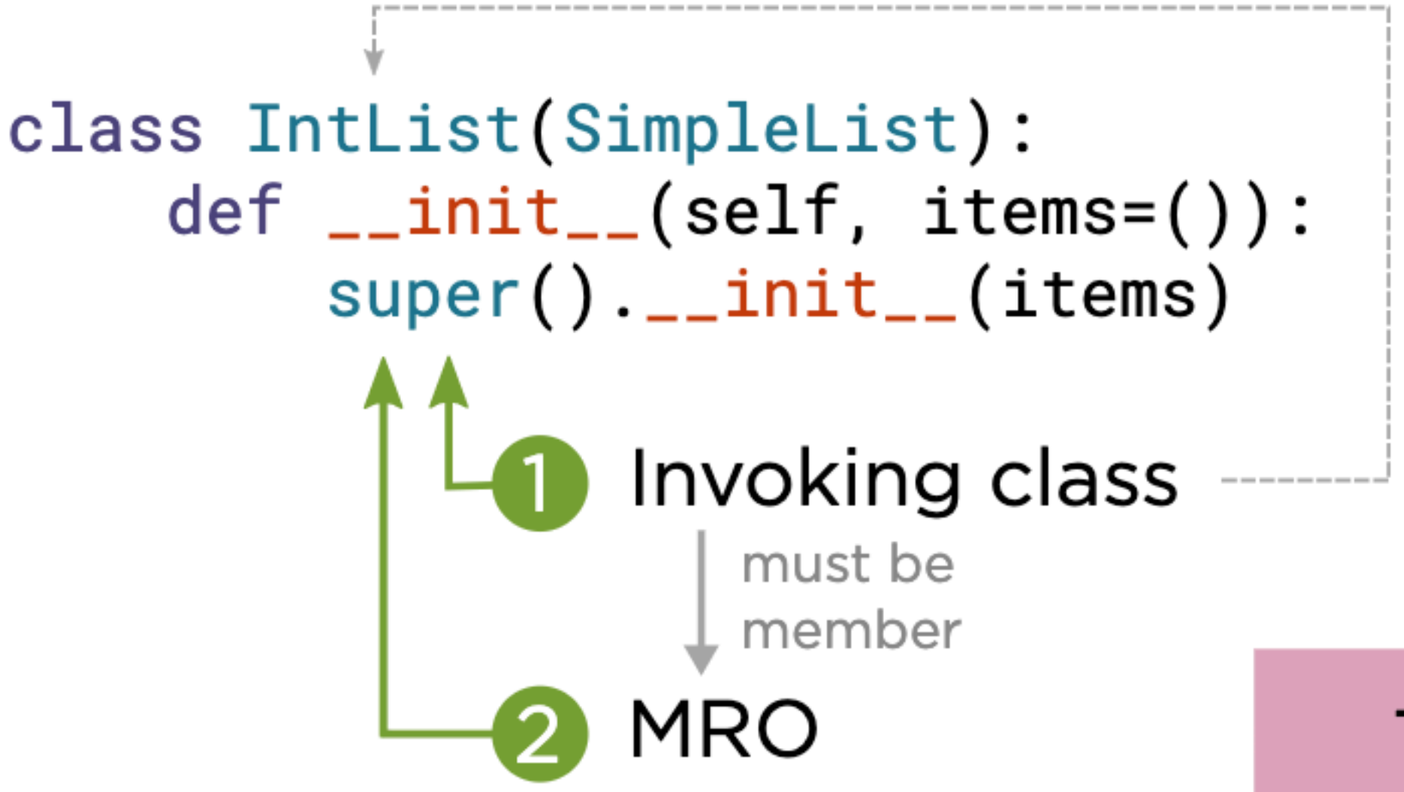
super() works with the MRO of an object, not just its base classes.

`super()` gives you a proxy object

The proxy resolves the correct implementation if any requested method

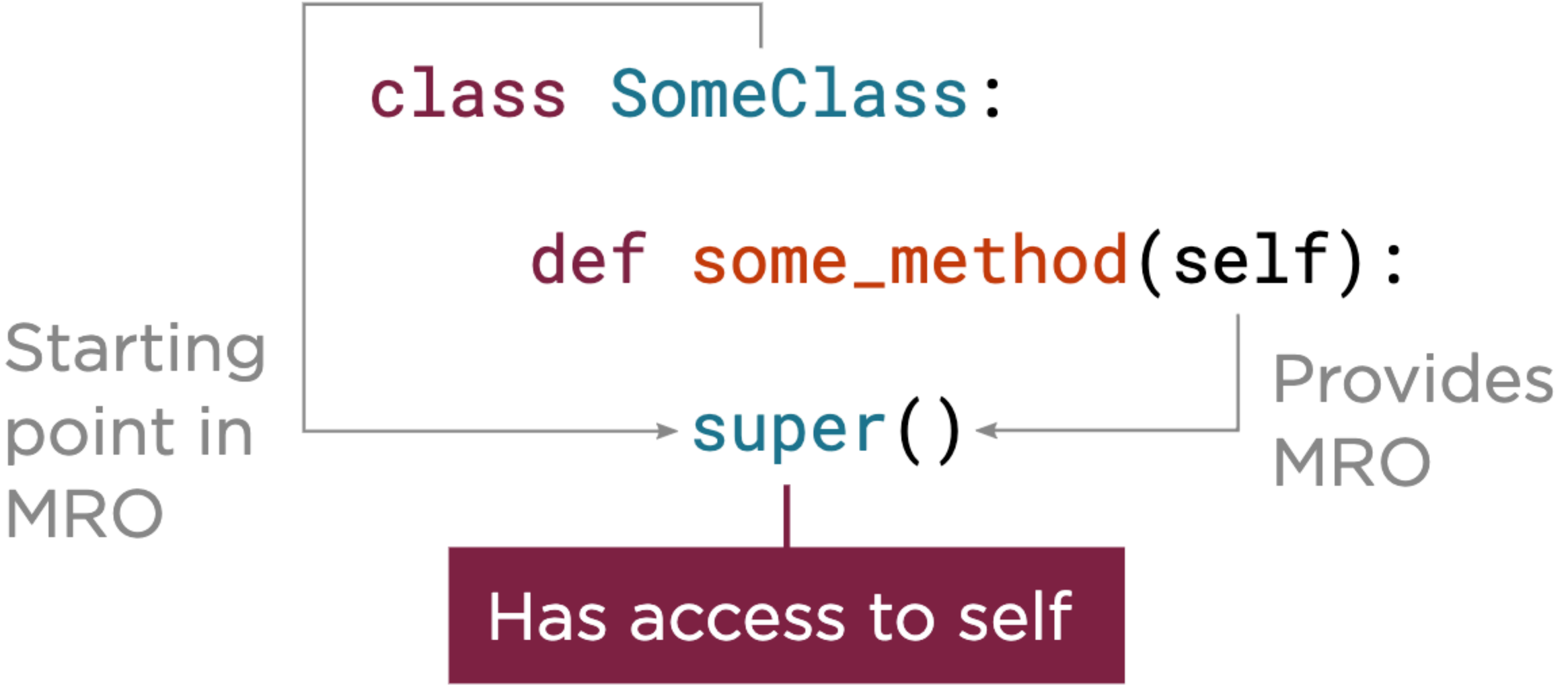
It has access to the entire inheritance graph of the object

super()



The proxy resolves **__init__** using classes in 2 after 1

Instance-bound Super Proxies

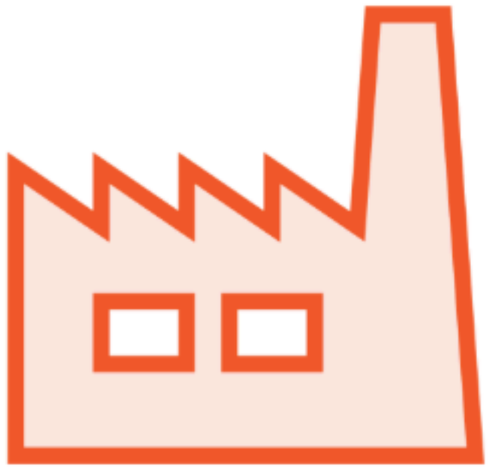



words
simple_list.py
32 def __init__(self, items=()):
33 for x in items: self._validate(x)
34 s = super()
35 print(s)
36 s.__init__(items)
37 print(s.__init__)
38
39 @staticmethod
40 def _validate(x):
41 if not isinstance(x, int):
42 raise TypeError('IntList only supports integer values.')
43
IntList › __init__()
Python Console
>>> SortedIntList()
<super: <class 'IntList'>, <SortedIntList object>>
<bound method SortedList.__init__ of SortedIntList([])>
SortedIntList([])
>>>
Replay server listening on port 14415: You just opened simple_list (2 minutes ago)
1:1 LF UTF-8 4 spaces Python 3.8 (code_editor)

`super()` uses the full MRO of an object, not just the base classes from a class definition.

Class-bound Super Proxies

Class-bound Super Proxies



What happens if you use `super()` in a classmethod?

We don't have an instance to work with, but we do have a class object .

`super()` derives the MRO from the class object rather than the type of `self` .

words

animals.py

```
9     def description(cls):
10        s = super()
11        print(s)
12        print(s.description)
13        return s.description() + " with wings"
14
15
16 class Flamingo(Bird):
17    @classmethod
18    def description(cls):
19        return super().description() + " and fabulous pink feathers"
20
```

Bird › description()

Python Console

```
>>> Flamingo.description()
<super: <class 'Bird'>, <Flamingo object>>
<bound method Animal.description of <class 'animals.Flamingo'>>
'An animal with wings and fabulous pink feathers'

>>>
```

Special Variables

For the most part, `super()` will behave in an intuitive way.

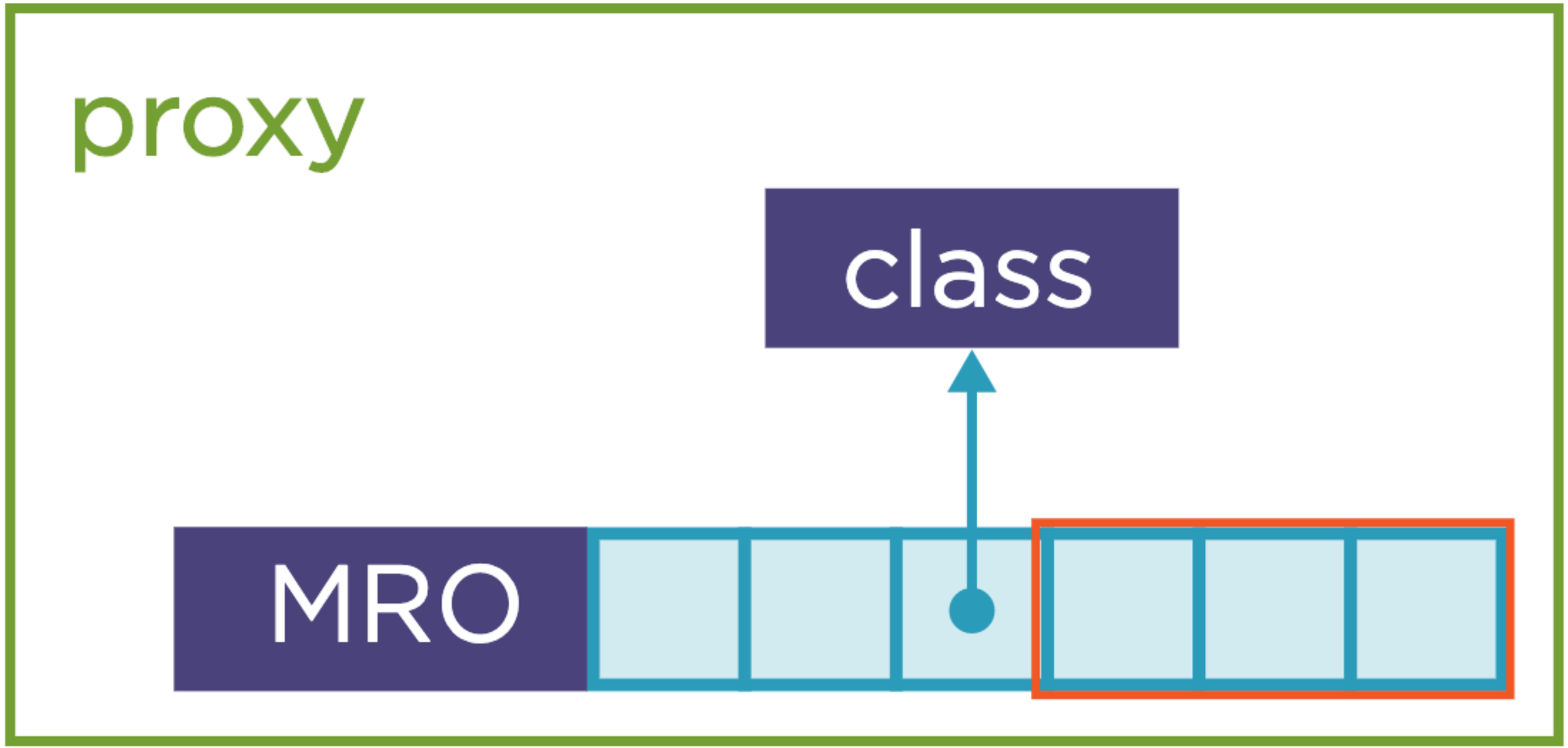
Explicit Arguments to `super()`

```
super(class-object, instance-or-class)
```

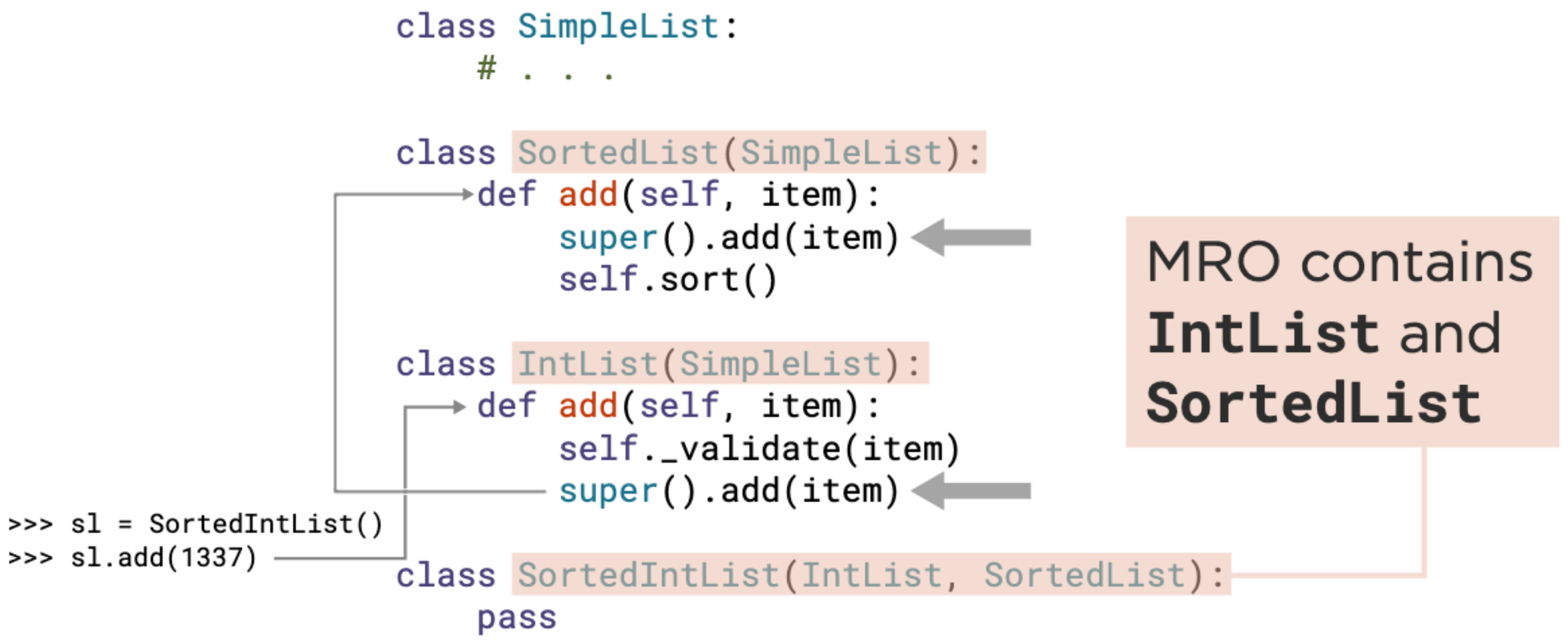
Explicit `super()` Arguments

```
>>> from simple_list import *
>>> s = SortedIntList()
>>> super(IntList, s).add
<bound method SortedList.add of SortedIntList([])>
>>> super(IntList, s).add("I am not a number! I am a free man!")
>>> s
SortedIntList(['I am not a number! I am a free man!'])
>>>
```

super()



Resolution


class SimpleList:
# . . .
class SortedList(SimpleList):
def add(self, item):
super().add(item)
self.sort()
class IntList(SimpleList):
def add(self, item):
self._validate(item)
super().add(item)
class SortedIntList(IntList, SortedList):
pass
>>> sl = SortedIntList()
>>> sl.add(1337)
MRO contains
IntList and
SortedList

If you understand how `SortedIntList` works, then you have a good understanding of multiple inheritance, `super()`, and method resolution order.

object

object

```
>>> from simple_list import *
>>> IntList.__mro__
(<class 'simple_list.IntList'>, <class 'simple_list.SimpleList'>, <class 'object
'>)
>>> SortedIntList.__mro__
(<class 'simple_list.SortedIntList'>, <class 'simple_list.IntList'>, <class 'sim
ple_list.SortedList'>, <class 'simple_list.SimpleList'>, <class 'object'>)
>>> list.__mro__
(<class 'list'>, <class 'object'>)
>>> int.__mro__
(<class 'int'>, <class 'object'>)
>>> class NoBaseClass: pass
...
>>> NoBaseClass.__bases__
(<class 'object'>,)
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge
__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>>
```

In Python, the type of an object doesn't determine if it can be used in a particular context.

Python uses duck typing where fitness for purpose is determined at the time of use.

Functions don't specify their types.

You can call any method on any object, and Python won't complain until runtime.

Inheritance in Statically Typed Languages

Type matching

Compilers enforce that only objects of the correct type are passed to functions

Development effort

Satisfying the type system can become a very significant element in your development effort

Inheritance for Implementation Sharing

With no need to satisfy a static type system, you don't need to use inheritance in Python to bestow objects with types.

Instead, inheritance in Python is best used for sharing implementation.

Summary

- Use a comma-separated list of class names for multiple base classes
- A class can have as many base classes as you want
- You should generally explicitly initialize base classes
- Python will call, at most, the initializer of the first of multiple base class
- Python will call a base class initializer only if the subclass doesn't define one

Summary

__bases__ is a tuple defining the base classes for the class

__bases__ is in the same order as in the class definition

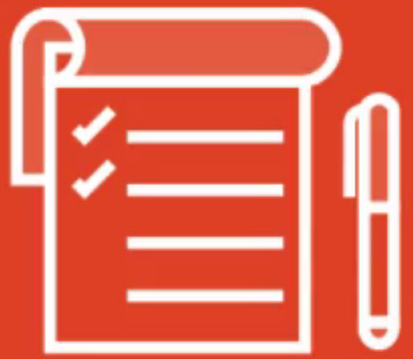
__bases__ is populated for both single and multiple inheritance

Method resolution order is the order in which Python searches an inheritance graph

MRO is a tuple of types in the __mro__ attribute

Python uses the first entry in an MRO which has the method

Summary



MRO is dependent on base class declaration order

MRO is calculated by Python using the C3 algorithm

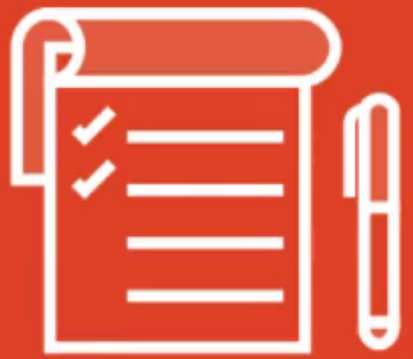
C3 preserves base-class declaration order

C3 puts subclasses before base classes

It is possible to specify an inconsistent base class ordering

`super()` uses the elements in an MRO after some specified type

Summary



`super()` returns a proxy object

A `super` proxy uses a subset of an MRO for name resolution

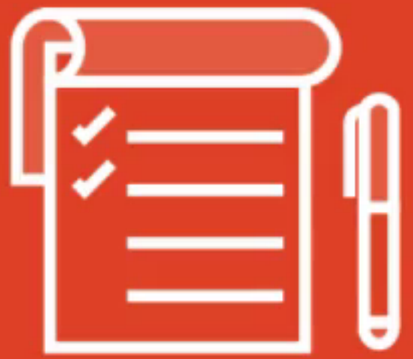
You can't directly call instance methods on class-bound proxies

Inappropriate use of `super()` can violate design constraints

Classes can be designed to cooperate without a priori knowledge of one another

`object` is at the core of Python's object model

Summary



`object` is the ultimate base class for all other classes

Python will automatically provide `object` as a base class

`object` provides default implementations of many common Python methods

`object` implements the core attribute functionality in Python

Inheritance in Python is best used as a way to share implementation