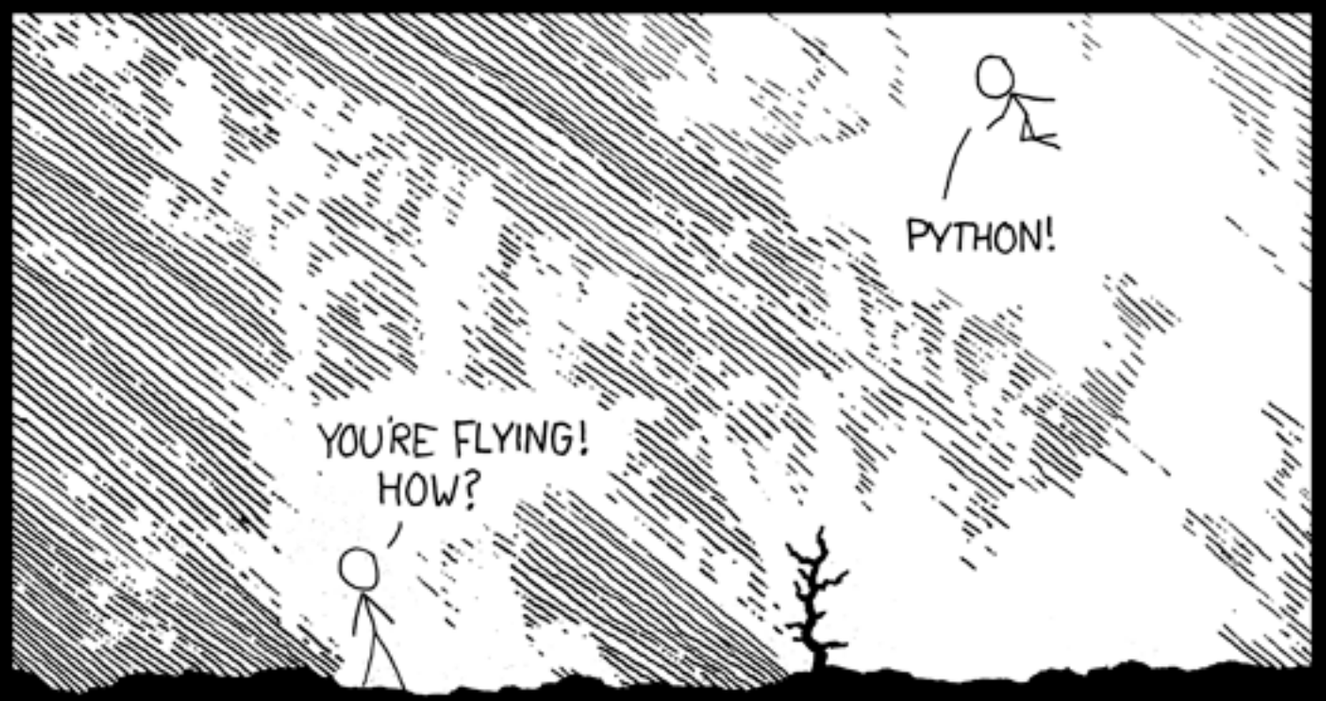
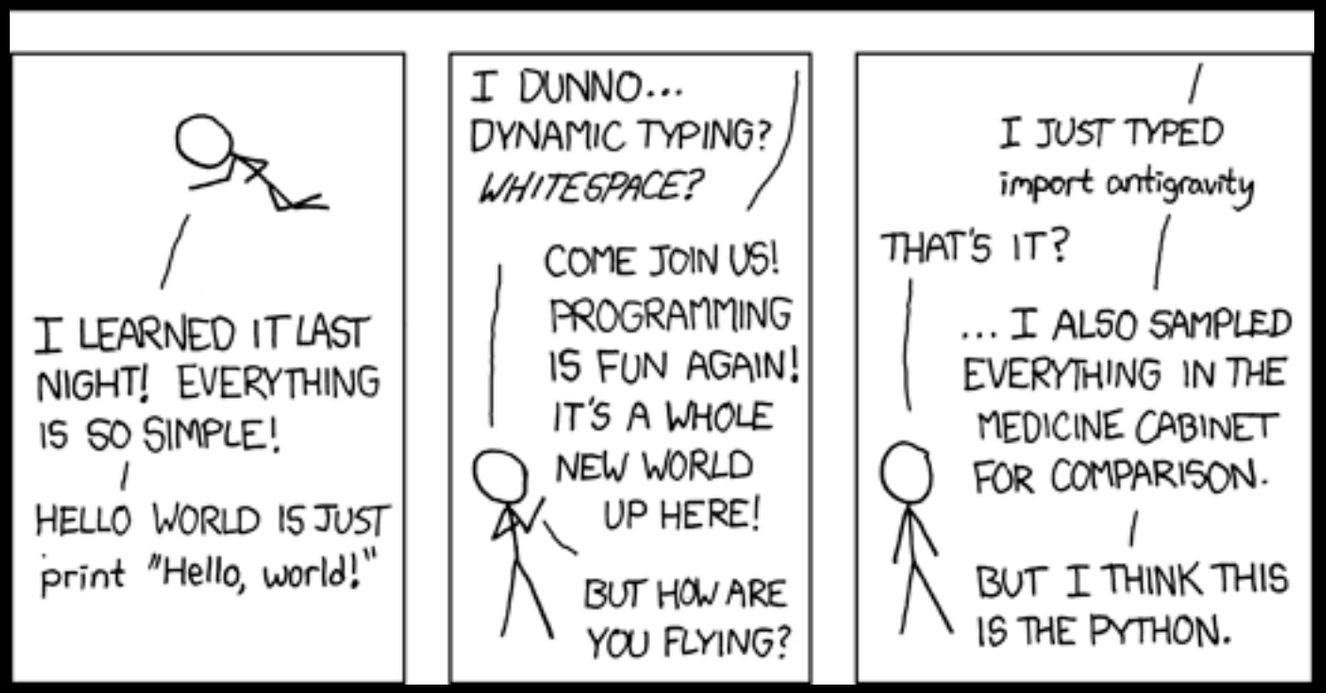
Object oriented Python

Nicholas Beshouri — nbeshouri@gmail.com


PYTHON!
YOU'RE FLYING! HOW?


I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

What is an object?

What is an object?

- It has attributes.
- It has code that can operate on those (and other) attributes.

```
dragon = {
    'id': 83948,
    'npc_class': 'dragon',
    'name': 'Oliver',
    'hit_points': 100,
    'attack_power': 10,
    'position': [343.34, 89.33]
}


goat = {
    'id': 13452,
    'npc_class': 'ruminant',
    'name': 'James',
    'hit_points': 1,
    'attack_power': 0,
    'position': [99.81, 91.45]
}


def move_to(npc, position):
    ...

# Note: Only use for dragon's. Behavior
# undefined for goats.
def breath_fire(dragon, target):
    ...
```

```
goat = Goat()
dragon = Dragon()

dragon.eat(goat)
```

“...limiting your project to C means that people don't screw that up, and also means that you get a lot of programmers that do actually understand low-level issues and don't screw things up with any idiotic "object model" crap.”

–Linus Torvalds, whom you can also blame/thank for git

Why non-torvalds might want to use objects

- Encapsulation: Easy to hide complexity and present a clean interface.
- Inheritance: Easy to reuse existing code by subclassing existing classes.
- Instantiation: Easy to create and manage slightly different copies of the same thing.
- Intuitive at some human level, which makes code easier to read and write.

Objects in Python

Everything is an object.

```
In [54]: "Python".__add__('ic')
Out[54]: 'Pythonic'
```

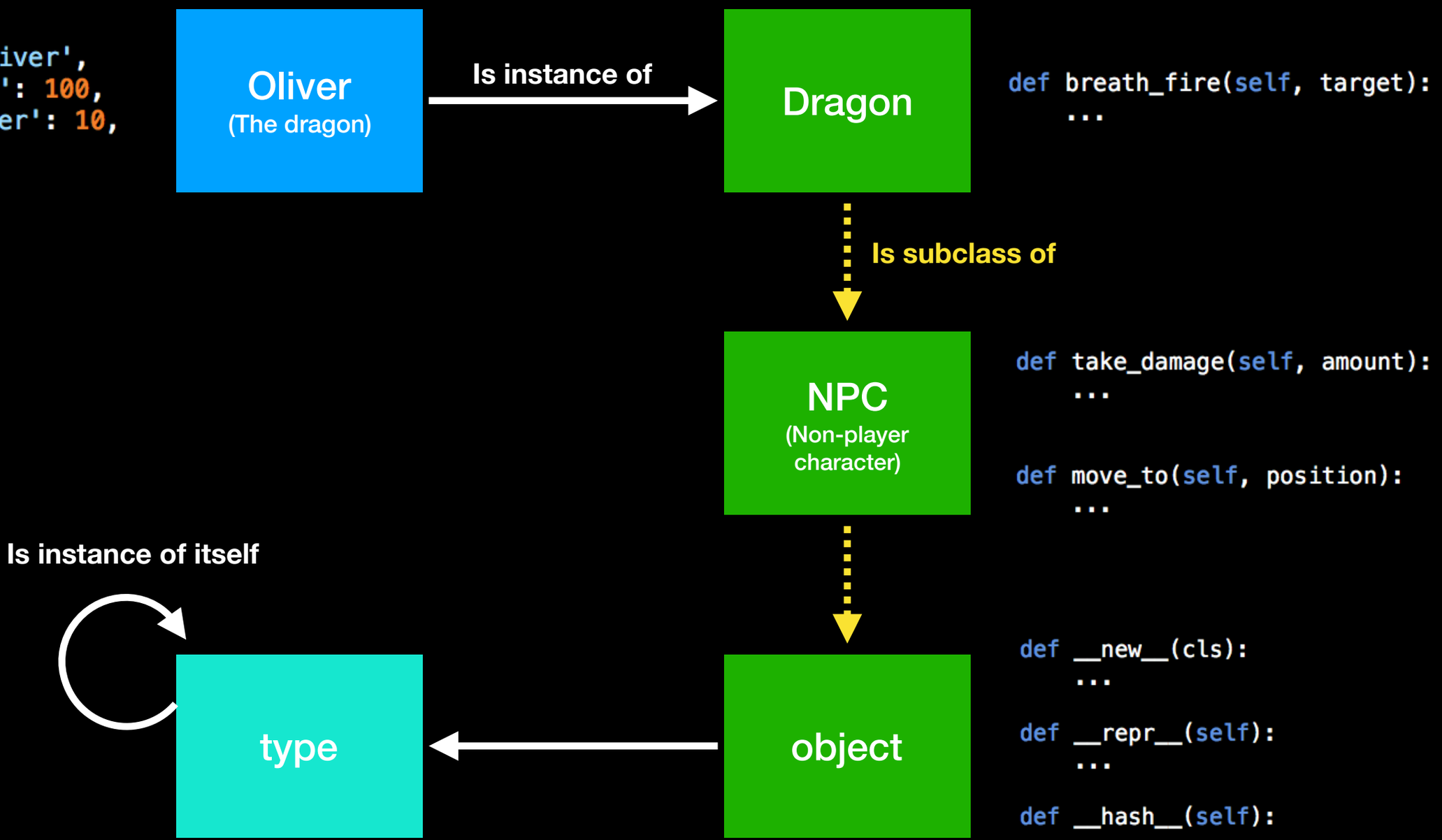
```
In [66]: number = 9
    ...: number.__str__()
    ...:
Out[66]: '9'

In [67]: number.__add__(7)
Out[67]: 16
```

But how does it work?

Class based

```
{
    'name': 'Oliver',
    'hit_points': 100,
    'attack_power': 10,
    ...
}
```




```
def breath_fire(self, target):
    ...
```

```
def take_damage(self, amount):
    ...

def move_to(self, position):
    ...
```

```
def __new__(cls):
    ...

def __repr__(self):
    ...

def __hash__(self):
    ...
```

The `class` statement

```
class Dragon(NPC):

    # This is a class variable. It lives in the class's
    # __dict__ and there's only ever one version.
    _all_dragons = []

    def __init__(self, name):
        # Call the __init__ the super class, if needed.
        super().__init__(name)  # super(NPC, self) in Python 2.

        # Declare some instance variables. Unlike class variables
        # each instance has its own copy.
        self._hit_points = 100
        self._attack_power = 10

        self._all_dragons.add(self)

    # Instance methods access the instance via the self parameter.
    def eat(self, target_npc):
        ...

    # If you implement a method with the same name as one on the
    # parent class, the parent class's version will be "overriden"
    # by that method.
    def move_to(self, target):
        super().move_to(position)
        # Do dragon specific stuff.
        ...

    # Classmethods only have direct access to the class.
    @classmethod
    def kill_all_dragons(cls):
        for dragon in cls._all_dragons:
            dragon.kill()

    # Staticmethods don't have access to the class or the instance.
    @staticmethod
    def _calculate_flame_vector(target_point):
        ...
```

Properties

```
class NPC(object):

    def __init__(self, name):
        self.name = name
        self._hit_points = 1
        self._attack_power = 0

    # Properties can be accessed like attributes, but you have
    # more control.
    @property
    def hit_points(self):
        return self._hit_points

    @hit_points.setter
    def hit_points(self, value):
        self._hit_points = max(0, value)
        if self._hit_points == 0:
            self.die()

    def move_to(self, position):
        ...
```

self

self

```
class Dragon(NPC):

    def breath_fire(self, target):
        ...
```

```
oliver = Dragon('Oliver')
oliver.breath_fire(an_goat)
```

~~oliver = Dragon('Oliver')~~
~~oliver.breath_fire(oliver, an_goat)~~

Method lookup

- When you call `oliver.breath_fire(target)`, Python *looks in Oliver's instance __dict__ for an attribute named "breath_fire".
- When it doesn't find one, it checks Dragon's class `__dict__` and finds a function with the signature `breath_fire(self, target)`.
- Then it checks to see if `breath_fire(self)` is a descriptor, which means it has a `__get__(instance, cls)` method.
- Functions objects have this method, so Python returns the result of calling `__get__(instance, cls)` on `breath_fire(self)` instead of the original function.
- This returns a bound method, `breath_fire()`, which wraps the original function and binds its first parameter to the instance in `__get__(instance, cls)`.
- Having this layer of indirection allows objects to customize what gets returned when they're looked up as attributes of other objects, enabling `properties`, `staticmethods`, `classmethods`, or anything you want to create.

***Technically, it checks Dragon first to look for overriding descriptors—see appendix.**

But couldn't Python just add `self` for me in the class definition?

self

Unfortunately, there's no way for Python to know if you intend a function to have a reference to the instance.

```
class Foo:

@decorator_with_unknown_effects
def bar():
    pass
```

See Guido's discussion of this topic at http://neopythonic.blogspot.com/2008/10/why-explicit-self-has-to-stay.html

self

```
class Foo:

    decorator_with_unknown_effects = staticmethod
    @decorator_with_unknown_effects
    def bar():
        pass
```

See Guido's discussion of this topic at http://neopythonic.blogspot.com/2008/10/why-explicit-self-has-to-stay.html

self

```
class StaticMethod:

    def __init__(self, method):
        self._method = method

    def __get__(self, obj, obj_class):
        return self._method


class Foo:

    @StaticMethod
    def bar():
        pass
```

See: http://neopythonic.blogspot.com/2008/10/why-explicit-self-has-to-stay.html

Why do we need `super`?

Why do I need to call super?

Why can't I just call the base class directly?

```
class Dragon(NPC):
    def __init__(self, name):
        NPC.__init__(self, name)
```

Multiple inheritance

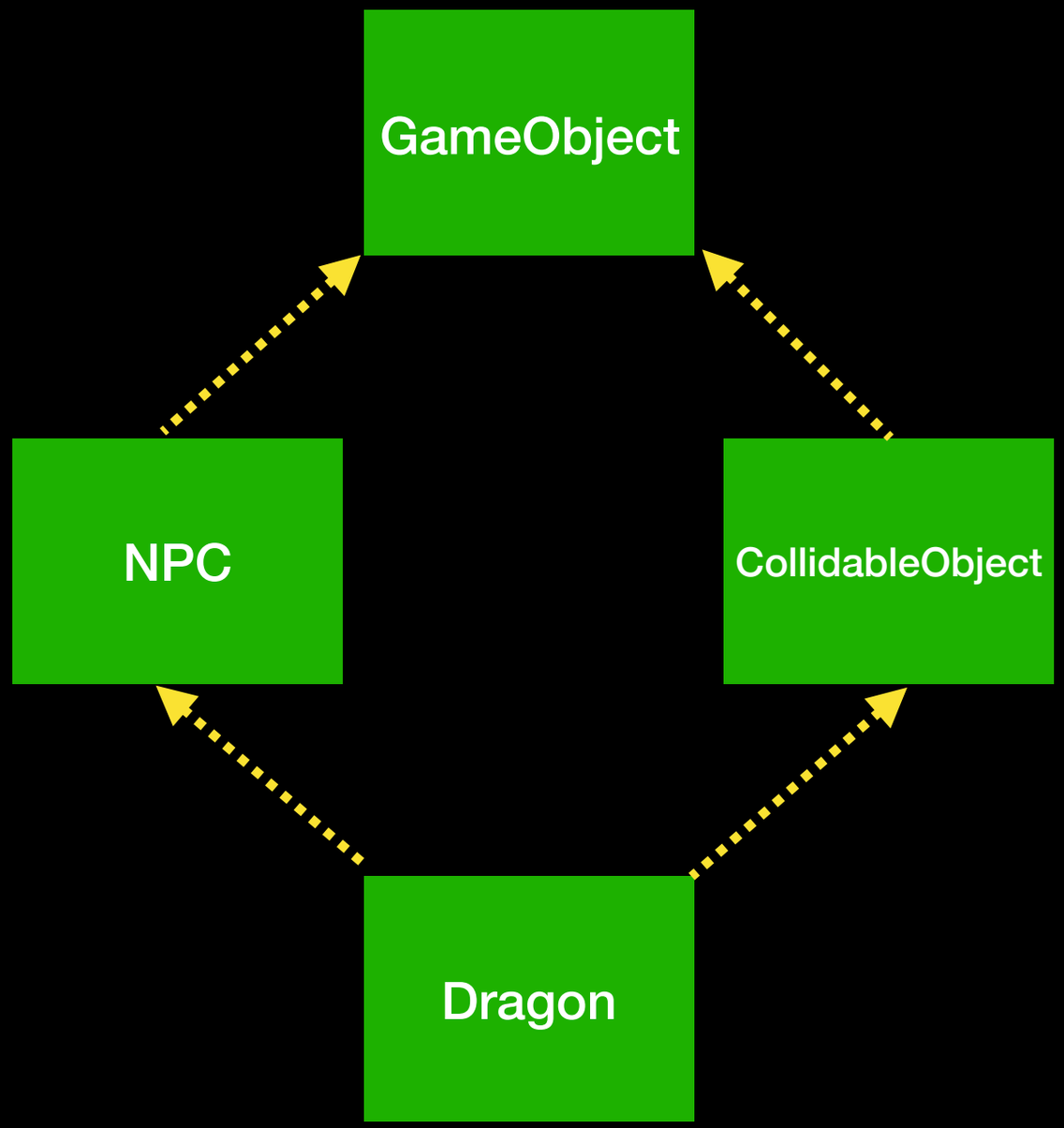
This becomes more complicated when you've got more than one base class.

```
class Dragon(NPC, CollidableObject):
    def __init__(self, name):
        NPC.__init__(self, name)
        CollidableObject.__init__(self)
```

Dimond inheritance

And really complicated when you've got diamond inheritance.

Here the naive implementation below will call GameObject's `__init__` method twice.

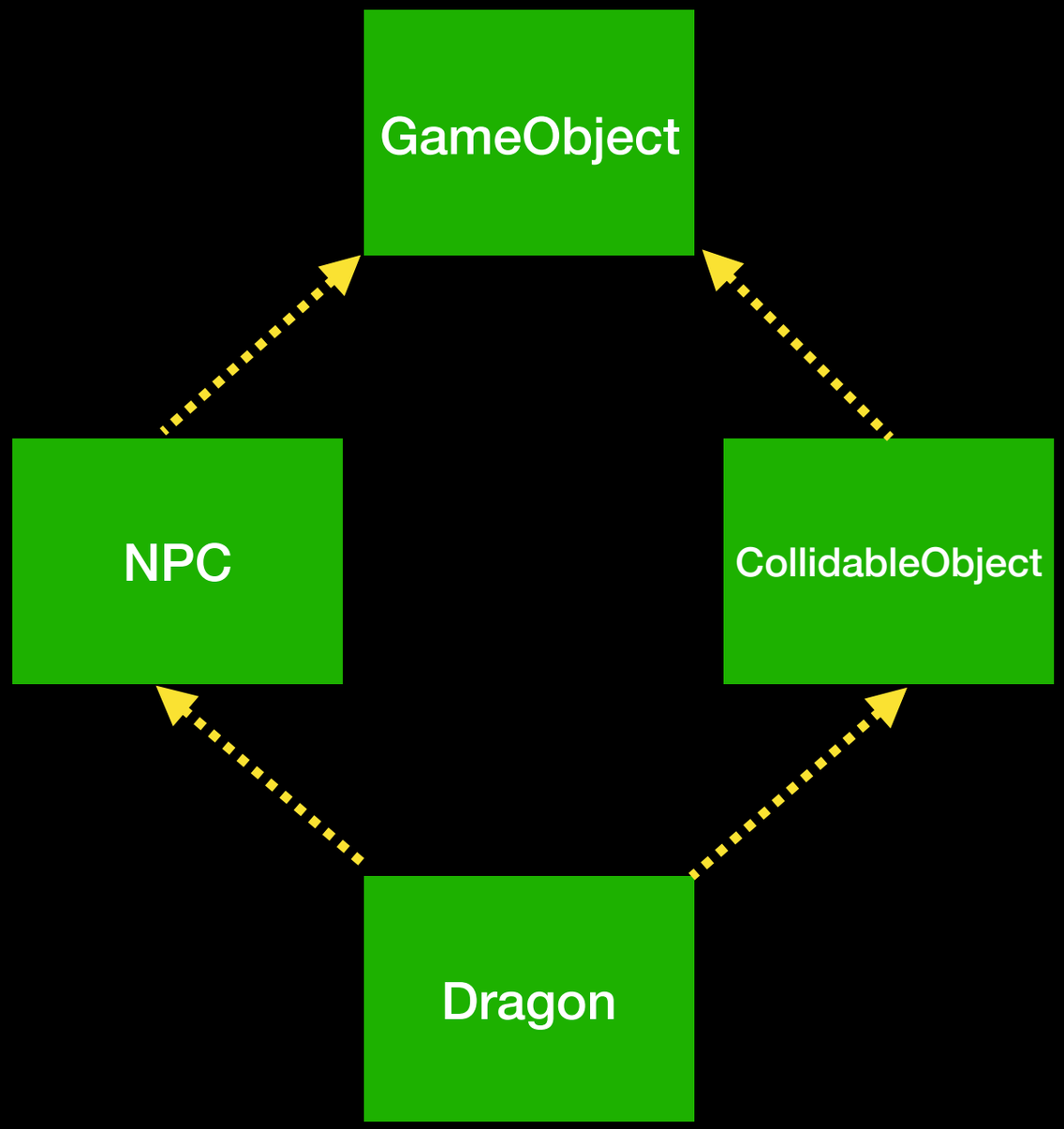



```
class Dragon(NPC, CollidableObject):

    def __init__(self, name):
        NPC.__init__(self, name)
        CollidableObject.__init__(self)
```

Dimond inheritance

Super solves this problem by walking up the MRO in a linear way.




```
class Dragon(NPC, CollidableObject):

    def __init__(self, name):
        # `super(cls, object)` will walk up the MRO
        # in a consistent linear way, searching
        # for the requested attribute. The search
        # starts after `cls` in the MRO of
        # `type(object)`. And remember that `object`
        # is the same instance for all these calls,
        # so the MRO is always the same.
        super(Dragon, self).__init__(name)
```

```
Dragon.mro() == [Dragon, NPC, CollidableObject, GameObject]
```

And one more thing...

“There is a big difference between a coherent passage of writing and a flaunting of one’s erudition, a running journal of one’s thoughts, or a published version of one’s notes. A coherent text is a designed object: an ordered tree of sections within sections, crisscrossed by arcs that track topics, points, actors, and themes, and held together by connectors that tie one proposition to the next. Like other designed objects, it comes about not by accident but by drafting a blueprint, attending to details, and maintaining a sense of harmony and balance.”

–Steven Pinker, in *A Sense of Style*

```
In [51]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Appendix

Attribute lookups on instances

When you use the syntax *x.name* to refer to an attribute of instance *x* of class *C*, the lookup proceeds in three steps:

1. When *'name'* is found in *C* (or in one of *C*'s ancestor classes) as the name of an overriding descriptor *v* (i.e., `type(v)` supplies methods `__get__` and `__set__`)

   - The value of *x.name* is the result of `type(v).__get__(v, x, C)`

2. Otherwise, when *'name'* is a key in `x.__dict__`

   - *x.name* fetches and returns the value at `x.__dict__['name']`

3. Otherwise, *x.name* delegates the lookup to *x*'s class (according to the same two-step lookup used for *C.name*, as just detailed)

   - When a descriptor *v* is found, the overall result of the attribute lookup is, again, `type(v).__get__(v, x, C)`
   - When a nondescriptor value *v* is found, the overall result of the attribute lookup is just *v*

This behavior is defined by `__getattribute__(self, name)` on the object's class.

Source: *Python in a Nutshell* p. 109

Attribute lookups on classes

When you use the syntax *C.name* to refer to an attribute on a class object *C*, the lookup proceeds in two steps:

1. When *'name'* is a key in `C.__dict__`, *C.name* fetches the value *v* from `C.__dict__['name']`. Then, when *v* is a descriptor (i.e., `type(v)` supplies a method named `__get__`), the value of *C.name* is the result of calling `type(v).__get__(v, None, C)`. When *v* is not a descriptor, the value of *C.name* is *v*.

2. When *'name'* is *not* a key in `C.__dict__`, *C.name* delegates the lookup to *C*'s base classes, meaning it loops on *C*'s ancestor classes and tries the *name* lookup on each (in *method resolution order*, as covered in "Method resolution order" on page 113).

This behavior is defined by `__getattribute__(self, name)` on the class's metaclass.