Object-Oriented Python!

classes, subclasses, self and super

Kate MacInnis

PyLadies-ATX Tech Talk
July 18, 2016

In this presentation...

All code samples are written with Python 3 in mind.

We will be editing code in a text editor, and running it in the interactive console. When we make changes to our text file, we will need to do the following:

if using the built-in console:

```
quit()
python3
from example import *
```

if using ipython:

```
run example.py
```

You will need to do this after each change to the code in a text file. Alternatively, you could use an IDE, such as PyCharm, and that might make the process easier as well.
To make life easier, code examples have been put online at:
pydancing.wordpress.com/oop

Before we begin. . . some nifty python tricks

Save this in a text file called stars.py:

```
def func0(myvar):
    print('myvar = {}'.format(myvar))
    print('type(myvar) = {}'.format(type(myvar)))

def func1(*args):
    print('args = {}'.format(args))
    print('type(args) = {}'.format(type(args)))

def func2(myvar, *args):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))

def func3(myvar, *args, **kwargs):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))
    print('kwargs = {}'.format(kwargs))
    print('type(kwargs) = {}'.format(type(kwargs)))
```

Nifty python tricks

stars.py (for reference)

```
def func0(myvar):
    print('myvar = {}'.format(myvar))
    print('type(myvar) = {}'.format(type(myvar)))

def func1(*args):
    print('args = {}'.format(args))
    print('type(args) = {}'.format(type(args)))

def func2(myvar, *args):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))

def func3(myvar, *args, **kwargs):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))
    print('kwargs = {}'.format(kwargs))
    print('type(kwargs) = {}'.format(type(kwargs)))
```

Predict the outcome of each of these, and test them in the console:

```
func0(5)
func0()
func0(1,2,3)
```

Nifty python tricks

stars.py (for reference)

```
def func0(myvar):
    print('myvar = {}'.format(myvar))
    print('type(myvar) = {}'.format(type(myvar)))

def func1(*args):
    print('args = {}'.format(args))
    print('type(args) = {}'.format(type(args)))

def func2(myvar, *args):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))

def func3(myvar, *args, **kwargs):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))
    print('kwargs = {}'.format(kwargs))
    print('type(kwargs) = {}'.format(type(kwargs)))
```

Predict the outcome of each of these, and test them in the console:

```
func0(5)
func0()
func0(1,2,3)
```

```
func1(5)
func1()
func1(1,2,3)
```

Nifty python tricks

stars.py (for reference)

```
def func0(myvar):
    print('myvar = {}'.format(myvar))
    print('type(myvar) = {}'.format(type(myvar)))

def func1(*args):
    print('args = {}'.format(args))
    print('type(args) = {}'.format(type(args)))

def func2(myvar, *args):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))

def func3(myvar, *args, **kwargs):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))
    print('kwargs = {}'.format(kwargs))
    print('type(kwargs) = {}'.format(type(kwargs)))
```

And these:

```
func2(1,2,3,4,5)
func2()
func2(1)
func2(colors)
```

Nifty python tricks

stars.py (for reference)

```
def func0(myvar):
    print('myvar = {}'.format(myvar))
    print('type(myvar) = {}'.format(type(myvar)))

def func1(*args):
    print('args = {}'.format(args))
    print('type(args) = {}'.format(type(args)))

def func2(myvar, *args):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))

def func3(myvar, *args, **kwargs):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))
    print('kwargs = {}'.format(kwargs))
    print('type(kwargs) = {}'.format(type(kwargs)))
```

And these:

```
func2(1,2,3,4,5)
func2()
func2(1)
func2(colors)
```

```
func2(*colors)
func2(*triples)
func2(*range(8))
```

Nifty python tricks

stars.py (for reference)

```
def func0(myvar):
    print('myvar = {}'.format(myvar))
    print('type(myvar) = {}'.format(type(myvar)))

def func1(*args):
    print('args = {}'.format(args))
    print('type(args) = {}'.format(type(args)))

def func2(myvar, *args):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))

def func3(myvar, *args, **kwargs):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))
    print('kwargs = {}'.format(kwargs))
    print('type(kwargs) = {}'.format(type(kwargs)))
```

And these:

```
func3(1,2,3,4,5)
func3(1,2,3,a=88,b=99)
func3(a=55,b=66,myvar=0)
func3(mathyfolks)
func3(0, mathyfolks)
```

Nifty python tricks

stars.py (for reference)

```
def func0(myvar):
    print('myvar = {}'.format(myvar))
    print('type(myvar) = {}'.format(type(myvar)))

def func1(*args):
    print('args = {}'.format(args))
    print('type(args) = {}'.format(type(args)))

def func2(myvar, *args):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))

def func3(myvar, *args, **kwargs):
    print('myvar = {}'.format(myvar))
    print('args = {}'.format(args))
    print('kwargs = {}'.format(kwargs))
    print('type(kwargs) = {}'.format(type(kwargs)))
```

And these:

```
func3(1,2,3,4,5)
func3(1,2,3,a=88,b=99)
func3(a=55,b=66,myvar=0)
func3(mathyfolks)
func3(0, mathyfolks)
```

```
func3(0,**mathyfolks)
func3(*colors,**mathyfolks)
func3(**mathyfolks)
func3(**junk)
```

Star Operators

Star Operators

- Within the parameters of a function definition:
- Within the arguments of a function call:

Star Operators

- Within the parameters of a function definition:
  - `*` packs all the extra positional arguments into a tuple
- Within the arguments of a function call:

Star Operators

- Within the parameters of a function definition:
  - `*` packs all the extra positional arguments into a tuple
- Within the arguments of a function call:
  - `*` unpacks a tuple, list, or other sequence type into positional arguments

Star Operators

- Within the parameters of a function definition:
  - `*` packs all the extra positional arguments into a tuple
  - `**` packs all the extra keyword arguments into a dictionary
- Within the arguments of a function call:
  - `*` unpacks a tuple, list, or other sequence type into positional arguments

Star Operators

- Within the parameters of a function definition:
  - `*` packs all the extra positional arguments into a tuple
  - `**` packs all the extra keyword arguments into a dictionary
- Within the arguments of a function call:
  - `*` unpacks a tuple, list, or other sequence type into positional arguments
  - `**` unpacks a dictionary into keyword arguments

And now…

And now…

Object-Oriented Programming

A quick note...

For this presentation, we won't always begin with the "right" way to do things.

I will frequently start with a not-great approach, just so we can examine the problems. We will refactor the code and make it better.

A quick note. . .

For this presentation, we won't always begin with the "right" way to do things.

I will frequently start with a not-great approach, just so we can examine the problems. We will refactor the code and make it better.

Programming Paradigms

In **Procedural Programming**, the focus is on step-by-step instructions for the computer. These procedures will operate on data, but the procedures and the data structures are sort of separate concepts.

Programming Paradigms

In **Procedural Programming**, the focus is on step-by-step instructions for the computer. These procedures will operate on data, but the procedures and the data structures are sort of separate concepts.

In **Object-Oriented Programming**, the code is conceptualized around objects that bundle together relevant values (attributes) and functions (methods).

What kinds of objects can you make?

- Bank Account
  - attributes: balance, minimum balance, interest rate
  - methods: deposit, withdraw, check balance
- GUI Widget
  - attributes: location, size, color
  - methods: click, drag, mouseover
- Game Character
  - attributes: strength, experience points, inventory
  - methods: move, interact, attack
- User
  - attributes: name, password, other profile information
  - methods: log in, edit profile,

Our first homemade object

Let's create our first class:

```python
class Thing:
    pass
```

That's all it takes!

Our first homemade object

Let's create our first class:

```
class Thing:
    pass
```

That's all it takes!

Now that we have a new class, we can create an **instance** of that class:

```
thing = Thing()
```

Our first homemade object

Let's create our first class:

```python
class Thing:
    pass
```

That's all it takes!

Now that we have a new class, we can create an **instance** of that class:

```python
thing = Thing()
```

And we can assign attributes to this object:

```python
thing.a = 5
thing.b = 'hi there'
```

Our first homemade object

Let's create our first class:

```python
class Thing:
    pass
```

That's all it takes!

Now that we have a new class, we can create an **instance** of that class:

```python
thing = Thing()
```

And we can assign attributes to this object:

```python
thing.a = 5
thing.b = 'hi there'
```

Of course, it doesn't actually do anything...

Our first interesting object

Most interesting objects need to be **initialized**.

Our first interesting object

Most interesting objects need to be **initialized**.

In python, we define an `__init__` method on our class to handle the initialization.

Our first interesting object

Most interesting objects need to be **initialized**.

In python, we define an `__init__` method on our class to handle the initialization.

```
class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length
```

(Note that there are double underscores at the start and end of the word init.)

Our first interesting object

To make this example more useful,
lets add another method.

```python
class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def area(self):
        return self.width * self.length
```

Our first interesting object

To make this example more useful, lets add another method.

```
class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def area(self):
        return self.width * self.length
```

Let's make a rectangle!
In the console:

```
from shapes import *
r = Rectangle(3,5)
r.width
r.length
r.area
r.area()
dir(r)
isinstance(r,Rectangle)
r
```

Our first interesting object

To make this example more useful,
lets add another method.

```
class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def area(self):
        return self.width * self.length
```

Let's make a rectangle!
In the console:

```
from shapes import *
r = Rectangle(3,5)
r.width
r.length
r.area
r.area()
dir(r)
isinstance(r,Rectangle)
r
```

We will want a better representation of our rectangle than `<__main__.Rectangle at 0x101e45ba8>`

Better represenation

There are two special methods that we can use to return a string representation of our objects: `__str__` and `__repr__`.

Better represenation

There are two special methods that we can use to return a string representation of our objects: `__str__` and `__repr__`.

- `__str__` should be used for a string representation that is useful to the end-user.

Better represenation

There are two special methods that we can use to return a string representation of our objects: `__str__` and `__repr__`.

- `__str__` should be used for a string representation that is useful to the end-user.
- `__repr__` should be used for a string representation that is useful to the programmer. This is what the console shows.

Better representation

Let's add a `__repr__` method to our class:

```
class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def __repr__(self):
        return 'Rectangle(%s,%s)' % (self.width, self.length)

    def area(self):
        return self.width * self.length
```

Load the new code in the interactive interpreter, make a new rectangle object, and see how it is displayed.

One more method...maybe an attribute?

Suppose we need the length of the diagonal of our rectangle.

One more method…maybe an attribute?

Suppose we need the length of the diagonal of our rectangle.

Since the diagonal of a rectangle with sides l and w is $\sqrt{l^2 + w^2}$, we can write a method:

One more method. . . maybe an attribute?

Suppose we need the length of the diagonal of our rectangle.

Since the diagonal of a rectangle with sides l and w is $\sqrt{l^2 + w^2}$, we can write a method:

```
from math import sqrt

class Rectangle:
        ⋮

    def diagonal(self):
        return round(sqrt(self.length**2 + self.width**2), 2)
```

One more method…maybe an attribute?

Suppose we need the length of the diagonal of our rectangle.

Since the diagonal of a rectangle with sides l and w is $\sqrt{l^2 + w^2}$, we can write a method:

```
from math import sqrt

class Rectangle:
        ⋮
    @property
    def diagonal(self):
        return round(sqrt(self.length**2 + self.width**2), 2)
```

The `@property` decorator makes a method seem like an attribute.

Subclassing

We can create a subclass of any existing class— whether that class is a built-in class or something that we′ve made ourselves.

Subclassing

We can create a subclass of any existing class— whether that class is a built-in class or something that we've made ourselves.

In shapes.py, add:

```
class Square(Rectangle):
    def __init__(self, side):
        self.side = side
```

Subclassing

We can create a subclass of any existing class— whether that class is a built-in class or something that we′ve made ourselves.

In shapes.py, add:

```
class Square(Rectangle):
    def __init__(self, side):
        self.side = side
```

In the console:

```
s = Square(5)
dir(s)
```

Subclassing

We can create a subclass of any existing class— whether that class is a built-in class or something that we've made ourselves.

In shapes.py, add:

```
class Square(Rectangle):
    def __init__(self, side):
        self.side = side
```

In the console:

```
s = Square(5)
dir(s)
```

Where does the `area` method come from?
What happens when you run `s.area()` ?

Subclassing

When subclassing, you must make sure that all necessary attributes for the superclass are initialized on your subclass.

We could do that manually, or we could use the `super` function*.

The `super` function returns the superclass of the class you're currently writing.

Subclassing

We can fix our `Square` class like this:

```
class Square(Rectangle):
    def __init__(self, side):
        self.side = side
        super().__init__(side, side)
```

Subclassing

We can fix our `Square` class like this:

```
class Square(Rectangle):
    def __init__(self, side):
        self.side = side
        super().__init__(side, side)
```

Different in Python 2

Subclassing

We can fix our `Square` class like this:

```
class Square(Rectangle):
    def __init__(self, side):
        self.side = side
        super().__init__(side, side)
```

Different in Python 2

Now, try this in the terminal:

```
s = Square(8)
s.side
s.width
s.height
s.area()
```

```
isinstance(s, Square)
isinstance(s, Rectangle)
s
```

Refactoring time

When you find yourself having to make lots and lots of little changes every time you make a new subclass, it's time to take a step back, think over your entire project, and refactor your code.

Let's create a new base class, `Shape`, which we won't instantiate directly, but will hold the some code that will be common to all our shapes.

Refactoring time

```python
class Shape:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs
```

Refactoring time

```
class Shape:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs

    def __repr__(self):
```

Refactoring time

```python
class Shape:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs

    def __repr__(self):
        shape = self.__class__.__name__
```

Refactoring time

```
class Shape:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs

    def __repr__(self):
        shape = self.__class__.__name__
        all_args = []
        for arg in self.args:
            all_args.append(repr(arg))
```

Refactoring time

```python
class Shape:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs

    def __repr__(self):
        shape = self.__class__.__name__
        all_args = []
        for arg in self.args:
            all_args.append(repr(arg))
        for kw, arg in self.kwargs.items():
            all_args.append('{0}={1}'.format(kw, repr(arg)))
```

Refactoring time

```python
class Shape:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs

    def __repr__(self):
        shape = self.__class__.__name__
        all_args = []
        for arg in self.args:
            all_args.append(repr(arg))
        for kw, arg in self.kwargs.items():
            all_args.append('{0}={1}'.format(kw, repr(arg)))
        comma_sep_args = ', '.join(all_args)
```

Refactoring time

```
class Shape:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs

    def __repr__(self):
        shape = self.__class__.__name__
        all_args = []
        for arg in self.args:
            all_args.append(repr(arg))
        for kw, arg in self.kwargs.items():
            all_args.append('{0}={1}'.format(kw, repr(arg)))
        comma_sep_args = ', '.join(all_args)
        return '{0}({1})'.format(shape, comma_sep_args)
```

Changes to the Base Class

Let's assume that we want every subclass of shape to be able to calculate its area.

Changes to the Base Class

Let's assume that we want every subclass of shape to be able to calculate its area.

```
class Shape:
        ⋮
    def area(self):
        raise NotImplementedError
```

Changes to the Base Class

Let's assume that we want every subclass of shape to be able to calculate its area.

```
class Shape:
        ⋮
    def area(self):
        raise NotImplementedError
```

This doesn't force subclasses to implement an area method, it's just a convention for working with other programmers.

Changes to the Base Class

Finally, let's make `shape` a property of our base class:

```
class Shape:
         ⋮
    @property
    def shape(self):
        return self.__class__.__name__
```

Now let's get subclassin'

There are two main ways that we could handle the parameters for initializing the Rectangle class:

```python
class Rectangle(Shape):
    def __init__(self, width, length):
        self.width = width
        self.length = length
        super().__init__(width, length)
```

```python
class Rectangle(Shape):
    def __init__(self, *args):
        self.width, self.length = sorted(args)
        super().__init__(*args)
```

Putting conditions on init

```
class Triangle(Shape):

    def __init__(self, *args):
        a, b, c = sorted(args)
        if a + b <= c:
            raise ValueError('Cannot construct triangle with sides given.
        self.sides = (a,b,c)
        super().__init__(*args)

    def area(self):
        semi = sum(self.sides)/2
        a,b,c = self.sides
        return math.sqrt(semi*(semi-a)*(semi-b)*(semi-c))
```

Errors are objects too

You can subclass errors, to get your own, special ones:

Errors are objects too

You can subclass errors, to get your own, special ones:

```
class GeometryError(ValueError):
    pass
```

Errors are objects too

You can subclass errors, to get your own, special ones:

```python
class GeometryError(ValueError):
    pass
```

Let's change the previous error line to use our new error:

```python
raise GeometryError('Cannot construct triangle with sides given.')
```

Your turn...

Make a `Circle` class.

Your turn...

Make a `Circle` class.

This is one possible solution:

```
class Circle(Shape):

    def __init__(self, radius):
        self.radius = radius
        super().__init__(radius)

    def area(self):
        return 3.14 * self.radius**2
```

Equality!

Load shapes.py into the console, and try this:

```
Circle(3) == Circle(3)
```

Equality!

Load shapes.py into the console, and try this:

```
Circle(3) == Circle(3)
```

What happens? Any ideas why?

Equality!

Load shapes.py into the console, and try this:

```
Circle(3) == Circle(3)
```

What happens? Any ideas why?

If python doesn't have any instructions on how to determine equality, it goes by an object's location in memory.

Since these are two different instances, python says they're not equal, even though they're identical.

If we want different behavior, we have to tell it how to determine equality.

Equality!

To tell python how to determine equality, we add another special method, called `__eq__`:

Equality!

To tell python how to determine equality, we add another special method, called `__eq__`:

```
class Shape:
      ⋮
    def __eq__(self, other):
        if self.__class__ != other.__class__:
            return False
        if (self.args == other.args):
            return True
        return False
```

Equality!

To tell python how to determine equality, we add another special method, called `__eq__`:

```
class Shape:
      ⋮
    def __eq__(self, other):
        if self.__class__ != other.__class__:
            return False
        if (self.args == other.args):
            return True
        return False
```

Now try `Circle(3) == Circle(3)` again.

Operator Overloading / More Special Methods

Open `fractions.py` in a text editor.

```
class Fraction:
    def __init__(self, numerator, denominator=1):
        if not all([isinstance(i,int) for i in (numerator,denominator)]):
            raise ValueError('Arguments of Fraction must be integers')
        g = gcd(numerator,denominator)
        self.n = int(numerator/g)
        self.d = int(denominator/g)

    def __neg__(self):
        return Fraction(-self.n,self.d)

    def __abs__(self):
        return Fraction(abs(self.n),abs(self.d))

    def __add__(self, other):
        if not isinstance(other, Fraction):
            other = Fraction(other)
        n = self.n * other.d + self.d * other.n
        d = self.d * other.d
        return Fraction(n, d)

    def __sub__(self, other):
        return self + (-other)

    def __mul__(self, other):
        if not isinstance(other, Fraction):
            other = Fraction(other)
        n1, d1 = self.n, self.d
```

Operator Overloading / More Special Methods

Here are some common operators and the methods to implement on your classes.

operation	method
`obj + other`	`__add__`
`obj - other`	`__sub__`
`obj * other`	`__mul__`
`obj // other`	`__floordiv__`
`obj / other` *	`__div__`
`obj / other` **	`__truediv__`
`obj % other`	`__mod__`
`obj ** other`	`__pow__`

operation	method
`other + obj`	`__radd__`
`other - obj`	`__rsub__`
`other * obj`	`__rmul__`
`other // obj`	`__rfloordiv__`
`other / obj` *	`__rdiv__`
`other / obj` **	`__rtruediv__`
`other % obj`	`__rmod__`
`other ** obj`	`__rpow__`

* only in Python 2
** in Python 3, or in Python 2, with `from __future__ import division`

Operator Overloading / More Special Methods

We implemented `__eq__` on our shapes class, but there are more comparison operators than just that.

operation	method
`obj == other`	`__eq__`
`obj != other`	`__ne__`
`obj < other`	`__lt__`
`obj > other`	`__gt__`
`obj <= other`	`__le__`
`obj >= other`	`__ge__`

Multiple Inheritance

Python allows a class to inherit from multiple classes.

Multiple Inheritance: Method Resolution Order

In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

Multiple Inheritance: Method Resolution Order

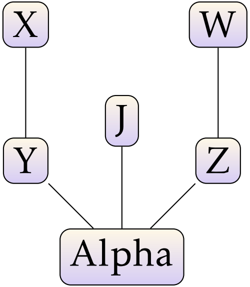
In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

```
class X:
    pass

class Y(X):
    pass

class W:
    pass

class Z(W):
    pass

class J:
    pass

class Alpha(Y,J,Z):
    pass
```

Multiple Inheritance: Method Resolution Order

In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

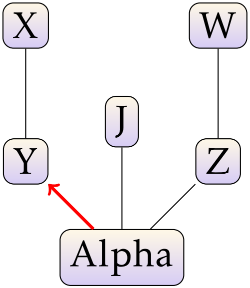
```
class X:
    pass

class Y(X):
    pass

class W:
    pass

class Z(W):
    pass

class J:
    pass

class Alpha(Y,J,Z):
    pass
```



Multiple Inheritance: Method Resolution Order

In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

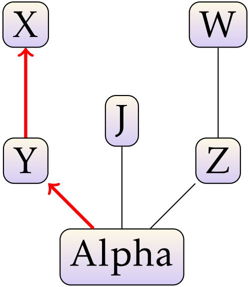
```python
class X:
    pass

class Y(X):
    pass

class W:
    pass

class Z(W):
    pass

class J:
    pass

class Alpha(Y,J,Z):
    pass
```



Multiple Inheritance: Method Resolution Order

In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

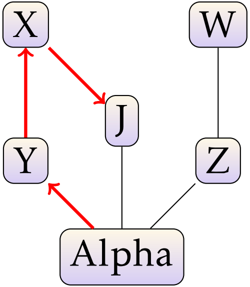
```python
class X:
    pass

class Y(X):
    pass

class W:
    pass

class Z(W):
    pass

class J:
    pass

class Alpha(Y,J,Z):
    pass
```



Multiple Inheritance: Method Resolution Order

In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

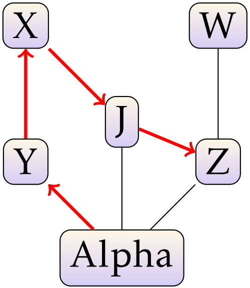
```
class X:
    pass

class Y(X):
    pass

class W:
    pass

class Z(W):
    pass

class J:
    pass

class Alpha(Y,J,Z):
    pass
```



Multiple Inheritance: Method Resolution Order

In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

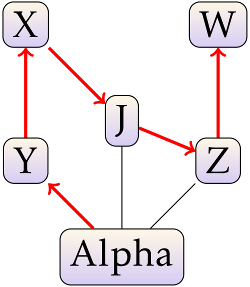
```
class X:
    pass

class Y(X):
    pass

class W:
    pass

class Z(W):
    pass

class J:
    pass

class Alpha(Y,J,Z):
    pass
```



Multiple Inheritance: Method Resolution Order

In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

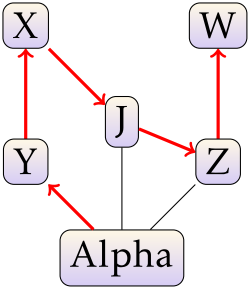
```
class X:
    pass

class Y(X):
    pass

class W:
    pass

class Z(W):
    pass

class J:
    pass

class Alpha(Y,J,Z):
    pass
```



Multiple Inheritance: Method Resolution Order

In uncomplicated cases, when Python is looking for a method or attribute, it checks the parent classes from left to right, depth first.

```python
class X:
    pass

class Y(X):
    pass

class W:
    pass

class Z(W):
    pass

class J:
    pass

class Alpha(Y,J,Z):
    pass
```




```
>>> Alpha.mro()
[<class 'multi.Alpha'>, <class 'multi.Y'>,
<class 'multi.X'>, <class 'multi.J'>,
<class 'multi.Z'>, <class 'multi.W'>, <class 'object'>]
```

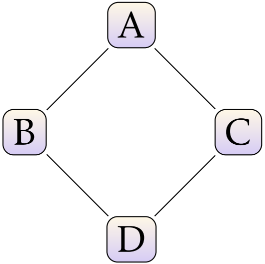
Multiple Inheritance: Method Resolution Order

MROs get more complicated if the diagram of classes contains cycles.

Multiple Inheritance: Method Resolution Order

MROs get more complicated if the diagram of classes contains cycles.

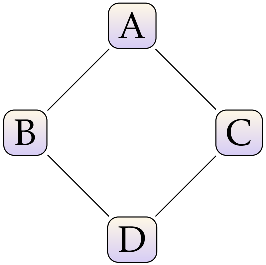
```python
class A:
    #stuff

class B(A):
    #stuff

class C(A):
    #stuff

class D(B,C):
    #stuff
```



Multiple Inheritance: Method Resolution Order

MROs get more complicated if the diagram of classes contains cycles.

```python
class A:
    #stuff

class B(A):
    #stuff

class C(A):
    #stuff

class D(B,C):
    #stuff
```

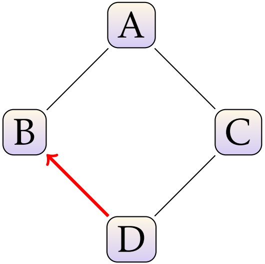



```
>>> D.mro()
[<class 'multi.D'>, <class 'multi.B'>,
<class 'multi.C'>, <class 'multi.A'>,
<class 'object'>]
```

Multiple Inheritance: Method Resolution Order

MROs get more complicated if the diagram of classes contains cycles.

```
class A:
    #stuff

class B(A):
    #stuff

class C(A):
    #stuff

class D(B,C):
    #stuff
```

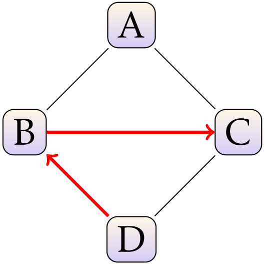



```
>>> D.mro()
[<class 'multi.D'>, <class 'multi.B'>,
<class 'multi.C'>, <class 'multi.A'>,
<class 'object'>]
```

Multiple Inheritance: Method Resolution Order

MROs get more complicated if the diagram of classes contains cycles.

```python
class A:
    #stuff

class B(A):
    #stuff

class C(A):
    #stuff

class D(B,C):
    #stuff
```

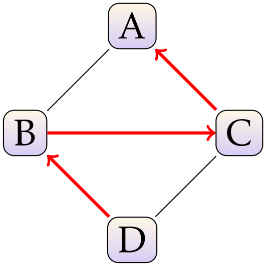



```
>>> D.mro()
[<class 'multi.D'>, <class 'multi.B'>,
<class 'multi.C'>, <class 'multi.A'>,
<class 'object'>]
```

Multiple Inheritance: Method Resolution Order

MROs get more complicated if the diagram of classes contains cycles.

```python
class A:
    #stuff

class B(A):
    #stuff

class C(A):
    #stuff

class D(B,C):
    #stuff
```



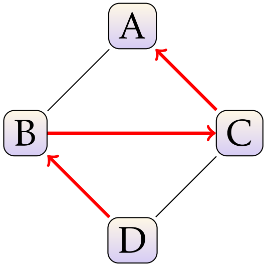

```
>>> D.mro()
[<class 'multi.D'>, <class 'multi.B'>,
<class 'multi.C'>, <class 'multi.A'>,
<class 'object'>]
```

Multiple Inheritance: Method Resolution Order

Now let's look at what happens when we actually call some methods from a class that uses multiple inheritance.

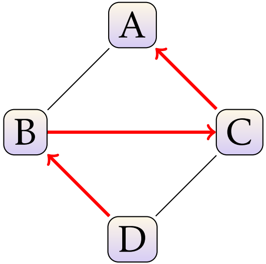
Multiple Inheritance: Method Resolution Order

```
class A:
    def foo(self):
        print('foo from A')

class B(A):
    def foo(self):
        print('foo from B')
        super().foo()
    def bar(self):
        print('bar from B')

class C(A):
    def foo(self):
        print('foo from C')
        super().foo()
    def bar(self):
        print('bar from C')

class D(B,C):
    def foo(self):
        print('foo from D')
        super().foo()
```



Multiple Inheritance: Method Resolution Order

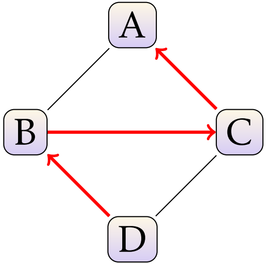
```
class A:
    def foo(self):
        print('foo from A')

class B(A):
    def foo(self):
        print('foo from B')
        super().foo()
    def bar(self):
        print('bar from B')

class C(A):
    def foo(self):
        print('foo from C')
        super().foo()
    def bar(self):
        print('bar from C')

class D(B,C):
    def foo(self):
        print('foo from D')
        super().foo()
```




```
d = D()
d.foo()
```

Multiple Inheritance: Method Resolution Order

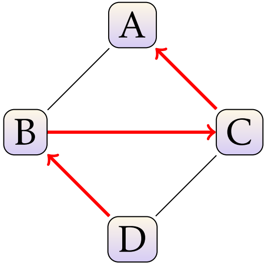
```
class A:
    def foo(self):
        print('foo from A')

class B(A):
    def foo(self):
        print('foo from B')
        super().foo()
    def bar(self):
        print('bar from B')

class C(A):
    def foo(self):
        print('foo from C')
        super().foo()
    def bar(self):
        print('bar from C')

class D(B,C):
    def foo(self):
        print('foo from D')
        super().foo()
```




```
d = D()
d.foo()
d.bar()
```

Multiple Inheritance: Method Resolution Order

```python
class A:
    def foo(self):
        print('foo from A')

class B(A):
    def foo(self):
        print('foo from B')
        super().foo()
    def bar(self):
        print('bar from B')

class C(A):
    def foo(self):
        print('foo from C')
        super().foo()
    def bar(self):
        print('bar from C')

class D(B,C):
    def foo(self):
        print('foo from D')
        super().foo()
```

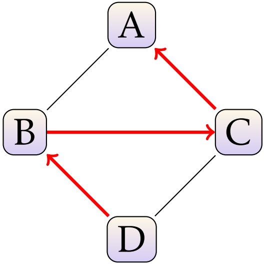



```python
d = D()
d.foo()
d.bar()
```

```python
b = B()
b.foo()
```

Multiple Inheritance: Method Resolution Order

```python
class A:
    def foo(self):
        print('foo from A')

class B(A):
    def foo(self):
        print('foo from B')
        super().foo()
    def bar(self):
        print('bar from B')

class C(A):
    def foo(self):
        print('foo from C')
        super().foo()
    def bar(self):
        print('bar from C')

class D(B,C):
    def foo(self):
        print('foo from D')
        super().foo()
```

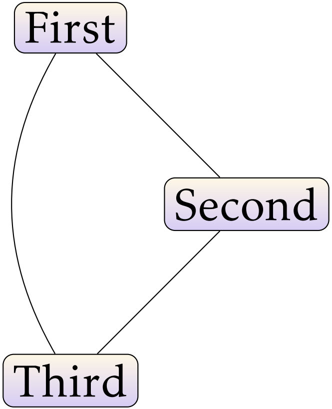



```python
d = D()
d.foo()
d.bar()
```

```python
b = B()
b.foo()
b.bar()
```

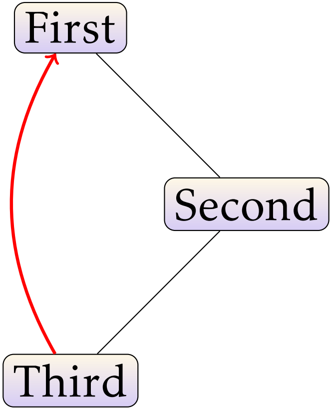
Multiple Inheritance: My Weirdest Example

```
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



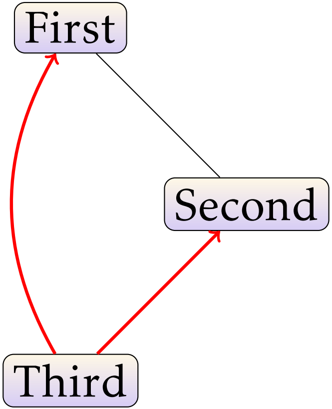
Multiple Inheritance: My Weirdest Example

```python
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



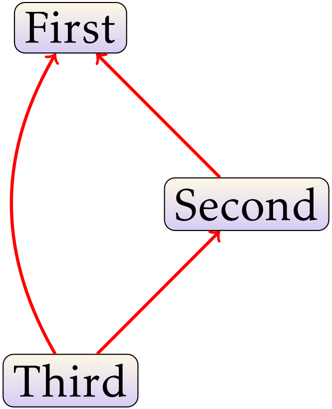
Multiple Inheritance: My Weirdest Example

```python
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



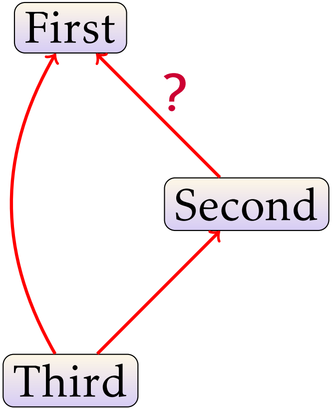
Multiple Inheritance: My Weirdest Example

```
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



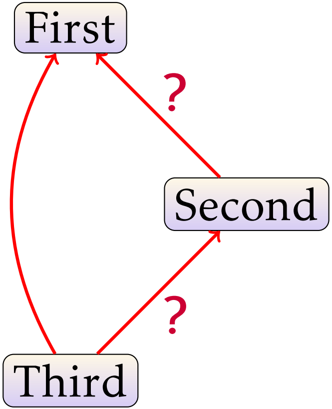
Multiple Inheritance: My Weirdest Example

```python
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



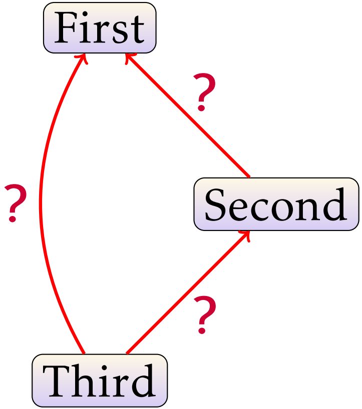
Multiple Inheritance: My Weirdest Example

```python
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



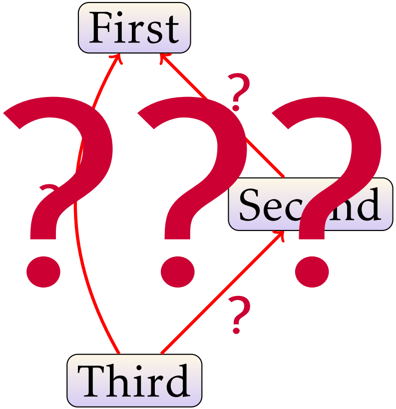
Multiple Inheritance: My Weirdest Example

```python
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



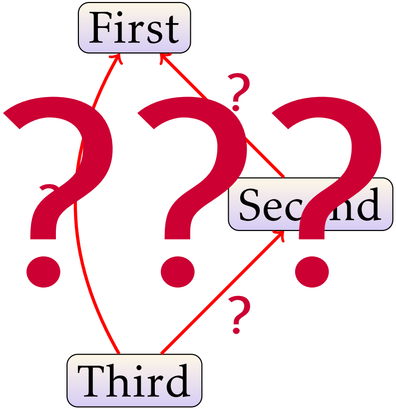
Multiple Inheritance: My Weirdest Example

```python
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



Multiple Inheritance: My Weirdest Example

```python
class First:
    pass

class Second(First):
    pass

class Third(First, Second):
    pass
```



Let's try running this code, and see what happens.

Multiple Inheritance: Important Points

Python's multiple inheritance allows you to create a class that inherits from more than one class. It's not as hard as some claim, but you do need to be aware of a few things:

Multiple Inheritance: Important Points

Python's multiple inheritance allows you to create a class that inherits from more than one class. It's not as hard as some claim, but you do need to be aware of a few things:

- Python determines where to look for methods and attributes by the Method Resolution Order. This also determines what method gets called by `super()`

Multiple Inheritance: Important Points

Python's multiple inheritance allows you to create a class that inherits from more than one class. It's not as hard as some claim, but you do need to be aware of a few things:

- Python determines where to look for methods and attributes by the Method Resolution Order. This also determines what method gets called by `super()`
- The approach to the Method Resolution Order, is left-to-right, depth-first, guarantees that each class only appears once, and that parent classes appear after all classes that inherit from them.

Multiple Inheritance: Important Points

Python's multiple inheritance allows you to create a class that inherits from more than one class. It's not as hard as some claim, but you do need to be aware of a few things:

- Python determines where to look for methods and attributes by the Method Resolution Order. This also determines what method gets called by `super()`
- The approach to the Method Resolution Order, is left-to-right, depth-first, guarantees that each class only appears once, and that parent classes appear after all classes that inherit from them.
- A common pattern is to have a base class that you want to modify with small modifier classes Because the overall approach to the Method Resolution Order is left-to-right, you should **not** use `class MyClass(BaseClass, Modifier1, Modifier2)`.

  Instead, use `class MyClass(Modifier2, Modifier1, BaseClass):`.

Back to the shapes

We're going to work with the shapes.py code again, so please copy the slightly updated version from pydancing.wordpress.com/oop

Mixin Classes

A **mixin** class is a class meant to be inherited together with a “true” base class.

Usually mixins provide a feature that you might want to add to several classes.

Mixin: Colors With Our Shapes

Let's create a mixin class that allows us to create colored versions of the shapes we have in shapes.py.

```
class ColorMixin:
    is_colored = True

    def __init__(self, *args, **kwargs):
        color = kwargs.get('color', None)
        if color is None:
            color = random.choice(COLORS)
            kwargs['color'] = color
        self.color = color
        super().__init__(*args, **kwargs)
```

Mixin: Colors With Our Shapes

Let's create a mixin class that allows us to create colored versions of the shapes we have in shapes.py.

```
class ColorMixin:
    is_colored = True

    def __init__(self, *args, **kwargs):
        color = kwargs.get('color', None)
        if color is None:
            color = random.choice(COLORS)
            kwargs['color'] = color
        self.color = color
        super().__init__(*args, **kwargs)
```

Now, what should we do to create a `ColoredCircle` class?

Mixin: Inheriting Color

For each shape that we want a colored version of, we need to create a class that inherits from our `ColorMixin` and the original shape.

```
class ColoredCircle(ColorMixin, Circle):
    pass
```

Since all the work is done in the parent classes, we don't need much code here. In real-world code, it is best to include a docstring and perhaps some doctests.

Mixin: Inheriting Color

For each shape that we want a colored version of, we need to create a class that inherits from our `ColorMixin` and the original shape.

```
class ColoredCircle(ColorMixin, Circle):
    pass
```

Since all the work is done in the parent classes, we don't need much code here. In real-world code, it is best to include a docstring and perhaps some doctests.

Then to instantiate our new class:

```
y = ColoredCircle(5, color='Yellow')
r = ColoredCircle(11)
```

Pythonic Protocols

Python doesn't have interfaces the way Java and many other staticly-typed languages do.

Because traditionally Python relies heavily on **duck typing**, for your object to be treated like one of a category of objects, all you have to is implement the right methods. This is sometimes loosely called a Protocol.

Protocols

To create a container object, implement:

- `__len__`
- `__getitem__`

Protocols

To create a container object, implement:

- `__len__`
- `__getitem__`
- `__setitem__` (if you want a mutable container)
- `__delitem__` (if you want a mutable container)

Protocols

To create a container object, implement:

- `__len__`
- `__getitem__`
- `__setitem__` (if you want a mutable container)
- `__delitem__` (if you want a mutable container)
- `__iter__` (if you want your container to be iterable)

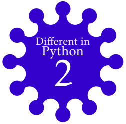
Protocols

To create a container object, implement:

- `__len__`
- `__getitem__`
- `__setitem__` (if you want a mutable container)
- `__delitem__` (if you want a mutable container)
- `__iter__` (if you want your container to be iterable)

To create an iterator, either write a generator (a function that uses the `yield` statement to return a sequence of values) or write a new class that implements:

- `__iter__`
- `__next__`



Protocols

To create a context manager object that works in a `with` statement:

- `__enter__`
- `__exit__`

```
with Thing(foo) as bar:
    bar.blah()
    # do stuff with bar in the indented block
    # and when the block ends, perform whatever
    # clean-up actions are necessary.
```

Protocols

To create a context manager object that works in a `with` statement:

- `__enter__`
- `__exit__`

```python
with Thing(foo) as bar:
    bar.blah()
    # do stuff with bar in the indented block
    # and when the block ends, perform whatever
    # clean-up actions are necessary.
```

To create an object that can be called like a function, implement:

- `__call__`

Subclassing Built-In Types

Although duck typing is the "Pythonic" way, if your object needs to interact with code that will do `isinstance` checks, you should subclass the type that you need it to be. You can then change the methods however you want.

The End.

Frequently Asked Question

by people who have done OOP in other languages

How do I make attributes of an object private or protected?

Frequently Asked Question

by people who have done OOP in other languages

How do I make attributes of an object private or protected?

This is somewhat discouraged in Python. In general, the Pythonic mindset is to document your code well, follow conventions, and trust that other programmers won't do stupid things unless they have a really good reason.

Frequently Asked Question

by people who have done OOP in other languages

How do I make attributes of an object private or protected?

This is somewhat discouraged in Python. In general, the Pythonic mindset is to document your code well, follow conventions, and trust that other programmers won't do stupid things unless they have a really good reason.

A convention is that names that begin with one underscore shouldn't be accessed directly. If you do this, you should provide another way to access the attribute.

Protection for Attributes

Let's change our `Circle` class to follow this convention:

```
class Circle(Shape):

    def __init__(self, radius, **kwargs):
        self._radius = radius
        super().__init__(radius, **kwargs)

    @property
    def radius(self):
        return self._radius

    def area(self, dp=2):
        return round(math.pi * self._radius**2, dp)
```

That does not prevent someone else from using or changing the attribute.
`mycircle._radius = 0` will work.

Stronger Protection for Attributes

There is a stronger form of protection you can use:

```python
class Circle(Shape):
    def __init__(self, radius, **kwargs):
        self.__radius = radius
        super().__init__(radius, **kwargs)

    @property
    def radius(self):
        return self.__radius

    @radius.setter # circle.radius = <something>  will call this
    def radius(self, name):
        raise AttributeError("Don't do that!")

    def area(self, dp=2):
        return round(math.pi * self.__radius**2, dp)
```

Stronger Protection for Attributes

Behind the scenes, what Python does in this case is called **name-mangling**.

It stores `self.__radius` as `self._Circle__radius`.

Stronger Protection for Attributes

Behind the scenes, what Python does in this case is called **name-mangling**.

It stores `self.__radius` as `self._Circle__radius`.

The good news is `mycircle._radius = 0` will result in an error message. The bad news is, if someone is determmined to break things, they can do `mycircle._Circle__radius = 0`.

The End.

(I'm serious this time.)