

A Duck Tour Of Object-Oriented Programming

Terry Yin

October 31, 2014



Welcome on board our programming duck tour! But, wait a minute, just what is a duck?


1 The Object-Oriented Tour

1.1 Class and method

Well, duck is a general kind of bird. In object-oriented programming term, `Duck` is a **class**.


Python is a programming language that supports multiple **paradigms**, like structured programming and object-oriented programming, [Wikipedia \(2014b\)](#). I will use Python 2.7 as the introduction language here.

This is how you define a class in Python:

```
  
class Duck:  
    pass
```

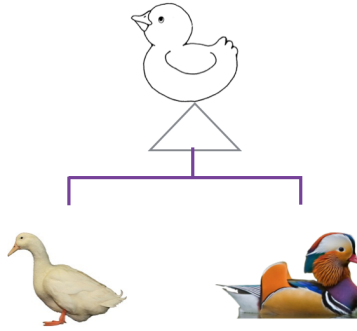
The keyword `pass` here means to skip the content definition of the class.

What does a duck do? It **quacks** and **walks**. So, let's define these behaviours of the duck class:


```
  
class Duck:  
    def quack(self):  
        print "quack"  
    def walk(self):  
        print "walk"
```

1.2 Inheritance

But `Duck` is still too general. There are more specific types of ducks. For example, `DomesticDuck` is a type of duck. `MandarinDuck` is another type of duck. The “**is a**” relationship in OO programming is called **inheritance**.




In Python:

```
  
class DomesticDuck(Duck):  
    pass  
  
class MandarinDuck(Duck):  
    pass
```

Then we can say A DomesticDuck **is a** Duck, because DomesticDuck is **derived** from the Duck class. This is especially important for **strong typed** language, e.g. Java & C++. Because in strong typed language, only the instance of the Duck class or its derived classes are considered duck. Not matter how similar it is to a duck, as long as it doesn't have the class Duck in its inheritance hierarchy, it's not a duck. However in a **dynamic programming language**, like Python, as long as the object has the **quack** method and **walk** method that we need, we can consider it a duck without needing inheritance. Or "if it walks like a duck and quacks like a duck, it is a duck." This is called **duck typing**, [Wikipedia \(2014a\)](#).


1.3 Object

So an instance of a class could have other values as its member. We just used the term **instance**. It also has another name, **object**. A domestic duck is an instance of the DomesticDuck class. Let's create two DomesticDuck instances:

```
  
jimmy = DomesticDuck()  
tommy = DomesticDuck()
```


1.4 Polymorphism

Although we didn't define method **quack** for the class DomesticDuck, it inherits the method from its **parent class** (another name for derived class) Duck. We can also override the method in a **subclass**. Let's say the DomesticDucks are noisier than other ducks:



```
class DomesticDuck(Duck):
    def quack(self):
        print "quack-quack-quack"
```

Now the DomesticDuck shares the same **interface** quack as the more generic Duck, but has different behaviours. This is called **polymorphism**.




```
jimmy = DomesticDuck()
romeo = MandarinDuck()
jimmy.quack()
romeo.quack()
```

```
quack-quack-quack
quack
```


1.5 Composition

A domestic duck unlike the wild ducks, they have owners. So a DomesticDuck **has an** owner. In Python, that could be expressed by:



```
class DomesticDuck(Duck):
    def __init__(self):
        self.owner = "the nature"
```


Then we can access the **method** and **property** we defined for the class:



```
jimmy.owner = "Terry"
print "Jimmy duck belongs to", jimmy.owner
```

```
Jimmy duck belongs to Terry
```

The **owner** information is a field composes the DomesticDuck instances. For this kind of **has a** relationship, we call it **composition**. There's another type of composition. Considering ducks could have 0 to n children.




```
class DomesticDuck(Duck):
    def __init__(self):
        self.owner = "the nature"
        self.children = [] # a list of 0 to n other ducks
```

The children ducks are not information that composes a (mother) duck, but only related to the (mother) duck. This is also a **has a** (or, has many) relationship. We call it **aggregation**.

1.6 Encapsulation

Children ducks follow mother duck. So the behaviour of a duck could be:


```

class DomesticDuck(Duck):
    def __init__(self):
        self.owner = "the nature"
        self.children = [] # a list of 0 to n other ducks

    def walk(self):
        for child in self.children:
            child.walk()
```


Now when you call the `move` method on a mother duck, `motherDuck.move()` all its children, if there is any, will also move. We group the children information and the operation on the information together in the `DomesticDuck` class. This is called **Encapsulation**.

1.7 Information hiding

I think that's enough about duck. Let's talk about the duck tour. A “duck tour” is a sightseeing tour on a special vehicle that goes on both land and water, which can be found in cities like Singapore and San Francisco. But let's make a simple version of “duck tour”, which is the tour of a happy duck.

```

def duck_tour(duck):
    duck.quack()
    duck.walk()
    duck.quack()
    duck.quack()
    duck.walk()
    duck.quack()
    duck.quack()
    duck.quack()
    duck.quack()
    duck.walk()
```

In the `duck_tour` function, it takes one parameter `duck`. What duck is it? This function doesn't care. It only “talks” to the duck using the interface `quack` and `walk`. The detailed implementation of `quack` and `walk` is hiding from function `duck_tour`. This is called **information hiding**, Colburn & Shute (2007).


```

> duck_tour(jimmy)
```

```
quack quack quack
walk
quack quack quack
quack quack quack
walk
quack quack quack
quack quack quack
quack quack quack
```

walk

2 Structured programming

The previous `duck_tour` function doesn't belong to any class or object. It's a set of statements that is set aside from the states it operates. We call this the **structured programming**. Our function could be improved to have better structure by introducing a loop:

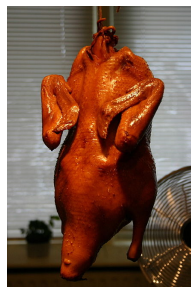
```
def duck_tour(duck):  
    for i in range(1, 4):  
        for _ in range(i):  
            duck.quack()  
        duck.walk()
```

3 The duck tour story is just an analogy!

You might like the duck tour story and examples because it helped you to understand the object-oriented programming concepts (hopefully). But it's important to remember: **The duck story is not OO programming. It's only an analogy of OO programming.** OO programming is NOT to model the real world. It's a programming paradigm that lowers the coupling in the program by creating cohesive objects. So it rather depends on how the information inside a program related to each other and communicate with each other. It's just sometimes coincidentally similar to the concepts we see in a real world.

4 Conclusion

I couldn't find any evidence that the OO programming really improved anything overall, comparing to structured programming. It is true that I know quite many good OO design principles and patterns that could make very good OO design. But I might be able to make the similar quality and maintainable design with structured programming style as well. And what really happened is program written in traditional OO language like C++ is sometimes even harder to be maintained than in simple structured programming language like C. Because traditional OO language is usually harder to master than structured language. A Peking Duck is also called a duck, but it doesn't quack and it doesn't walk. It tastes good though. This will confuse a lot of so-called OO programmers.



Does that mean we should abandon OO programming? I don't think that's going to happen. My personal opinion is, if you are using an OO language, you need to do proper OO design. Otherwise, things only get worse. If you are using a more flexible language, As Kevlin Henney said, the programming **paradigms** are

probably not that important. We may have a hybrid of all different **styles**, Henney (n.d.). We can borrow things from any paradigm as long as it's good.

This paper is created using IPython Notebook, Pérez & Granger (2007).

References

- Colburn, T. & Shute, G. (2007), 'Abstraction in computer science', *Minds and Machines* **17**(2), 169–184.
- Henney, K. (n.d.), 'C++ stylistics', <https://www.youtube.com/watch?gl=SG&hl=en-GB&v=zh8W4Zgl0lw>. Accessed: 2014-10-30.
- Pérez, F. & Granger, B. E. (2007), 'IPython: a System for Interactive Scientific Computing', *Computing in Science & Engineering* **9**(3), 21–29. URL: <http://ipython.org>.
- Wikipedia (2014a), 'Duck typing — wikipedia, the free encyclopedia'. [Online; accessed 31-October-2014].
URL: http://en.wikipedia.org/w/index.php?title=Duck_typing&oldid=631664731
- Wikipedia (2014b), 'Python (programming language) — wikipedia, the free encyclopedia'. [Online; accessed 31-October-2014].
URL: [http://en.wikipedia.org/w/index.php?title=Python_\(programming_language\)&oldid=631855502](http://en.wikipedia.org/w/index.php?title=Python_(programming_language)&oldid=631855502)