# Lesson 11 Inheritance

September 8, 2020

## 1 Agenda

- Q&A
- Inheritance Recap
- Multiple inheritance and method resolution order
- Using Inheritance

## 2 References

- Python Documentation - Inheritance
- Method Resolution Order

## 3 Inheritance Recap

- Inheritance is a concept associated with object-oriented programming. By leveraging inheritance we can create a hierarchical structure where the child classes acquire all the properties and behaviours of the parent classes
- Code reusability: We can build new classes based on existent ones and extend, overwrite or reuse the parent functionality
- Multiple inheritance: Ability to inherit from multiple parent classes

### 3.1 Exercise:

---

```
We will build this exercise in class and add the solution in the course material
```

---

We started a company that builds Polygons.

Our first two products are:

- triangles

- squares

Requirements:

1. Be able to create Polygons with n sides ( 3 sides -> triangle, 4 -> sides square, ..., n sides)
2. Display the number of sides of the Polygon when we print any Polygon object
3. Display the values of each side for all Polygon objects
    - Side 1 with lenght: 2
    - Side 2 with lenght: 4
    - Side 3 with length: 6
4. Compute area for Triangles and Squares

Definitions:

1. A polygon is a geometrycal figure with 3 or more sides
2. Triangle area = `s(s-a)(s-b)(s-c) ** 0.5` where s = `(a+b+c) / 2` and a,b,b are the lenght of the sides
3. Square area = `width * length`

Questions:

1. How many classes do we need?
2. How the classes hierarchy will look like?
3. How many methods do we need for each class and what are those?

[25]:
```
# %load polygon.py
class Polygons(object):

    def __init__(self, *args):
        self.sides = args

    def __str__(self):
        no_of_sides = len(self.sides)
        return '{} is the number of sides'.format(no_of_sides)

    def display(self):
        for side_index, length in enumerate(self.sides, start=1):
            print('Side {} with length: {}'.format(side_index, length))


class Triangle(Polygons):
    # `s(s-a)(s-b)(s-c) ** 0.5` where s = `(a+b+c) / 2`
    def __init__(self, *args):
        super().__init__(*args)

    def area(self):
        s1, s2, s3 = self.sides
        s_p = sum(self.sides) / 2
```

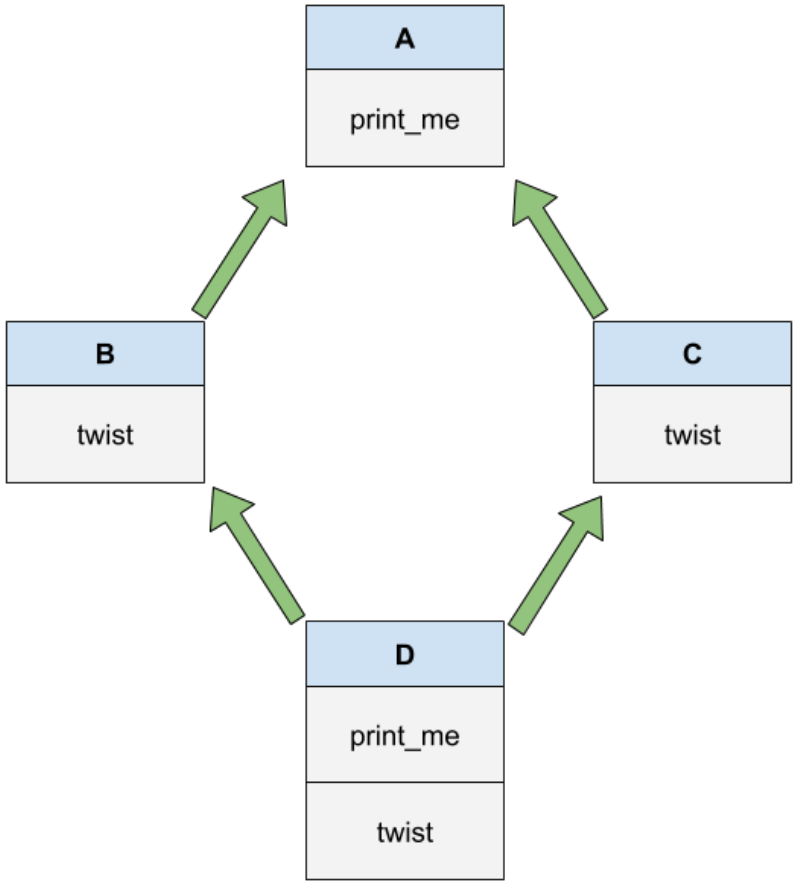
```
        return (s_p*(s_p - s1)*(s_p - s2)*(s_p - s3))** 0.5


class Square(Polygons):

    def __init__(self, *args):
        super().__init__(*args)

    def area(self):
        side, *_ = self.sides
        return side ** 2
```

## 4 Multiple inheritance and method resolution order

Python supports multiple inheritance. With multiple inheritance, we need to make sure we have a mechanism to deal with methods naming conflicts. Because our class can have multiple parents there are cases when two parent classes use the same naming for one or multiple methods.

To deal with this Python uses `Method Resolution Order (mro)` to determine on one side on how the inheritance will work in case of name conflict and on the other side in which order `super()` will call methods. The classic challenge of multiple inheritance is the 'diamond problem' shown in the diagram below where we have multiple paths to the parent class.

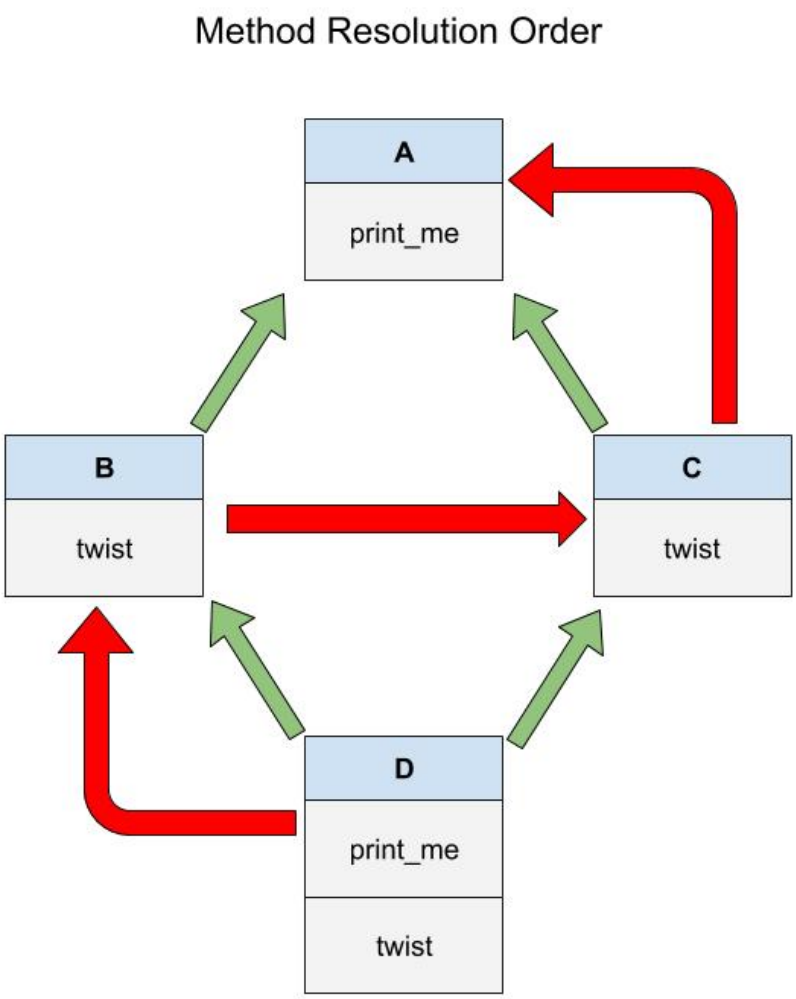
We can check the `mro` by accessing the `__mro__` attribute of class objects. This attribute contains a tuple of references to all parent classes all the way to the `object` class




[26]:
```
# %load multiple_inheritance.py
class A:

    def print_me(self):
        print('A: me', self)


class B(A):

    def twist(self):
        print('B: twist', self)


class C(A):

    def twist(self):
        print('C: new twist', self)
```

```
class D(B, C):

    def print_me(self):
        super().print_me()
        print('D: new me', self)

    def print_me_and_twist(self):
        self.print_me()
        super().print_me()
        self.twist()
        super().twist()
        C.twist(self)
        print('me and twist')


# Check method resolution order
print(D.__mro__)

d = D()
d.print_me_and_twist()
```

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>)
A: me <__main__.D object at 0x7fcc4c203710>
D: new me <__main__.D object at 0x7fcc4c203710>
A: me <__main__.D object at 0x7fcc4c203710>
B: twist <__main__.D object at 0x7fcc4c203710>
B: twist <__main__.D object at 0x7fcc4c203710>
C: new twist <__main__.D object at 0x7fcc4c203710>
me and twist
```




Let's see what happened when we call `d.print_me_and_twist()`

1. self.print_me() -> print_me() method is deffined in D class and will be used.
2. super().print_me() -> we want to call `print_me()` from the perent class. The first parent with `print_me()` method is `A`
3. self.twist() -> method not deffined in `D`. The first parent with `twist` method is `B`
4. super(B, self).twist() -> we know that B is the first parent with `twist` but we need the functionality of the `twist` method implemented by `C`. We tell super to start the search on `mro` from `B` not `D`.
5. C.twist(self) -> we bypass the mro by explicitly calling `twist` on `C` and passing `self` as an argument. We use `twist` as an unbound method that is why we need to pass `self`
6. print `me and twist`

What we did at step `5.` needs to be avoided. To cover that use case we can use super, just like we did at step `4.`

[27]:
```
# %load mro_examples.py
import numbers
import io

print(bool.__mro__)

print('-------------------------------------------')
print(int.__mro__)

print('-------------------------------------------')
print(numbers.Integral.__mro__)

print('-------------------------------------------')
print(io.BytesIO.__mro__)
```

```
(<class 'bool'>, <class 'int'>, <class 'object'>)
-------------------------------------------
(<class 'int'>, <class 'object'>)
-------------------------------------------
(<class 'numbers.Integral'>, <class 'numbers.Rational'>, <class 'numbers.Real'>,
<class 'numbers.Complex'>, <class 'numbers.Number'>, <class 'object'>)
-------------------------------------------
(<class '_io.BytesIO'>, <class '_io._BufferedIOBase'>, <class '_io._IOBase'>,
<class 'object'>)
```

Python `mro` is using an algorithm called C3 to determine the inheritance order you can see above. Detail explanation of the algorithm can be found here: Method Resolution Order

## 5 Using Inheritance

Bellow, I will list a couple of examples where we can put inheritance to good use and avoid making complicated and hard to work with hierarchies

**Interface inheritance and implementation inheritance** It is in our advantage to make a distinction between interface inheritance and implementation inheritance.

Interface inheritance is used for creating a sub-type, just like what we did in previous courses where we inherited from python collections. By doing so our object will have a collection interface that Python knows how to work to in for loops for example.

Implementation inheritance is when we inherit the functionality to avoid code duplication. This is the case of mixins

[28]:
```
# %load interface_inheritance.py
class BaseUser:

    def create_profile(self, **kwargs):
```

```
        raise NotImplementedError('All subclasses of user need to implement␣
↪this method')

    def save_profile(self, **kwargs):
        raise NotImplementedError('All subclasses of user need to implement␣
↪this method')

    def bio(self, **kwargs):
        raise NotImplementedError('All subclasses of user need to implement␣
↪this method')


class BioPlaceholderMixin:

    def bio(self):
        print('This is a placeholder bio')


class User(BaseUser):

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name


class Student(BioPlaceholderMixin, User):

    def __init__(self, first_name, last_name):
        super().__init__(first_name, last_name)


user = User('George', 'D')
student = Student('George', 'D')
```

In the above example, we have a `BaseUser` class that describes the standard we have in our system for creating users. As you can see there is no functionality in the `BaseUser` class just how we expect all the subclasses to behave. All the subclasses need to have `create_profile`, `save_profile` and `bio` methods. If we want to call anyone of those it will result in an exception.

We have a special case where we have to create Student users that don't have a bio. To fix this we can use a mixin. It is always recommended to use the `Mixin` suffix when creating a mixin class, by doing so it is more explicit and it tells the users of the class how to work with it. It is important to mention that mixin classes are not intended to be used alone nor instantiated.

---

Question: Why we place the mixin class first? how the `mro` of class Student looked before adding the mixin?

---

## 6 Challenge:

We have new customers for our Polygons company.

- They need to create square objects with a certain area
- They need a method to compute the perimeter only for triangle objects

Requirements: * add an alternative constructor to Square class that takes the area as an argument and creates a square object with the apropriate sides

Example :

sq = Square.from_area(8)

print(sq)

Side 1 with lenght: 2

Side 2 with lenght: 4

Side 3 with lenght: 2

Side 4 with lenght: 4

- we want to make our perimeter method also available to other shapes in the future. Create a mixin class for perimeter that contains the perimeter method