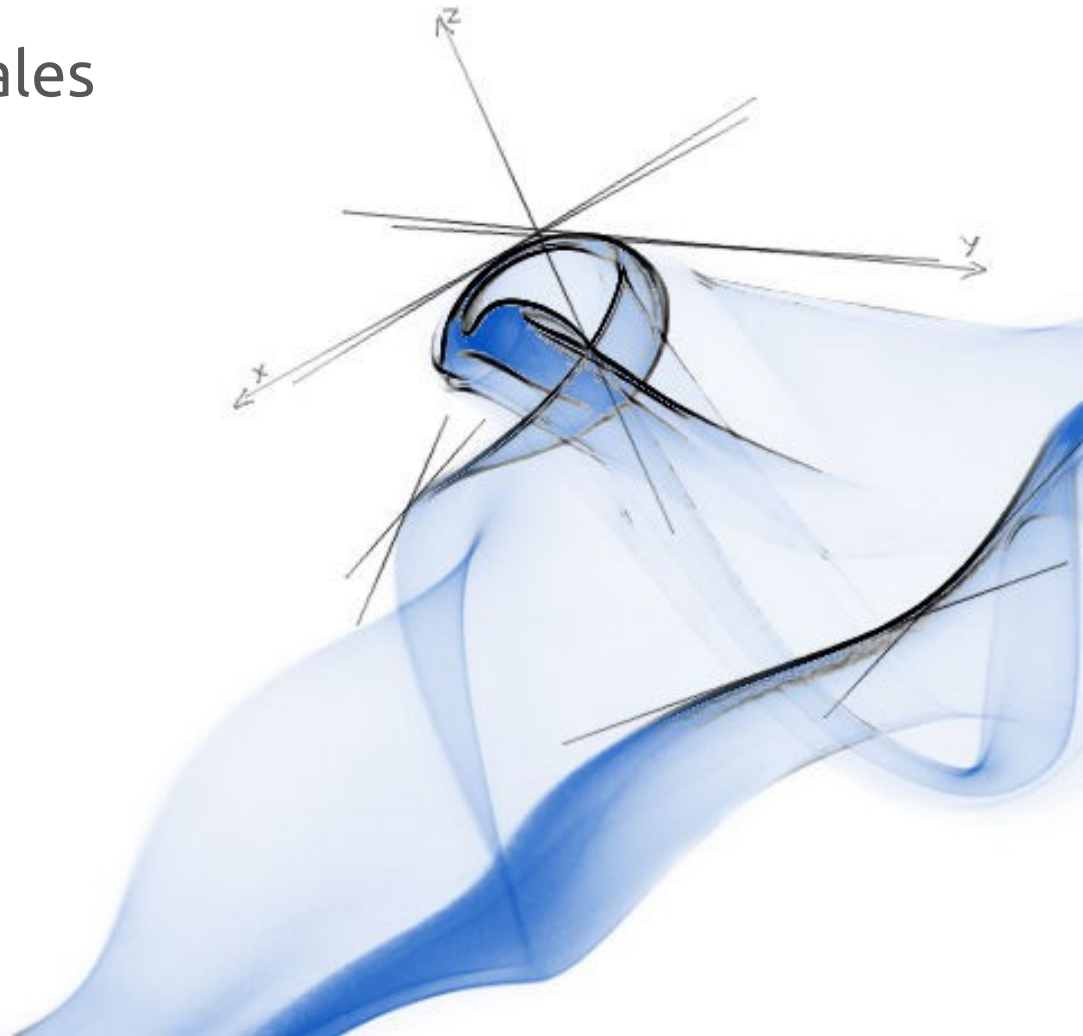


Dive into Object-oriented Python

DjangoCon Europe 2015 – Cardiff, Wales



Leonardo Giordani
lgiordani.com



<https://github.com/lgiordani/oopy>

IPython Notebooks
DjangoCon_Europe_2015/notebooks

Python 3 code
DjangoCon_Europe_2015/code/python3

Python 2 code
DjangoCon_Europe_2015/notebooks

I'm a software engineer, interested in operating systems, versioning, Python and software architecture.

Coder since around 1988, Linux user since 1998, Python lover since 1999.

Currently working with Python and C in the field of satellite remote sensing.

about me



The Digital Cat
lgiordani.com



https://twitter.com/tw_lgiordani



<https://github.com/lgiordani>



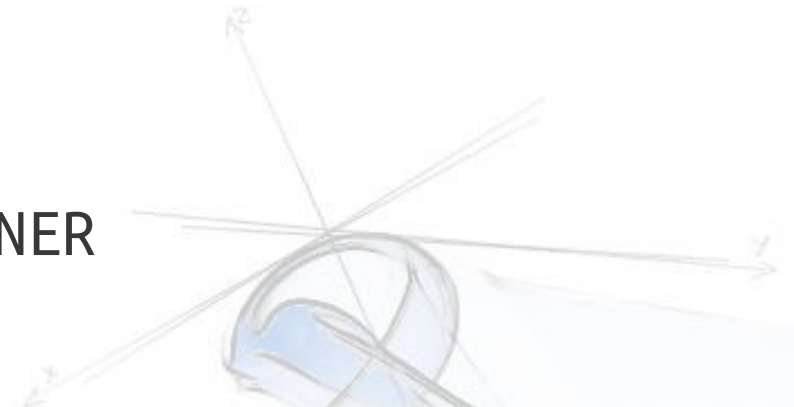
<https://plus.google.com/u/LeonardoGiordani>

about_you.py

```
import levels
from djce2015 import oop_workshop

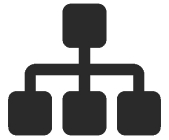
def can_understand(user):
    return user.level >= levels.BEGINNER

for you in oop_workshop.attendees:
    assert can_understand(you)
```





Objects and types



Classes and members



Delegation



Polymorphism

python³



powered

ABOUT THIS TUTORIAL

PART 1

Objects and types



```
data = (13, 63, 5, 378, 58, 40)
```

```
def avg(d):  
    return sum(d)/len(d)
```

```
>>> avg(data)  
92.83333333333333
```

Procedures can modify data

```
door1 = [1, 'closed']  
door2 = [2, 'closed']
```

```
def open_door(door):  
    door[1] = 'open'
```

```
>>> open_door(door1)  
>>> door1  
[1, 'open']
```

Things can get complicated

```
door1 = [1, 'closed']  
door2 = [2, 'closed']
```

```
ldoor1 = [1, 'closed', 'unlocked']
```

```
def open_door(door):  
    door[1] = 'open'
```

```
def open_ldoor(door):  
    if door[2] == 'unlocked':  
        door[1] = 'open'
```

Things can get complicated

```
door1 = [1, 'closed']  
door2 = [2, 'closed']
```

```
ldoor1 = [1, 'closed', 'unlocked']
```

```
def open_door(door):  
    door[1] = 'open'
```

```
def open_ldoor(door):  
    if door[2] == 'unlocked':  
        door[1] = 'open'
```

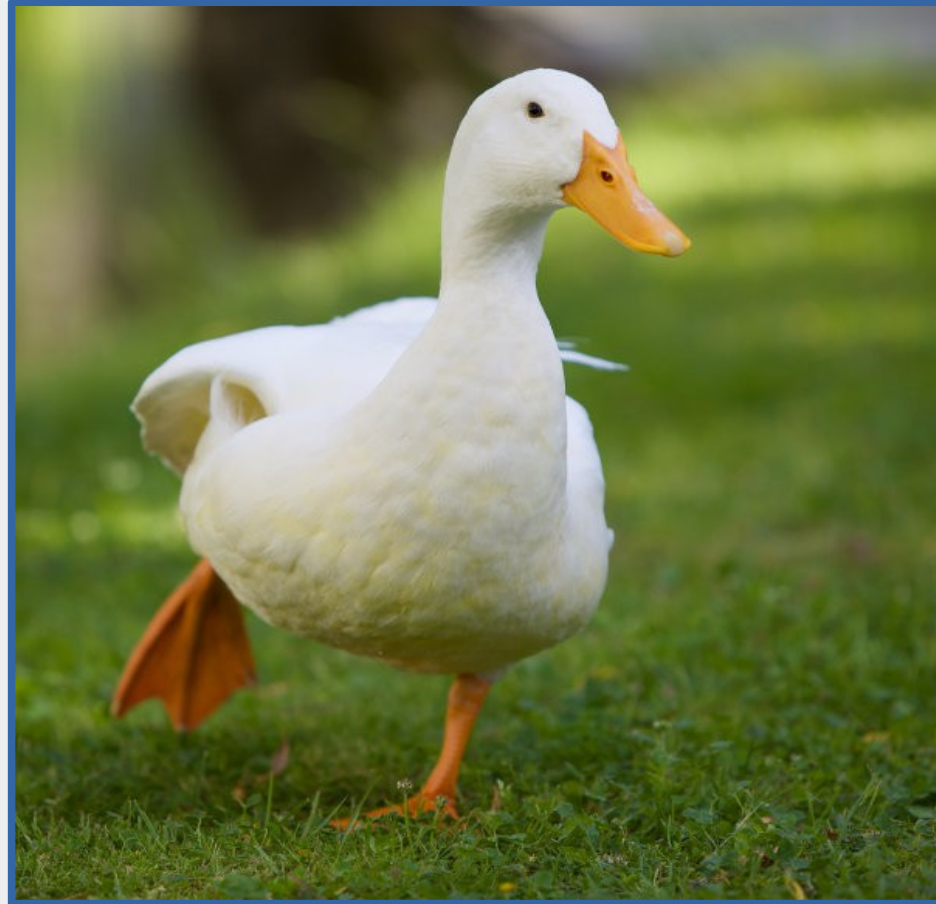
```
>>> open_door(door1)
```

```
>>> door1  
[1, 'open']
```

```
>>> open_ldoor(ldoor1)
```

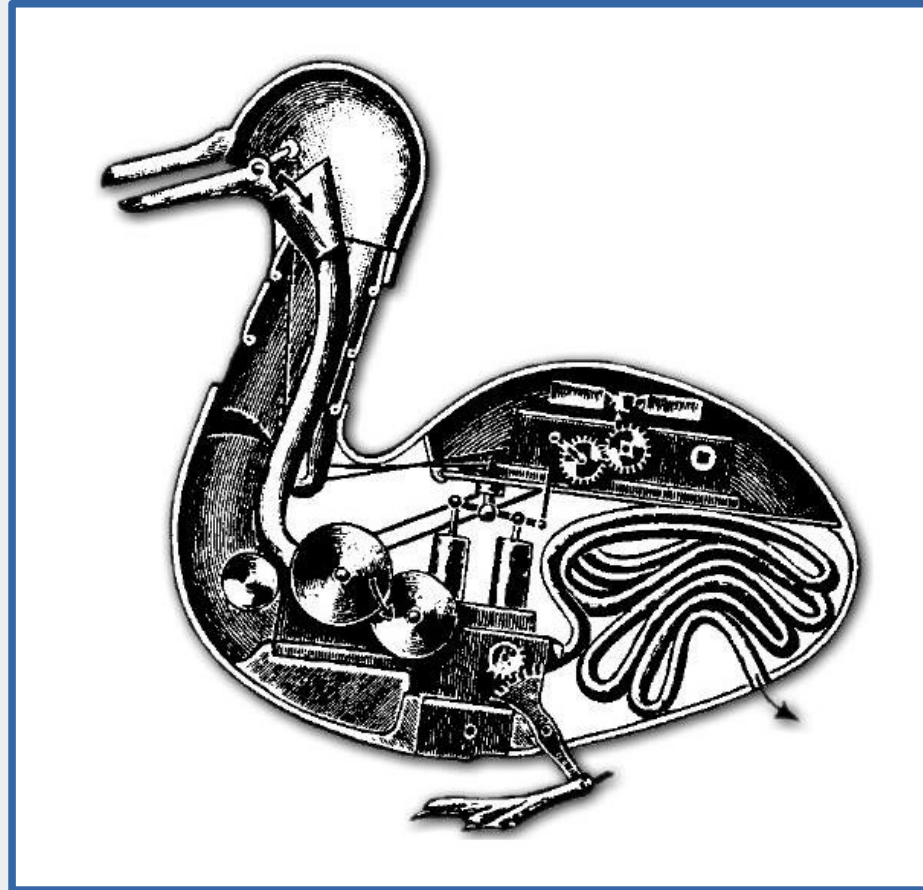
```
>>> ldoor1  
[1, 'open', 'unlocked']
```

The meaning of the word 'type'



Behavioural meaning

The meaning of the word 'type'



Structural meaning

The behavioural meaning is important

The meaning of the word 'class'



The meaning of the word 'instance'



You already used classes

```
>>> a = 6
```

```
>>> a
```

```
6
```

```
>>> type(a)
```

```
<class 'int'>
```

```
>>> a = int(6)
```

```
>>> a
```

```
6
```

```
>>> a.__class__
```

```
<class 'int'>
```

```
>>> a = 6
```

```
>>> a
```

```
6
```

```
>>> type(a)
```

```
<type 'int'>
```

```
>>> a = int(6)
```

```
>>> a
```

```
6
```

```
>>> a.__class__
```

```
<type 'int'>
```

1. Find some types that Python provides out of the box.
2. Can you create a float variable without using the '.' character?
3. What is the difference between {1,2,3} and {'a':1, 'b':2}?

1. Find some types that Python provides out of the box.

```
>>> type(4)
<class 'int'>
>>> type(4.5)
<class 'float'>
>>> type('some words')
<class 'str'>
>>> type([1,2,3,4])
<class 'list'>
>>> type((1,2,3,4))
<class 'tuple'>
>>> type({1,2,3,4})
<class 'set'>
>>> type({'first name': 'Ray', 'last name': 'Stantz'})
<class 'dict'>
>>> type(True)
<class 'bool'>
```

2. Can you create a float variable without using the '.' character?

```
>>> a = float(4)
>>> type(a)
<class 'float'>
```

3. What is the difference between {1,2,3} and {'a':1, 'b':2}?

```
>>> type({1,2,3})  
<class 'set'>  
>>> type({'a':1, 'b':2})  
<class 'dict'>
```

Keep data and functions together

```
>>> door1 = Door(1, 'closed')  
>>> type(door1)  
<class Door>
```

Keep data and functions together

```
>>> door1 = Door(1, 'closed')
>>> type(door1)
<class Door>
>>> door1.number
1
>>> door1.status
'closed'
```

Keep data and functions together

```
>>> door1 = Door(1, 'closed')
>>> type(door1)
<class Door>
>>> door1.number
1
>>> door1.status
'closed'
>>> door1.open()
```

Keep data and functions together

```
>>> door1 = Door(1, 'closed')
>>> type(door1)
<class Door>
>>> door1.number
1
>>> door1.status
'closed'
>>> door1.open()
>>> door1.number
1
>>> door1.status
'open'
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
class Door(object):  
    def __init__(self, number, status):  
        self.number = number  
        self.status = status  
  
    def open(self):  
        self.status = 'open'  
  
    def close(self):  
        self.status = 'closed'
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
>>> door1 = Door(1, 'closed')
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
>>> door1 = Door(1, 'closed')
>>> type(door1)
<class '__main__.Door'>
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
>>> door1 = Door(1, 'closed')
>>> type(door1)
<class '__main__.Door'>
>>> door1.number
1
>>> door1.status
'closed'
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
>>> door1 = Door(1, 'closed')
>>> type(door1)
<class '__main__.Door'>
>>> door1.number
1
>>> door1.status
'closed'
>>> door1.open()
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
>>> door1 = Door(1, 'closed')
>>> type(door1)
<class '__main__.Door'>
>>> door1.number
1
>>> door1.status
'closed'
>>> door1.open()
>>> door1.number
1
>>> door1.status
'open'
```

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

1. Create a ColouredDoor class that has the 'colour' attribute.
2. Create a ClosedDoor class that has a default status of 'closed'.
3. Create a ToggleDoor class that has a method toggle() that toggles the status of the door.

1. Create a ColouredDoor class that has the 'colour' attribute.

```
class ColouredDoor:
    def __init__(self, number, status, colour):
        self.number = number
        self.status = status
        self.colour = colour

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

2. Create a ClosedDoor class that has a default status of 'closed'.

```
class ClosedDoor:
    def __init__(self, number, status='closed'):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

3. Create a `ToggleDoor` class that has a method `toggle()` that toggles the status of the door.

```
class ToggleDoor:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

    def toggle(self):
        d = {'open': 'closed', 'closed': 'open'}
        self.status = d[self.status]
```

PART 2

Classes and members



Where is the class of an object?

```
>>> door1 = Door(1, 'closed')  
>>> door2 = Door(1, 'closed')
```

Where is the class of an object?

```
>>> door1 = Door(1, 'closed')
>>> door2 = Door(1, 'closed')
>>> hex(id(door1))
'0xb67e148c'
>>> hex(id(door2))
'0xb67e144c'
```

Where is the class of an object?

```
>>> door1 = Door(1, 'closed')
>>> door2 = Door(1, 'closed')
>>> hex(id(door1))
'0xb67e148c'
>>> hex(id(door2))
'0xb67e144c'
>>> hex(id(door1.__class__))
'0xb685f56c'
>>> hex(id(door2.__class__))
'0xb685f56c'
```

```
class Door:
    colour = 'brown'

    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

```
>>> door1 = Door(1, 'closed')  
>>> door2 = Door(2, 'closed')
```

```
>>> door1 = Door(1, 'closed')
>>> door2 = Door(2, 'closed')
>>> Door.colour
'brown'
>>> door1.colour
'brown'
>>> door2.colour
'brown'
```

```
>>> Door.colour = 'white'
```

```
>>> Door.colour = 'white'
>>> Door.colour
'white'
>>> door1.colour
'white'
>>> door2.colour
'white'
```

```
>>> Door.colour = 'white'
>>> Door.colour
'white'
>>> door1.colour
'white'
>>> door2.colour
'white'
>>> hex(id(Door.colour))
'0xb67e1500'
>>> hex(id(door1.colour))
'0xb67e1500'
>>> hex(id(door2.colour))
'0xb67e1500'
```

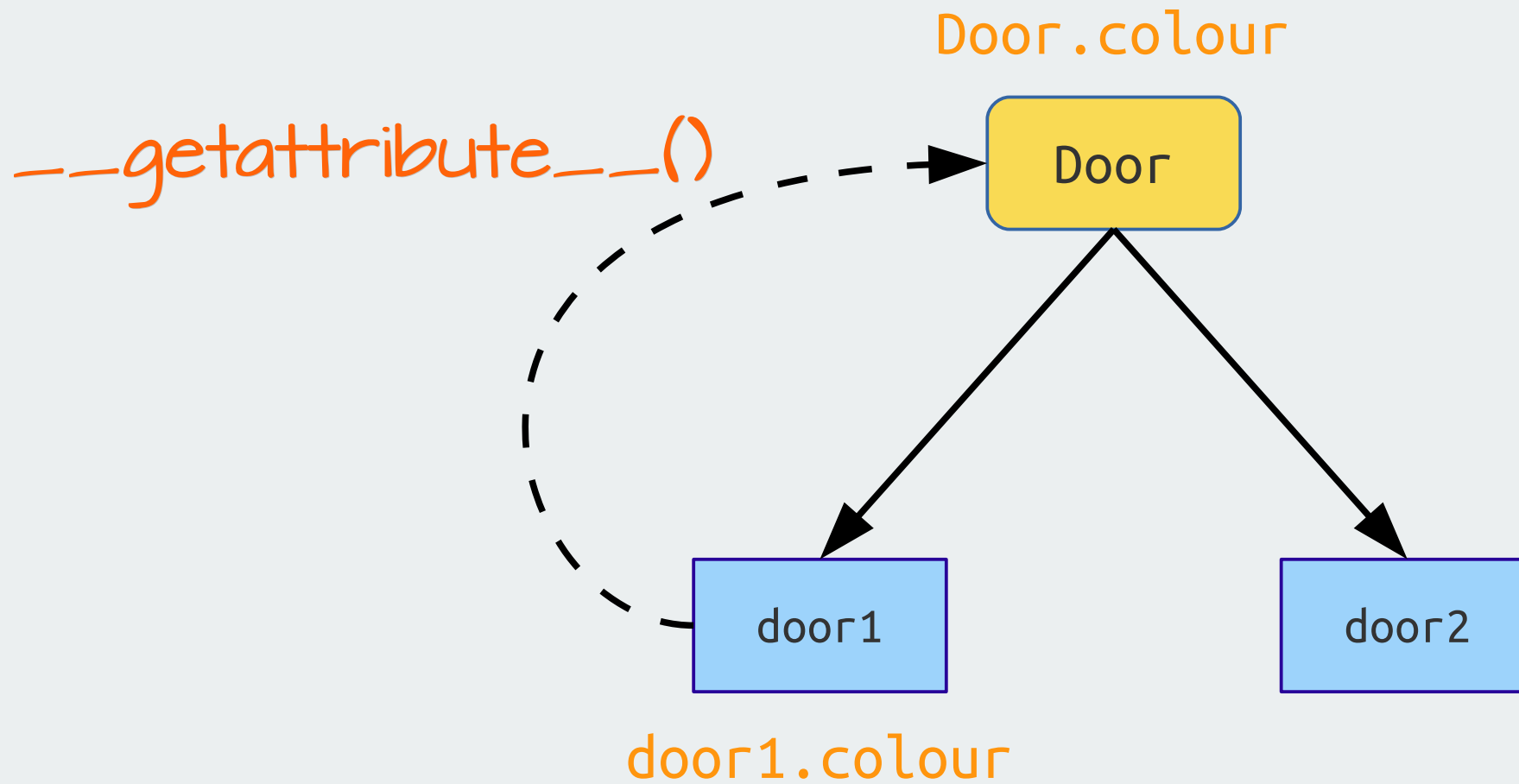
```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
  'colour': 'white',
  '__dict__': <attribute '__dict__' of 'Door' objects>,
  '__weakref__': <attribute '__weakref__' of 'Door' objects>,
  '__init__': <function Door.__init__ at 0xb7062854>,
  '__module__': '__main__',
  '__doc__': None,
  'close': <function Door.close at 0xb686041c>})
```

```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
             'colour': 'white',
             '__dict__': <attribute '__dict__' of 'Door' objects>,
             '__weakref__': <attribute '__weakref__' of 'Door' objects>,
             '__init__': <function Door.__init__ at 0xb7062854>,
             '__module__': '__main__',
             '__doc__': None,
             'close': <function Door.close at 0xb686041c>})
>>> door1.__dict__
{'number': 1, 'status': 'closed'}
```

```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
              'colour': 'white',
              '__dict__': <attribute '__dict__' of 'Door' objects>,
              '__weakref__': <attribute '__weakref__' of 'Door' objects>,
              '__init__': <function Door.__init__ at 0xb7062854>,
              '__module__': '__main__',
              '__doc__': None,
              'close': <function Door.close at 0xb686041c>})
>>> door1.__dict__
{'number': 1, 'status': 'closed'}
>>> door1.__dict__['colour']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'colour'
```

```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
             'colour': 'white',
             '__dict__': <attribute '__dict__' of 'Door' objects>,
             '__weakref__': <attribute '__weakref__' of 'Door' objects>,
             '__init__': <function Door.__init__ at 0xb7062854>,
             '__module__': '__main__',
             '__doc__': None,
             'close': <function Door.close at 0xb686041c>})
>>> door1.__dict__
{'number': 1, 'status': 'closed'}
>>> door1.__dict__['colour']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'colour'
>>> door1.__class__.__dict__['colour']
'white'
>>> door1.colour is Door.colour
True
```

Let's dive into attribute resolution



Let's dive into attribute resolution

```
>>> door1 = Door(1, 'closed')
>>> door1.colour = 'white'
>>> door1.__dict__['colour']
'white'
>>> door1.__class__.__dict__['colour']
'brown'
```

Let's dive into attribute resolution

```
>>> door1 = Door(1, 'closed')
>>> door1.colour = 'white'
>>> door1.__dict__['colour']
'white'
>>> door1.__class__.__dict__['colour']
'brown'
>>> door1.colour
'white'
>>> Door.colour
'brown'
```

Let's dive into attribute resolution

```
>>> door1 = Door(1, 'closed')
>>> door1.colour = 'white'
>>> door1.__dict__['colour']
'white'
>>> door1.__class__.__dict__['colour']
'brown'
>>> door1.colour
'white'
>>> Door.colour
'brown'
>>> Door.colour = 'red'
```

Let's dive into attribute resolution

```
>>> door1 = Door(1, 'closed')
>>> door1.colour = 'white'
>>> door1.__dict__['colour']
'white'
>>> door1.__class__.__dict__['colour']
'brown'
>>> door1.colour
'white'
>>> Door.colour
'brown'
>>> Door.colour = 'red'
>>> door1.__dict__['colour']
'white'
>>> door1.__class__.__dict__['colour']
'red'
```

1. Modify the Door class adding a class attribute 'status' with value 'undefined'. Does it work? What happens to instances?
2. Modify the Door class adding a class attribute 'status' with value 'closed' and remove status from `__init__()`. Does it work?
3. Add a `toggle()` method to the previous class. What happens if you call `toggle()` on a fresh instance? Why?

1. Modify the Door class adding a class attribute 'status' with value 'undefined'. Does it work? What happens to instances?

```
class Door:
    colour = 'brown'
    status = 'undefined'

    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

2. Modify the Door class adding a class attribute 'status' with value 'closed' and remove status from `__init__()`. Does it work?

```
class Door:
    colour = 'brown'
    status = 'closed'

    def __init__(self, number):
        self.number = number

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

3. Add a `toggle()` method to the previous class. What happens if you call `toggle()` on a fresh instance? Why?

```
class Door:
    colour = 'brown'
    status = 'closed'

    def __init__(self, number):
        self.number = number

    def toggle(self):
        d = {'open': 'closed', 'closed': 'open'}
        self.status = d[self.status]
```

What about methods?

```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
  'colour': 'white',
  '__dict__': <attribute '__dict__' of 'Door' objects>,
  '__weakref__': <attribute '__weakref__' of 'Door' objects>,
  '__init__': <function Door.__init__ at 0xb7062854>,
  '__module__': '__main__',
  '__doc__': None,
  'close': <function Door.close at 0xb686041c>})
>>> door1.__dict__
{'number': 1, 'status': 'closed'}
>>> door1.colour is Door.colour
True
```

What about methods?

```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
  'colour': 'white',
  '__dict__': <attribute '__dict__' of 'Door' objects>,
  '__weakref__': <attribute '__weakref__' of 'Door' objects>,
  '__init__': <function Door.__init__ at 0xb7062854>,
  '__module__': '__main__',
  '__doc__': None,
  'close': <function Door.close at 0xb686041c>})
>>> door1.__dict__
{'number': 1, 'status': 'closed'}
>>> door1.colour is Door.colour
True
>>> door1.open is Door.open
False
```

What about methods?

```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
             'colour': 'white',
             '__dict__': <attribute '__dict__' of 'Door' objects>,
             '__weakref__': <attribute '__weakref__' of 'Door' objects>,
             '__init__': <function Door.__init__ at 0xb7062854>,
             '__module__': '__main__',
             '__doc__': None,
             'close': <function Door.close at 0xb686041c>})
>>> door1.__dict__
{'number': 1, 'status': 'closed'}
>>> door1.colour is Door.colour
True
>>> door1.open is Door.open
False
>>> Door.__dict__['open']
<function Door.open at 0xb68604ac>
>>> Door.open
<function Door.open at 0xb68604ac>
```

What about methods?

```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
             'colour': 'white',
             '__dict__': <attribute '__dict__' of 'Door' objects>,
             '__weakref__': <attribute '__weakref__' of 'Door' objects>,
             '__init__': <function Door.__init__ at 0xb7062854>,
             '__module__': '__main__',
             '__doc__': None,
             'close': <function Door.close at 0xb686041c>})
>>> door1.__dict__
{'number': 1, 'status': 'closed'}
>>> door1.colour is Door.colour
True
>>> door1.open is Door.open
False
>>> Door.__dict__['open']
<function Door.open at 0xb68604ac>
>>> Door.open
<function Door.open at 0xb68604ac>
>>> door1.open
<bound method Door.open of <__main__.Door object at 0xb67e162c>>
```

```
>>> Door.__dict__
mappingproxy({'open': <function Door.open at 0xb68604ac>,
             'colour': 'white',
             '__dict__': <attribute '__dict__' of 'Door' objects>,
             '__weakref__': <attribute '__weakref__' of 'Door' objects>,
             '__init__': <function Door.__init__ at 0xb7062854>,
             '__module__': '__main__',
             '__doc__': None,
             'close': <function Door.close at 0xb686041c>})
>>> door1.__dict__
{'number': 1, 'status': 'closed'}
>>> door1.colour is Door.colour
True
>>> door1.open is Door.open
False
>>> Door.__dict__['open']
<function Door.open at 0xb68604ac>
>>> Door.open
<unbound method Door.open>
>>> door1.open
<bound method Door.open of <__main__.Door object at 0xb67e162c>>
```

From functions to bound methods

```
>>> Door.open()
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: open() missing 1 required positional argument: 'self'
```

```
>>> Door.open()
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: unbound method open() must be called with Door instance  
as first argument (got nothing instead)
```

From functions to bound methods

```
>>> Door.open()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: open() missing 1 required positional argument: 'self'  
>>> Door.open(door1)  
>>> door1.status  
'open'
```

From functions to bound methods

```
>>> Door.open()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: open() missing 1 required positional argument: 'self'
>>> Door.open(door1)
>>> door1.status
'open'
>>> door1.__class__.__dict__['open']
<function Door.open at 0xb68604ac>
```

From functions to bound methods

```
>>> Door.open()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: open() missing 1 required positional argument: 'self'
>>> Door.open(door1)
>>> door1.status
'open'
>>> door1.__class__.__dict__['open']
<function Door.open at 0xb68604ac>
>>> dir(door1.__class__.__dict__['open'])
['__annotations__', '__call__', '__class__', '__closure__',
 '__code__', '__defaults__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__get__', '__getattr__', '__globals__', '__gt__',
 '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
 '__module__', '__name__', '__ne__', '__new__', '__qualname__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
```

From functions to bound methods

```
>>> Door.open()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: open() missing 1 required positional argument: 'self'
>>> Door.open(door1)
>>> door1.status
'open'
>>> door1.__class__.__dict__['open']
<function Door.open at 0xb68604ac>
>>> dir(door1.__class__.__dict__['open'])
['__annotations__', '__call__', '__class__', '__closure__',
..., '__get__', ...]
>>> door1.__class__.__dict__['open'].__get__
<method-wrapper '__get__' of function object at 0xb68604ac>
```

From functions to bound methods

```
>>> Door.open()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: open() missing 1 required positional argument: 'self'
>>> Door.open(door1)
>>> door1.status
'open'
>>> door1.__class__.__dict__['open']
<function Door.open at 0xb68604ac>
>>> dir(door1.__class__.__dict__['open'])
['__annotations__', '__call__', '__class__', '__closure__',
..., '__get__', ...]
>>> door1.__class__.__dict__['open'].__get__
<method-wrapper '__get__' of function object at 0xb68604ac>
>>> door1.__class__.__dict__['open'].__get__(door1)
<bound method Door.open of <__main__.Door object at 0xb67e162c>>
```

```
>>> Door.open()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: open() missing 1 required positional argument: 'self'
>>> Door.open(door1)
>>> door1.status
'open'
>>> door1.__class__.__dict__['open']
<function Door.open at 0xb68604ac>
>>> dir(door1.__class__.__dict__['open'])
['__annotations__', '__call__', '__class__', '__closure__',
..., '__get__', ...]
>>> door1.__class__.__dict__['open'].__get__
<method-wrapper '__get__' of function object at 0xb68604ac>
>>> door1.__class__.__dict__['open'].__get__(door1)
<bound method ?.open of <__main__.Door instance at 0xb6977aac>>
>>> door1.__class__.__dict__['open'].__get__(door1, Door)
<bound method Door.open of <__main__.Door object at 0xb73f956c>>
```

```
class Door:
    colour = 'brown'

    def __init__(self, number, status):
        self.number = number
        self.status = status

    @classmethod
    def knock(cls):
        print("Knock!")

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

From functions to bound methods

```
>>> door1 = Door(1, 'closed')
>>> door1.knock()
Knock!
>>> Door.knock()
Knock!
```

1. Modify the Door class adding a class method 'paint' that accepts a 'colour' argument and changes the class attribute 'colour'.
2. Modify the Door class adding both a class method 'paint' and a standard method 'paint'. What happens?

1. Modify the Door class adding a class method 'paint' that accepts a 'colour' argument and changes the class attribute 'colour'.

```
class Door:
    colour = 'brown'

    def __init__(self, number, status):
        self.number = number
        self.status = status

    @classmethod
    def paint(cls, colour):
        cls.colour = colour

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'
```

2. Modify the Door class adding both a class method 'paint' and a standard method 'paint'. What happens?

```
class Door:
    colour = 'brown'

    def __init__(self, number, status):
        self.number = number
        self.status = status

    @classmethod
    def paint(cls, colour):
        cls.colour = colour

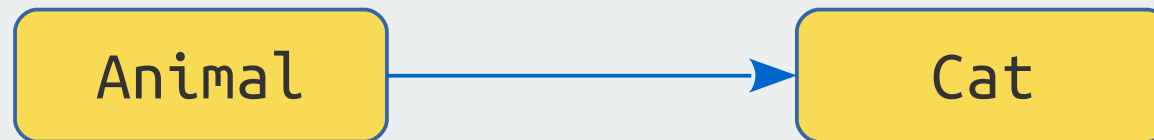
    def paint(self, colour):
        self.colour = colour

[...]
```

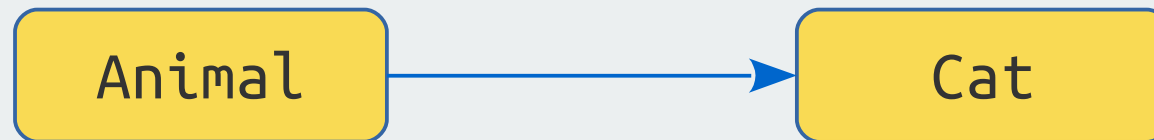
PART 3

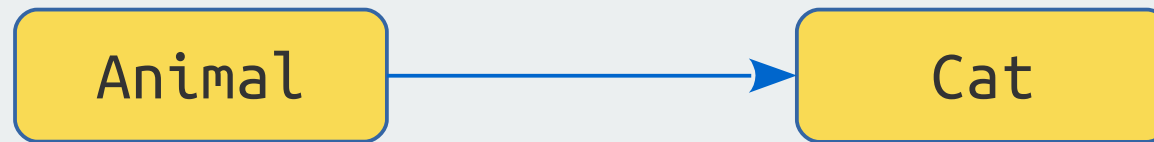
Delegation





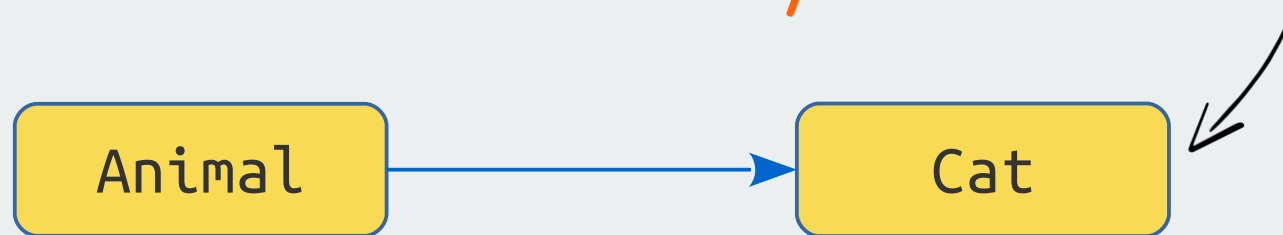
Cat has all the features of
Animal, i.e. 'moves'



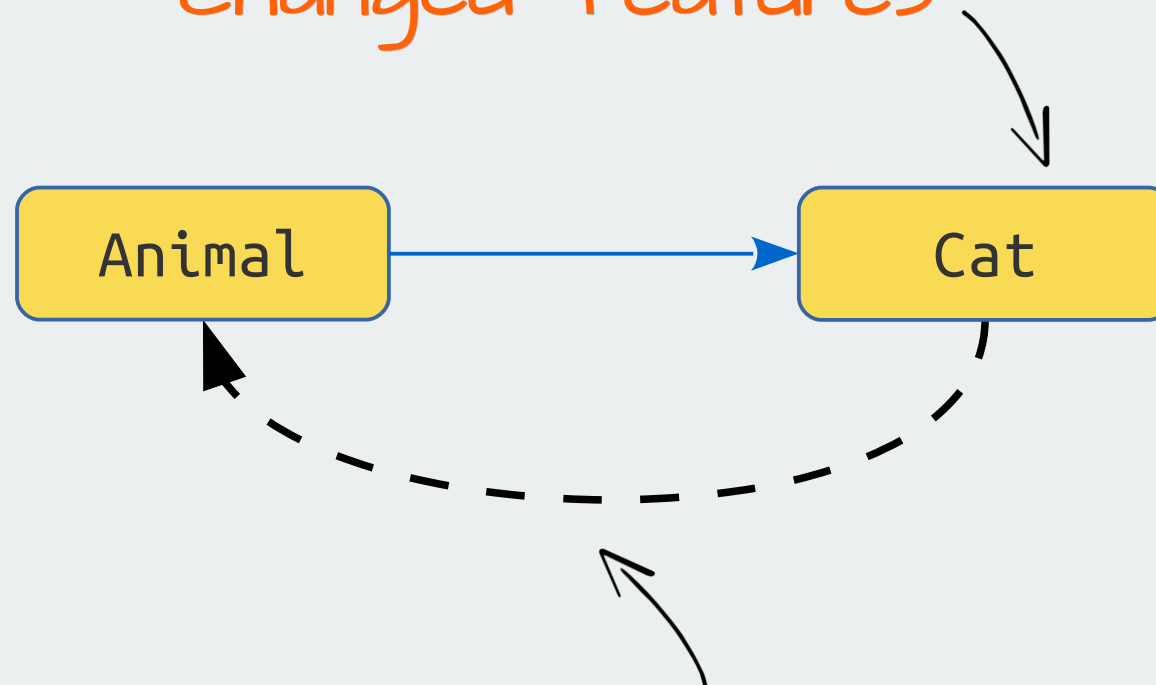


Cat may provide new features, i.e.
'has whiskers'

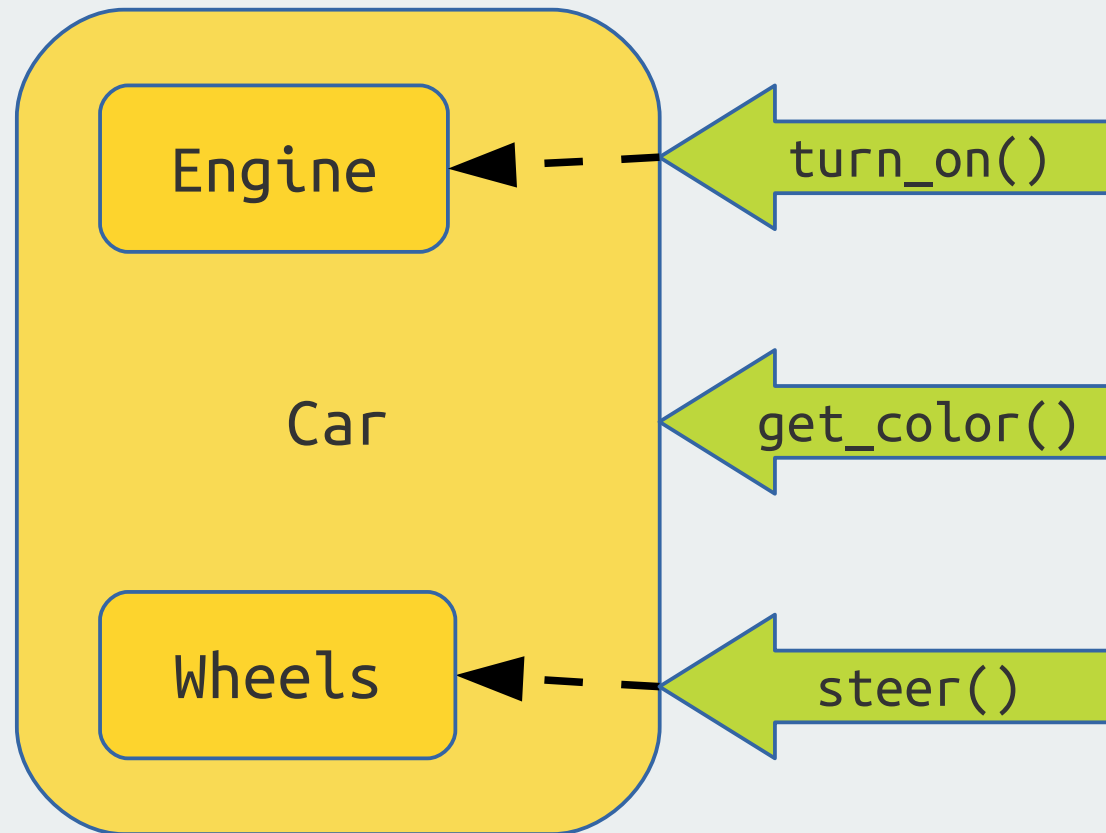
Cat performs some or all the tasks performed by Animal in a different way, i.e. 'moves silently'

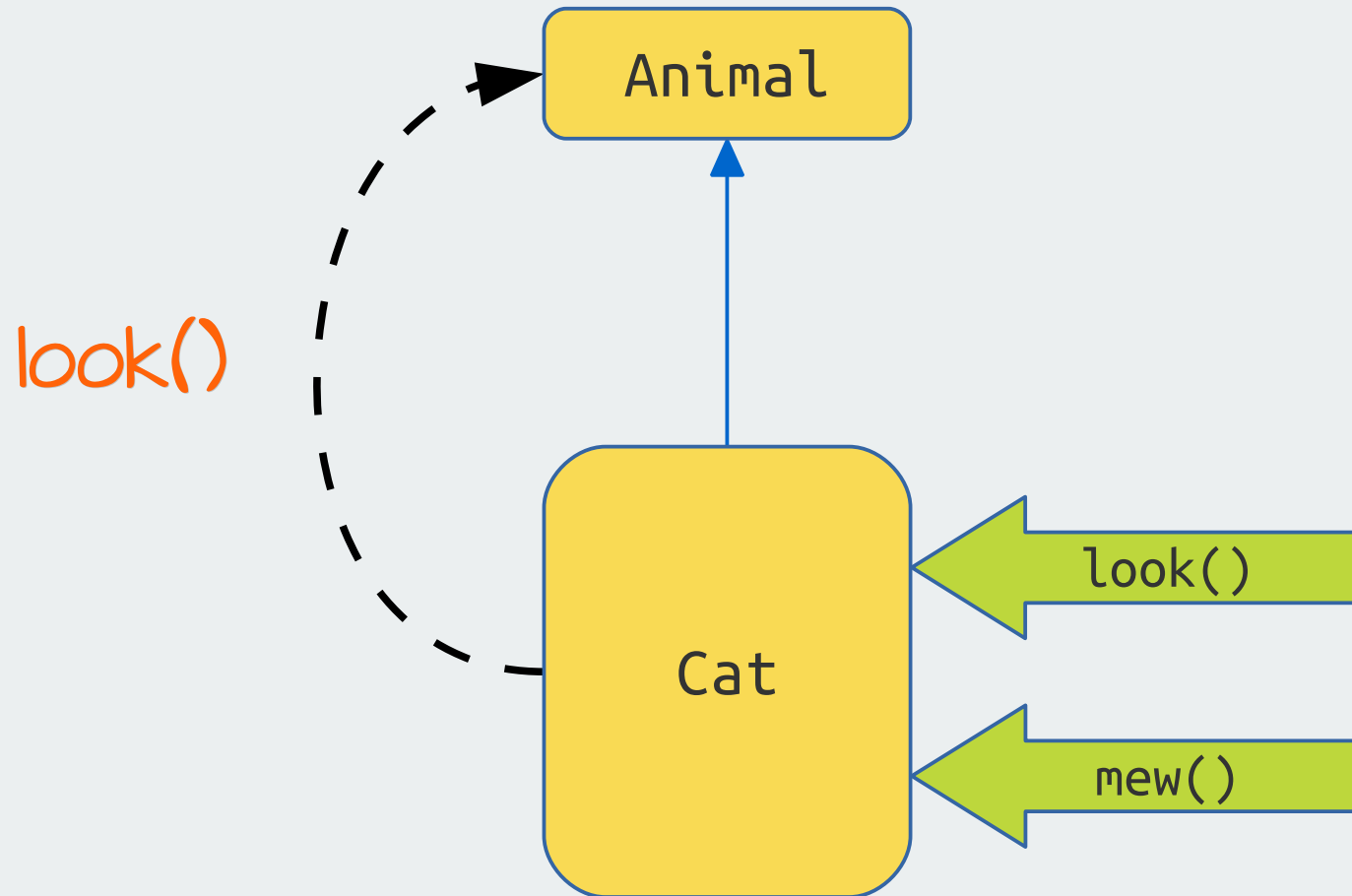


Cat implements only 'new' or
'changed' features



Cat delegates the remaining
features to Animal





LOVE YOUR SYSTEM

www.compositionoverinheritance.com



```
class SecurityDoor(Door):  
    pass
```

```
class SecurityDoor(Door):  
    pass
```

```
>>> sdoor = SecurityDoor(1, 'closed')
```

```
class SecurityDoor(Door):  
    pass
```

```
>>> sdoor = SecurityDoor(1, 'closed')  
>>> SecurityDoor.colour is Door.colour  
True  
>>> sdoor.colour is Door.colour  
True
```

Inheritance

```
class SecurityDoor(Door):  
    pass
```

```
>>> sdoor = SecurityDoor(1, 'closed')  
>>> SecurityDoor.colour is Door.colour  
True  
>>> sdoor.colour is Door.colour  
True
```



```
>>> sdoor.__dict__  
{'number': 1, 'status': 'closed'}
```

```
>>> sdoor.__dict__
{'number': 1, 'status': 'closed'}
>>> sdoor.__class__.__dict__
mappingproxy({'__doc__': None, '__module__': '__main__'})
>>> Door.__dict__
mappingproxy({'__dict__': <attribute '__dict__' of 'Door' objects>,
  'colour': 'yellow',
  'open': <function Door.open at 0xb687e224>,
  '__init__': <function Door.__init__ at 0xb687e14c>,
  '__doc__': None,
  'close': <function Door.close at 0xb687e1dc>,
  'knock': <classmethod object at 0xb67ff6ac>,
  '__weakref__': <attribute '__weakref__' of 'Door' objects>,
  '__module__': '__main__',
  'paint': <classmethod object at 0xb67ff6ec>})
```

```
>>> sdoor.__dict__
{'number': 1, 'status': 'closed'}
>>> sdoor.__class__.__dict__
mappingproxy({'__doc__': None, '__module__': '__main__'})
>>> Door.__dict__
mappingproxy({'__dict__': <attribute '__dict__' of 'Door' objects>,
  'colour': 'yellow',
  'open': <function Door.open at 0xb687e224>,
  '__init__': <function Door.__init__ at 0xb687e14c>,
  '__doc__': None,
  'close': <function Door.close at 0xb687e1dc>,
  'knock': <classmethod object at 0xb67ff6ac>,
  '__weakref__': <attribute '__weakref__' of 'Door' objects>,
  '__module__': '__main__',
  'paint': <classmethod object at 0xb67ff6ec>})
>>> SecurityDoor.__bases__
(<class '__main__.Door'>,)

```

```
>>> sdoor.__dict__
{'number': 1, 'status': 'closed'}
>>> sdoor.__class__.__dict__
mappingproxy({'__doc__': None, '__module__': '__main__'})
>>> Door.__dict__
mappingproxy({'__dict__': <attribute '__dict__' of 'Door' objects>,
  'colour': 'yellow',
  'open': <function Door.open at 0xb687e224>,
  '__init__': <function Door.__init__ at 0xb687e14c>,
  '__doc__': None,
  'close': <function Door.close at 0xb687e1dc>,
  'knock': <classmethod object at 0xb67ff6ac>,
  '__weakref__': <attribute '__weakref__' of 'Door' objects>,
  '__module__': '__main__',
  'paint': <classmethod object at 0xb67ff6ec>})
>>> SecurityDoor.__bases__
(<class '__main__.Door'>,)
>>> sdoor.knock
<bound method type.knock of <class '__main__.SecurityDoor'>>
>>> sdoor.__class__.__bases__[0].__dict__['knock'].__get__(sdoor)
<bound method type.knock of <class '__main__.SecurityDoor'>>
```

```
class SecurityDoor(Door):  
    colour = 'grey'  
    locked = True  
  
    def open(self):  
        if not self.locked:  
            self.status = 'open'
```

```
class SecurityDoor(Door):  
    colour = 'grey'  
    locked = True  
  
    def open(self):  
        if not self.locked:  
            self.status = 'open'
```

```
>>> SecurityDoor.__dict__  
mappingproxy({'__doc__': None,  
              '__module__': '__main__',  
              'open': <function SecurityDoor.open at 0xb6fcf89c>,  
              'colour': 'grey',  
              'locked': True})
```

1. Modify the SecurityDoor class adding a custom `close_and_lock()` method that changes status to 'closed' and locked to True. Test it.
2. Modify the SecurityDoor class adding a custom `close()` method that accepts a locked flag (with default).
3. Modify the SecurityDoor class adding a custom `__init__()` method that sets `self.locked`.

1. Modify the SecurityDoor class adding a custom `close_and_lock()` method that changes status to 'closed' and locked to True. Test it.

```
class SecurityDoor(Door):  
    colour = 'grey'  
    locked = False  
  
    def open(self):  
        if not self.locked:  
            self.status = 'open'  
  
    def close_and_lock(self):  
        self.status = 'closed'  
        self.locked = True
```

2. Modify the SecurityDoor class adding a custom close() method that accepts a locked flag (with default).

```
class SecurityDoor(Door):  
    colour = 'grey'  
    locked = False  
  
    def open(self):  
        if not self.locked:  
            self.status = 'open'  
  
    def close(self, locked=False):  
        self.status = 'closed'  
        self.locked = locked
```

3. Modify the SecurityDoor class adding a custom `__init__()` method that sets `self.locked`.

```
class SecurityDoor(Door):  
    colour = 'grey'  
  
    def __init__(self, number, status, locked=False):  
        self.number = number  
        self.status = status  
        self.locked = locked  
  
    def open(self):  
        if not self.locked:  
            self.status = 'open'
```

```
class SecurityDoor(Door):  
    colour = 'grey'  
  
    def __init__(self, number, status, locked=True):  
        Door.__init__(self, number, status)  
        self.locked = locked  
  
    def open(self):  
        if self.locked:  
            return  
        Door.open(self)
```

```
class SecurityDoor(Door):  
    colour = 'grey'  
  
    def __init__(self, number, status, locked=True):  
        Door.__init__(self, number, status)  
        self.locked = locked  
  
    def open(self):  
        if self.locked:  
            return  
        Door.open(self)  
  
>>> sdoor = SecurityDoor(1, 'closed')
```

```
class SecurityDoor(Door):
    colour = 'grey'

    def __init__(self, number, status, locked=True):
        Door.__init__(self, number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        Door.open(self)

>>> sdoor = SecurityDoor(1, 'closed')
>>> sdoor.open()
>>> sdoor.status
'closed'
```

```
class SecurityDoor(Door):  
    colour = 'grey'  
  
    def __init__(self, number, status, locked=True):  
        Door.__init__(self, number, status)  
        self.locked = locked  
  
    def open(self):  
        if self.locked:  
            return  
        Door.open(self)
```

```
>>> sdoor = SecurityDoor(1, 'closed')  
>>> sdoor.open()  
>>> sdoor.status  
'closed'  
>>> sdoor.locked = False  
>>> sdoor.open()  
>>> sdoor.status  
'open'
```

Avoid strong coupling

```
class SecurityDoor(Door):  
    colour = 'grey'  
  
    def __init__(self, number, status, locked=True):  
        super().__init__(number, status)  
        self.locked = locked  
  
    def open(self):  
        if self.locked:  
            return  
        super().open()
```

```
class SecurityDoor(Door):
    colour = 'grey'

    def __init__(self, number, status, locked=True):
        super(SecurityDoor, self).__init__(number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        super(SecurityDoor, self).open()
```

```
class SecurityDoor:
    colour = 'grey'

    def __init__(self, number, status, locked=True):
        self.door = Door(number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        self.door.open()

    def close(self):
        self.door.close()
```

```
class SecurityDoor:
    colour = 'grey'

    def __init__(self, number, status, locked=True):
        self.door = Door(number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        self.door.open()

    def close(self):
        self.door.close()

>>> sdoor = SecurityDoor(1, 'closed')
>>> sdoor.status
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'SecurityDorr' object has no attribute 'status'
```

```
class SecurityDoor:
    colour = 'grey'

    def __init__(self, number, status, locked=True):
        self.door = Door(number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        self.door.open()

    def close(self):
        self.door.close()

    def get_status(self):
        return self.door.status
    status = property(get_status)

>>> sdoor = SecurityDoor(1, 'closed')
>>> sdoor.status
'closed'
```

Python magic to the rescue

```
class SecurityDoor:
    colour = 'grey'

    def __init__(self, number, status, locked=True):
        self.door = Door(number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        self.door.open()

    def close(self):
        self.door.close()

    def __getattr__(self, attr):
        return getattr(self.door, attr)

>>> sdoor = SecurityDoor(1, 'closed')
>>> sdoor.status
'closed'
```

Python magic to the rescue

```
class SecurityDoor:
    colour = 'grey'

    def __init__(self, number, status, locked=True):
        self.door = Door(number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        self.door.open()

    #def close(self):
    #    self.door.close()

    def __getattr__(self, attr):
        return getattr(self.door, attr)
```

Composed inheritance?

```
class ComposedDoor:  
    def __init__(self, number, status):  
        self.door = Door(number, status)  
  
    def __getattr__(self, attr):  
        return getattr(self.door, attr)
```

1. Modify the SecurityDoor class adding a custom close_and_lock() method. Use super(). Try and implement it with composition.
2. Modify the SecurityDoor class adding a custom close() method that accepts a locked flag (with default). Use super(). Try and implement it with composition.

1. Modify the SecurityDoor class adding a custom `close_and_lock()` method. Use `super()`. Try and implement it with composition.

```
class SecurityDoor(Door):  
    colour = 'grey'  
  
    def __init__(self, number, status, locked=False):  
        super().__init__(number, status)  
        self.locked = locked  
  
    def open(self):  
        if self.locked:  
            return  
        super().open()  
  
    def close_and_lock(self):  
        super().close()  
        self.locked = True
```

1. Modify the SecurityDoor class adding a custom `close_and_lock()` method. Use `super()`. Try and implement it with composition.

```
class SecurityDoor:
    colour = 'grey'

    def __init__(self, number, status, locked=False):
        self.door = Door(number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        self.door.open()

    def __getattr__(self, attr):
        return getattr(self.door, attr)

    def close_and_lock(self):
        self.door.close()
        self.locked = True
```

2. Modify the SecurityDoor class adding a custom close() method that accepts a locked flag (with default). Use super(). Try and implement it with composition.

```
class SecurityDoor(Door):  
    colour = 'grey'  
  
    def __init__(self, number, status, locked=False):  
        super().__init__(number, status)  
        self.locked = locked  
  
    def open(self):  
        if self.locked:  
            return  
        super().open()  
  
    def close(self, locked=False):  
        super().close()  
        self.locked = locked
```

2. Modify the SecurityDoor class adding a custom close() method that accepts a locked flag (with default). Use super(). Try and implement it with composition.

```
class SecurityDoor:
    colour = 'grey'

    def __init__(self, number, status, locked=False):
        self.door = Door(number, status)
        self.locked = locked

    def open(self):
        if self.locked:
            return
        self.door.open()

    def __getattr__(self, attr):
        return getattr(self.door, attr)

    def close(self, locked=False):
        self.door.close()
        self.locked = locked
```

```
>>> l = [1,2,3]
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__',
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattr__', '__getitem__', '__gt__',
 '__hash__', '__iadd__', '__imul__', '__init__', '__iter__',
 '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
 '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
 '__str__', '__subclasshook__', 'append', 'clear', 'copy',
 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
```

```
>>> l = [1,2,3]
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__',
 ..., 'append', 'clear', 'copy', ...]
>>> l.append
<built-in method append of list object at 0xb70a2c2c>
```

```
>>> l = [1,2,3]
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__',
..., 'append', 'clear', 'copy', ...]
>>> l.append
<built-in method append of list object at 0xb70a2c2c>
>>> a = l.append
>>> a
<built-in method append of list object at 0xb70a2c2c>
```

```
>>> l = [1,2,3]
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__',
..., 'append', 'clear', 'copy', ...]
>>> l.append
<built-in method append of list object at 0xb70a2c2c>
>>> a = l.append
>>> a
<built-in method append of list object at 0xb70a2c2c>
>>> b = getattr(l, 'append')
>>> b
<built-in method append of list object at 0xb70a2c2c>
```

```
>>> l = [1,2,3]
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__',
..., 'append', 'clear', 'copy', ...]
>>> l.append
<built-in method append of list object at 0xb70a2c2c>
>>> a = l.append
>>> a
<built-in method append of list object at 0xb70a2c2c>
>>> b = getattr(l, 'append')
>>> b
<built-in method append of list object at 0xb70a2c2c>
>>> a == b
True
>>> a is b
False
```

PART 4

Polymorphism



```
>>> a = 5
```

```
>>> a
```

```
5
```

```
>>> a = 5
>>> a
5
>>> type(a)
<class 'int'>
>>> hex(id(a))
'0x83fe540'
```

```
>>> a = 5
>>> a
5
>>> type(a)
<class 'int'>
>>> hex(id(a))
'0x83fe540'
>>> a = 'five'
>>> a
'five'
```

```
>>> a = 5
>>> a
5
>>> type(a)
<class 'int'>
>>> hex(id(a))
'0x83fe540'
>>> a = 'five'
>>> a
'five'
>>> type(a)
<class 'str'>
>>> hex(id(a))
'0xb70d6560'
```

```
>>> a = 5
>>> a
5
>>> type(a)
<class 'int'>
>>> hex(id(a))
'0x83fe540'
>>> a = 'five'
>>> a
'five'
>>> type(a)
<class 'str'>
>>> hex(id(a))
'0xb70d6560'
```

Strong type system: every variable has a type



```
>>> a = 5
>>> a
5
>>> type(a)
<class 'int'>
>>> hex(id(a))
'0x83fe540'
>>> a = 'five'
>>> a
'five'
>>> type(a)
<class 'str'>
>>> hex(id(a))
'0xb70d6560'
```

Dynamic type system: the type changes with the content



Every variable is a reference

```
def echo(a):  
    return a
```

Every variable is a reference

```
def echo(a):  
    return a
```

```
>>> echo(5)  
5
```

```
>>> echo('five')  
'five'
```

What is polymorphism?

```
>>> 5 + 6  
11
```

What is polymorphism?

```
>>> 5 + 6
```

```
11
```

```
>>> 5.5 + 6.6
```

```
12.1
```

What is polymorphism?

```
>>> 5 + 6
```

```
11
```

```
>>> 5.5 + 6.6
```

```
12.1
```

```
>>> "just a" + " string"
```

```
'just a string'
```

What is polymorphism?

```
>>> 5 + 6
11
>>> 5.5 + 6.6
12.1
>>> "just a" + " string"
'just a string'
>>> [1,2,3] + [4,5,6]
[1, 2, 3, 4, 5, 6]
>>> (1,2,3) + (4,5,6)
(1, 2, 3, 4, 5, 6)
```

What is polymorphism?

```
>>> 5 + 6
```

```
11
```

```
>>> 5.5 + 6.6
```

```
12.1
```

```
>>> "just a" + " string"
```

```
'just a string'
```

```
>>> [1,2,3] + [4,5,6]
```

```
[1, 2, 3, 4, 5, 6]
```

```
>>> (1,2,3) + (4,5,6)
```

```
(1, 2, 3, 4, 5, 6)
```

```
>>> {'a':4, 'b':5} + {'c':7}
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +:  
'dict' and 'dict'
```

What is polymorphism?

```
>>> s = "Just a sentence"
>>> len(s)
15
>>> l = [1, 2, 3]
>>> len(l)
3
>>> d = {'a': 1, 'b': 2}
>>> len(d)
2
```

What is polymorphism?

```
>>> s = "Just a sentence"
```

```
>>> len(s)
```

```
15
```

```
>>> l = [1, 2, 3]
```

```
>>> len(l)
```

```
3
```

```
>>> d = {'a': 1, 'b': 2}
```

```
>>> len(d)
```

```
2
```

```
>>> i = 5
```

```
>>> len(i)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: object of type 'int' has no len()
```

What is polymorphism?

```
>>> s.__len__()
```

```
15
```

```
>>> l.__len__()
```

```
3
```

```
>>> d.__len__()
```

```
2
```

What is polymorphism?

```
>>> s.__len__()
```

```
15
```

```
>>> l.__len__()
```

```
3
```

```
>>> d.__len__()
```

```
2
```

```
>>> i.__len__()
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
AttributeError: 'int' object
```

```
has no attribute '__len__'
```

Polymorphism is based on delegation

```
>>> [1,2,3].__add__([4,5,6])  
[1, 2, 3, 4, 5, 6]
```

Polymorphism is based on delegation

```
>>> [1,2,3].__add__([4,5,6])  
[1, 2, 3, 4, 5, 6]  
>>> dir([1,2,3])  
['__add__', '__class__', '__contains__', ...]
```

Polymorphism is based on delegation

```
>>> [1,2,3].__add__([4,5,6])
[1, 2, 3, 4, 5, 6]
>>> dir([1,2,3])
['__add__', '__class__', '__contains__', ...]
>>> 1 in [1,2,3]
True
>>> [1,2,3].__contains__(1)
True
```

Polymorphism is based on delegation

```
>>> [1,2,3].__add__([4,5,6])
[1, 2, 3, 4, 5, 6]
>>> dir([1,2,3])
['__add__', '__class__', '__contains__', ...]
>>> 1 in [1,2,3]
True
>>> [1,2,3].__contains__(1)
True
>>> 6 in [1,2,3]
False
>>> [1,2,3].__contains__(6)
False
```

Polymorphism is based on delegation

```
def sum(a, b):  
    return a + b
```

Polymorphism is based on delegation

```
def sum(a, b):  
    return a + b
```

```
>>> sum(5,6)
```

```
11
```

```
>>> sum("Being ", "polymorphic")  
'Being polymorphic'
```

```
>>> sum([1,2,3], [4,5,6])  
[1, 2, 3, 4, 5, 6]
```

Polymorphism is based on delegation

```
def sum(a, b):  
    return a + b
```

```
>>> sum(5,6)
```

```
11
```

```
>>> sum("Being ", "polymorphic")
```

```
'Being polymorphic'
```

```
>>> sum([1,2,3], [4,5,6])
```

```
[1, 2, 3, 4, 5, 6]
```

```
>>> sum([1,2,3], 8)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 2, in sum
```

```
TypeError: can only concatenate list  
      (not "int") to list
```

1. Create a class that contains an integer as `self.value` and with a `__len__` method that returns the number of digits of the integer. Does `len()` work for instances of this class?
2. Add a `__contains__()` method that returns `True` if `self.value` contains the given digit. Does `'in'` work for this type?
3. Try `str()` on an instance of your class. What happens? How can you return a better string representation (e.g. to show the actual value)?

1. Create a class that contains an integer as `self.value` and with a `__len__` method that returns the number of digits of the integer. Does `len()` work for instances of this class?

```
class CustomInteger:
    def __init__(self, value):
        self.value = value

    def __len__(self):
        return len(str(self.value))
```

2. Add a `__contains__()` method that returns True if `self.value` contains the given digit. Does 'in' work for this type?

```
class CustomInteger:
    def __init__(self, value):
        self.value = value

    def __len__(self):
        return len(str(self.value))

    def __contains__(self, digit):
        return str(digit) in str(self.value)
```

3. Try `str()` on an instance of your class. What happens? How can you return a better string representation (e.g. to show the actual value)?

```
class CustomInteger:
    def __init__(self, value):
        self.value = value

    def __len__(self):
        return len(str(self.value))

    def __contains__(self, digit):
        return str(digit) in str(self.value)

    def __str__(self):
        return super().__str__() + ' [{}]' .format(self.value)
```

```
class Room:
    def __init__(self, door):
        self.door = door

    def open(self):
        self.door.open()

    def close(self):
        self.door.close()

    def is_open(self):
        return self.door.is_open()
```

```
class Door:
    def __init__(self):
        self.status = "closed"

    def open(self):
        self.status = "open"

    def close(self):
        self.status = "closed"

    def is_open(self):
        return self.status == "open"
```

```
class BooleanDoor:
    def __init__(self):
        self.status = True

    def open(self):
        self.status = True

    def close(self):
        self.status = False

    def is_open(self):
        return self.status
```

```
>>> door = Door()  
>>> bool_door = BooleanDoor()
```

Polymorphism in action

```
>>> door = Door()  
>>> bool_door = BooleanDoor()  
>>> room = Room(door)  
>>> bool_room = Room(bool_door)
```

Polymorphism in action

```
>>> door = Door()
>>> bool_door = BooleanDoor()
>>> room = Room(door)
>>> bool_room = Room(bool_door)
>>> room.open()
>>> room.is_open()
True
>>> room.close()
>>> room.is_open()
False
```

```
>>> door = Door()
>>> bool_door = BooleanDoor()
>>> room = Room(door)
>>> bool_room = Room(bool_door)
>>> room.open()
>>> room.is_open()
True
>>> room.close()
>>> room.is_open()
False
>>> bool_room.open()
>>> bool_room.is_open()
True
>>> bool_room.close()
>>> bool_room.is_open()
False
```

1. Modify the Room, the Door and the BooleanDoor classes to add a toggle() method that changes open status to closed and vice versa.
2. Change the Room class such that it accepts a class instead of an instance. How do you manage it into __init__()? Does it still work?
3. Create a PetDoor object that inherits from Door (or BooleanDoor). Add the suitable methods to manage the small pet door. Does the Room work?
4. May you reuse the Door to create the pet door?

1. Modify the Room, the Door and the BooleanDoor classes to add a `toggle()` method that changes open status to closed and vice versa.

```
class Room:
    def __init__(self, door):
        self.door = door

    def open(self):
        self.door.open()

    def close(self):
        self.door.close()

    def is_open(self):
        return self.door.is_open()

    def toggle(self):
        if self.door.is_open():
            self.door.close()
        else:
            self.door.open()
```

2. Change the Room class such that it accepts a class instead of an instance. How do you manage it into `__init__()`? Does it still work?

```
class Room:
    def __init__(self, door_cls):
        self.door = door_cls()

    def open(self):
        self.door.open()

    def close(self):
        self.door.close()

    def is_open(self):
        return self.door.is_open()

>>> room = Room(Door)
>>> bool_room = Room(BooleanDoor)
```

3. Create a PetDoor object that inherits from Door (or BooleanDoor). Add the suitable methods to manage the small pet door. Does the Room work?

```
class PetDoor(BooleanDoor):  
    def __init__(self):  
        super().__init__()  
        self.pet_door = "open"  
  
    def open_pet_door(self):  
        self.pet_door = "open"  
  
    def close_pet_door(self):  
        self.pet_door = "closed"
```

4. May you reuse the Door to create the pet door?

```
class PetDoor(BooleanDoor):  
    def __init__(self):  
        super().__init__()  
        self.pet_door = BooleanDoor()  
  
    def open_pet_door(self):  
        self.pet_door.open()  
  
    def close_pet_door(self):  
        self.pet_door.close()
```

It has the attribute



```
if hasattr(someobj, 'open'):
    [...]
else:
    [...]
```

It behaves like it has
the attribute



```
try:  
    someobj.open()  
    [...]  
except AttributeError:  
    [...]
```

CAST

A lot of people
provided free information and code

Font awesome icons
by Freepik

Python 3 OOP series
<http://lgiordani.com/blog/categories/python3/>

Some links about Python OOP
<http://goo.gl/UBdJDT>

Dive into Object-oriented Python

DjangoCon Europe 2015 – Cardiff, Wales



Leonardo Giordani
lgiordani.com

