Multiple Inheritance

SURP 2022 Python Bootcamp

Ohio State Astronomy

Slides by: James W. Johnson

What is it?

When an object inherits functionality from more than one base class

Objectives

- The basic syntax
- The method resolution order
- A simple example: A piece-wise function

The Basic Syntax

By definition, at least 2 classes to inherit from are required

```
In [1]: class A:
   ...:     def __init__(self, x):
   ...:         print("Start A.__init__")
   ...:         self.a = x
   ...:         print("End A.__init__")
   ...:

In [2]: class B:
   ...:     def __init__(self, x):
   ...:         print("Start B.__init__")
   ...:         self.b = x
   ...:         print("End B.__init__")
   ...:
```

The Basic Syntax

By definition, at least 2 classes to inherit from are required

Where you already specify the class to inherit from with single inheritance, add any additional classes! Simple, right? …*right?*

```
In [3]: class C(A, B):
   ...:     def __init__(self, x):
   ...:         print("Start C.__init__")
   ...:         super().__init__(x)
   ...:         self.c = x
   ...:         print("End C.__init__")
   ...:
```

The Basic Syntax

By definition, at least 2 classes to inherit from are required

Where you already specify the class to inherit from with single inheritance, add any additional classes! Simple, right? …*right?*

Unfortunately, multiple inheritance sort of breaks *super*

```
In [4]: example = C(1)
Start C.__init__
Start A.__init__
End A.__init__
End C.__init__

In [5]: example.a
Out[5]: 1

In [6]: example.b
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-6-41c8445e582b> in <module>
----> 1 example.b

AttributeError: 'C' object has no attribute 'b'

In [7]: example.c
Out[7]: 1
```

The Method Resolution Order

When you call a function within a class, it looks first within that class, then the first parent class, then the second parent class, and so on

Any call to *super* is a call to classes further down the MRO

- This has been true all along for single inheritance too!
- With multiple inheritance, *super* may not always do what you want

```
In [3]: class C(A, B):
   ...:     def __init__(self, x):
   ...:         print("Start C.__init__")
   ...:         super().__init__(x)
   ...:         self.c = x
   ...:         print("End C.__init__")
   ...:
```

```
In [8]: C.mro()
Out[8]: [__main__.C, __main__.A, __main__.B, object]
```

The Basic Syntax

Instead, invoke the inherited class directly and pass *self* as the first argument

- This takes advantage of the fact that *x.function(y)* is equivalent to *classname.function(x, y)*

```
In [3]: class C(A, B):
   ...:     def __init__(self, x):
   ...:         print("Start C.__init__")
   ...:         A.__init__(self, x)
   ...:         B.__init__(self, x)
   ...:         self.c = x
   ...:         print("End C.__init__")
   ...:
```

```
In [4]: example = C(1)
Start C.__init__
Start A.__init__
End A.__init__
Start B.__init__
End B.__init__
End C.__init__

In [5]: example.a
Out[5]: 1

In [6]: example.b
Out[6]: 1

In [7]: example.c
Out[7]: 1
```

Example: A Piece-Wise Function

The two base classes

- *exponential* describes a classic e-folding function
- *sinusoid* describes a sine or cosine function

```
# import the necessary pieces
from .exponential import exponential
from .sinusoid import sinusoid


class exposinusoid(exponential, sinusoid):

    r"""
    A mathematical function which is a sinusoid for x < 0 but an exponential
    for x >= 0.

    Parameters
    ----------
    kwargs : real numbers
        The attributes of the ``exponential`` and ``sinusoid`` classes.

    Attributes are inherited from the ``exponential`` and ``sinusoid``
    classes.
    """

    def __init__(self, amplitude = 1, frequency = 1, phase = 0,
        normalization = 1, rate = 1):

        exponential.__init__(self, normalization = normalization, rate = rate)
        sinusoid.__init__(self, amplitude = amplitude, frequency = frequency,
            phase = phase)
```

Example: A Piece-Wise Function

The two base classes

- *exponential* describes a classic e-folding function
- *sinusoid* describes a sine or cosine function

Invoking the inherited class directly can also be used to write the *__call__* function (or any function for that matter)

```
class exposinusoid(exponential, sinusoid):

    r"""
    A mathematical function which is a sinusoid for x < 0 but an exponential
    for x >= 0.

    Parameters
    ----------
    kwargs : real numbers
        The attributes of the ``exponential`` and ``sinusoid`` classes.

    Attributes are inherited from the ``exponential`` and ``sinusoid``
    classes.
    """

    def __init__(self, amplitude = 1, frequency = 1, phase = 0,
        normalization = 1, rate = 1):

        exponential.__init__(self, normalization = normalization, rate = rate)
        sinusoid.__init__(self, amplitude = amplitude, frequency = frequency,
            phase = phase)

    def __call__(self, x):
        if x >= 0:
            return exponential.__call__(self, x)
        else:
            return sinusoid.__call__(self, x)
```

The Method Resolution Order in Action

Replaced the body of the *__call__* function with a simple call to *super*

```
class exposinusoid(exponential, sinusoid):

    r"""
    A mathematical function which is a sinusoid for x < 0 but an exponential
    for x >= 0.

    Parameters
    ----------
    kwargs : real numbers
        The attributes of the ``exponential`` and ``sinusoid`` classes.

    Attributes are inherited from the ``exponential`` and ``sinusoid``
    classes.
    """

    def __init__(self, amplitude = 1, frequency = 1, phase = 0,
        normalization = 1, rate = 1):

        exponential.__init__(self, normalization = normalization, rate = rate)
        sinusoid.__init__(self, amplitude = amplitude, frequency = frequency,
            phase = phase)

    def __call__(self, x):
        return super().__call__(x)
        # if x >= 0:
        #   return exponential.__call__(self, x)
        # else:
        #   return sinusoid.__call__(self, x)
```

The Method Resolution Order in Action

Replaced the body of the *__call__* function with a simple call to *super*

Now it's only calling *exponential.__call__*

If *exposinusoid* inherited from *sinusoid* first, this would find *sinusoid.__call__* instead

```
In [1]: from mypkg.mathlib import exposinusoid

In [2]: x = exposinusoid()

In [3]: x(0)
Out[3]: 1.0

In [4]: x(-1)
Out[4]: 0.36787944117144233

In [5]: x(-2)
Out[5]: 0.1353352832366127

In [6]: x(-3)
Out[6]: 0.049787068367863944
```

Footnotes

In some specific instances, *super* is smart enough to make sure all inherited classes' *__init__* functions get called

- Example: "diamond inheritance" (B and C inherit from A, D inherits from B and C)

"Mixin" classes – designed for multiple inheritance

- Generally implement only one function each, then "mix" them by inheriting from multiple
- Conventionally have names ending in -Mixin

Footnotes

Developers often argue that multiple inheritance is bad practice

- This really only means it should be used sparingly, when no other options are available

Advice: Don't be afraid to use it when it offers a concise, readable solution

- …but only if the single inheritance version is noticeably less so
- Avoid *super* – it sacrifices readability. "Explicit is better than implicit."
  - With this approach, changing functionality requires changing the lines that implement that functionality, which means it's good code.