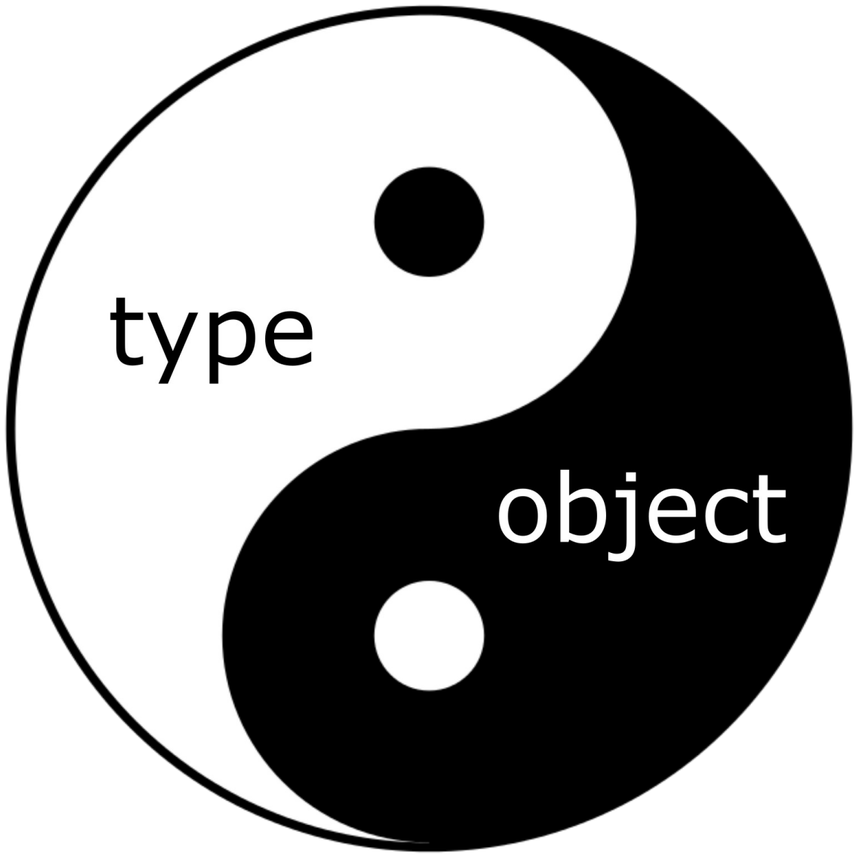
The Tao of Python

The intricate relationship between ”object” and ”type” and how metaclasses, classes and instances are related



About me

- Diploma Biology
- MSc Nanosciences and Nanotechnologies (VB .NET)
- PhD in Computer Simulations in Complex Networks (C#)
- Postdoc Researcher in Systems Biology (Heidelberg, Germany): "Constraint based modelling in biological networks" (Java)
- Postdoc Researcher in Bioinformatics (London): "Computer simulations and mathematical programming in biological networks" (Python)
- Now: Full stack web development with Django/DRF/Polymer/Web Components (Python, ES6, HTML, CSS)

Contents of this talk

- Object-oriented relationships
- Relationship rules
- What is a Python object?
- Classes as objects
- Metaclasses
- What is `type`?
- What is `object`?
- How are `type` and `object` related?
- The Python objects map

Most of this talk is based on this article by Shalabh Chaturvedi.

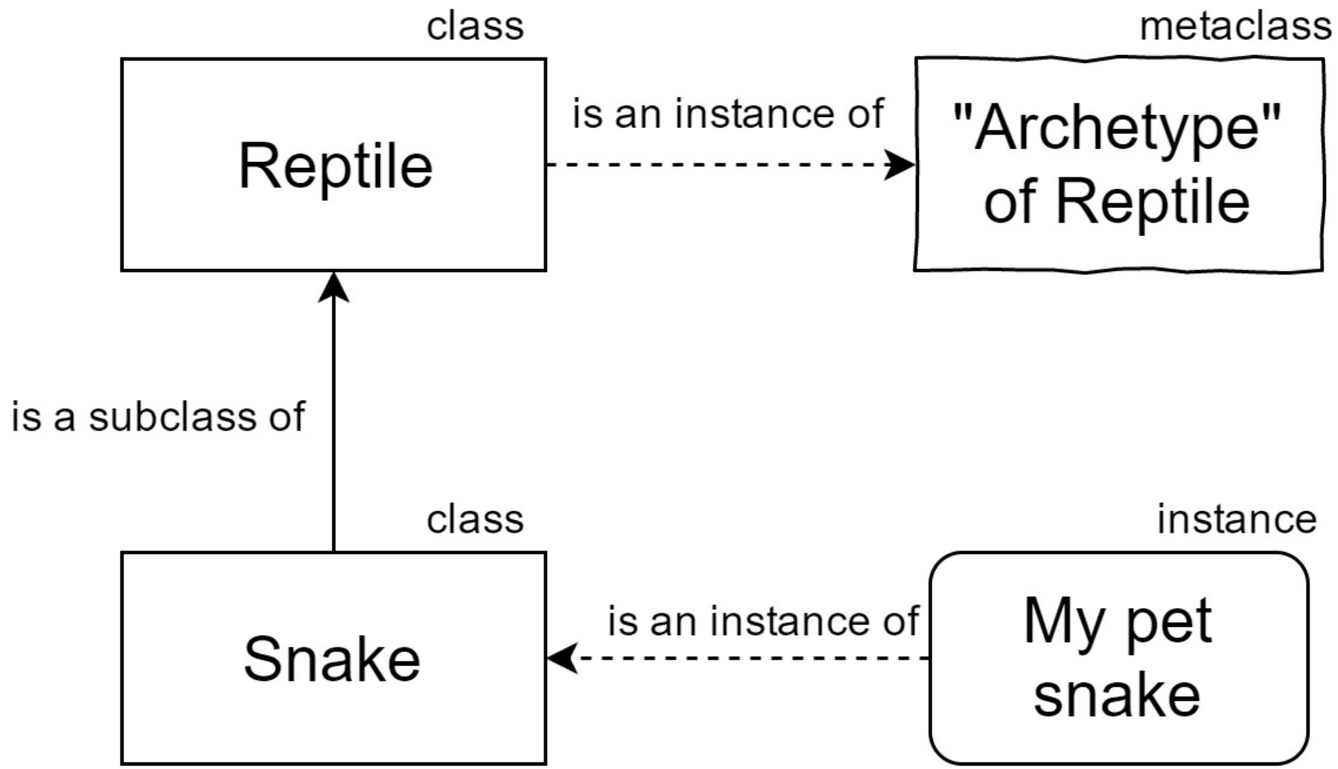
Why is this talk useful?

- Actually it's not terribly useful
- Deep understanding of the Python object model
- Clarification of the role and behavior of classes, metaclasses and instances
- Appreciation of the language on different level
- Zen-like satisfying moment of understanding
- Bragging rights :D

Object-oriented relationships

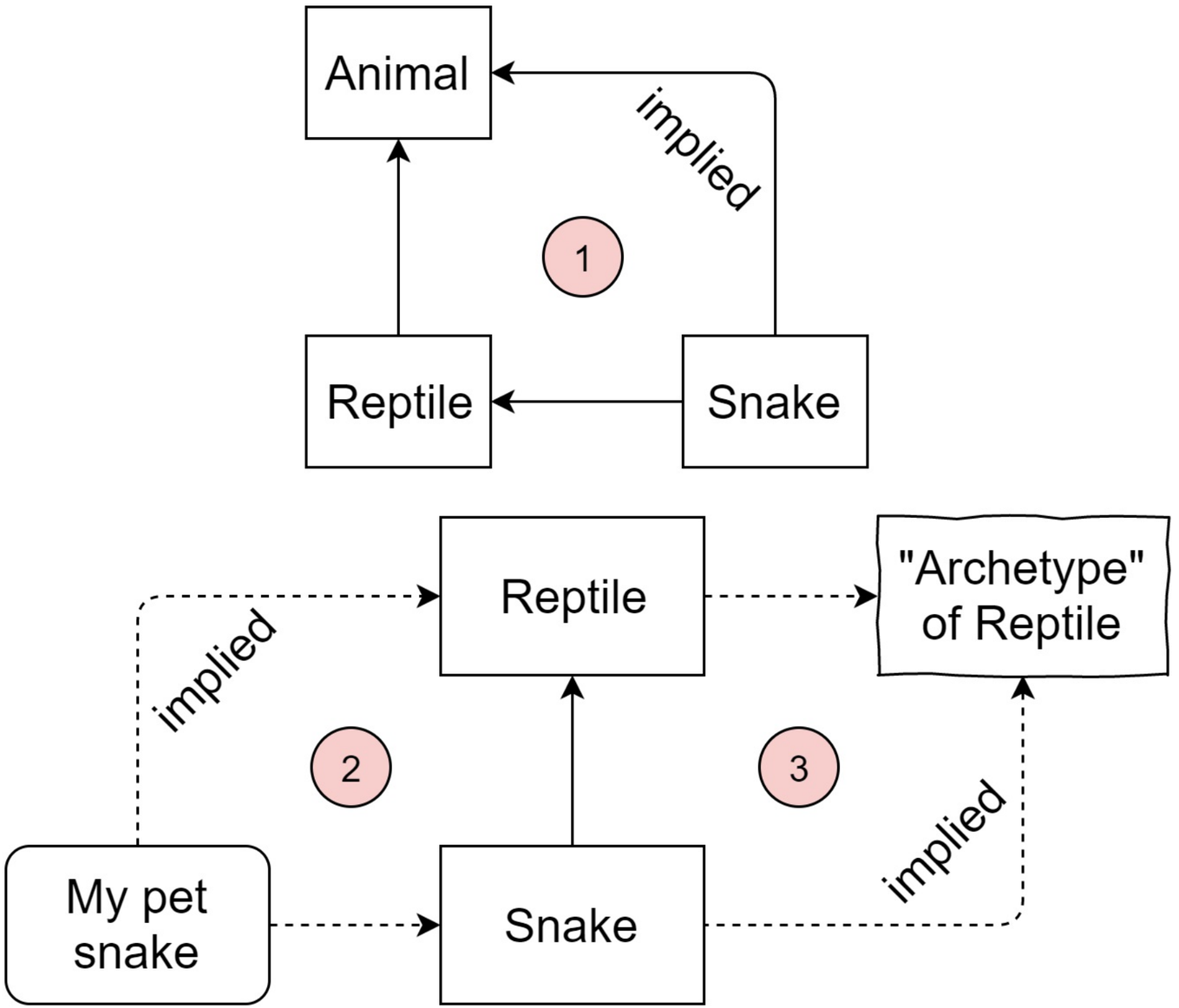
While we introduce many different objects, we only use two kinds of relationships:

1. ***is a kind of*** (solid line): Also known as **specialization** or **inheritance**, this relationship exists between two objects when one (the subclass) is a specialized version of the other (the superclass). A snake is *a kind of* reptile. It has all the traits of a reptile and some specific traits which identify a snake. Terms used: *subclass of*, *superclass of*, *superclass-subclass* or simply *is a*.
2. ***is an instance of*** (dashed line): Also known as **instantiation**, this relationship exists between two objects when one (the instance) is a concrete example of what the other specifies (the type). I have a pet snake named Squasher. Squasher is an instance of a snake. Terms used: *instance of*, *type of*.


class
Reptile
is an instance of
metaclass
"Archetype"
of Reptile
is a subclass of
class
Snake
is an instance of
instance
My pet
snake

Relationship rules

1. If A is a subclass of B, and B is a subclass of C, then A is a subclass of C.
2. If X is an instance of A, and A is a subclass of B, then X is an instance of B.
3. If B is an instance of M, and A is a subclass of B, then A is an instance of M.


Animal
implied
1
Reptile
Snake
Reptile
"Archetype"
of Reptile
implied
2
3
implied
My pet
snake
Snake

What is a Python object?

An object is an entity with the following characteristic properties:

1. *Identity* (i.e. given two names we can say for sure if they refer to one and the same object, or not).
2. A *value* - which may include a bunch of attributes (i.e. we can reach other objects through `objectname.attributename`).
3. A *type* - every object has exactly one *type*. For instance, the object `2` has a type `int` and the object `"joe"` has a type `string`.
4. One or more *bases*. A base is similar to a super-class or base-class in object-oriented lingo.

In [1]:
```
two = 2
print(type(two))
```

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

In [2]:
```
print(type(type(two)))
```

```
<class 'type'>
```

In [3]:
```
print(type(two).__bases__)
```

```
(<class 'object'>,)
```

In [4]:
```
print(dir(two))
```

```
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmo
d__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattrib
ute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__int__', '__invert__', '__le_
_', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '_
_pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloor
div__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshi
ft__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subcla
sshook__', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_byt
es', 'imag', 'numerator', 'real', 'to_bytes']
```

Rule 1: Everything is an object

And I mean **everything**. Even things that are "primitive types" in other languages.

- You can store them in variables
- You can pass them as parameters to functions
- You can return them as the result of functions
- You can construct them at runtime

And more importantly: You can treat *every* programming construct in a **uniform** and **consistent** way

Functions as objects

- When you use the keyword `def`, Python creates a function object.
- Functions can be passed around as arguments to other functions.
- These functions that take other functions as arguments are called **higher order functions**.
- e.g. the `map` function takes a function and an iterable and applies the function to each item in the iterable.

Classes as objects

1. When you use the keyword `class`, Python executes it and creates an object.
2. This object (the *class*) is itself capable of **creating objects** (the *instances*), and this is why it's a class.
3. Since classes are objects, they must be generated by something, this is *metaclasses*.
4. Since metaclasses objects, they must also be generated by something, this is **again** *metaclasses*.

Therefore: Objects are instances of classes, classes are instances of metaclasses and metaclasses are instances of themselves.

Metaclasses (these are objects too!)

1. Metaclasses are the "stuff" that creates classes.
2. You define classes in order to create objects, right?
3. We learned that Python classes are objects.
4. Well, metaclasses are what creates these objects. They are the classes' classes, you can picture them this way:
   - `MyClass = MetaClass()`
   - `MyObject = MyClass()`

> *Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).* ***Tim Peters***

What is `type`?

1. Remember the function `type`? The good old function that lets you know what type an object is.
2. `type` can also create classes on the fly. `type` can take the description of a class as parameters, and return a class as `type(name, bases, dct)`.

    - `name` is a string giving the name of the class to be constructed.
    - `bases` is a tuple giving the parent classes of the class to be constructed.
    - `dct` is a dictionary of the attributes and methods of the class to be constructed.
3. Why the heck is it written in lowercase, and not `Type`? Consistency with `str`, the class that creates strings objects, and `int` the class that creates integer objects. `type` is just the class that creates `class` objects.

In [5]:
```
class A:
    pass
a = A()
print(type(a))
print(type(A))
print(A.__bases__)
```

```
<class '__main__.A'>
<class 'type'>
(<class 'object'>,)
```

In [6]:
```
A = type('A', (), {})
a = A()
print(type(a))
print(type(A))
print(A.__bases__)
print(isinstance(a, A), isinstance(a, object), issubclass(A, object))
```

```
<class '__main__.A'>
<class 'type'>
(<class 'object'>,)
True True True
```

In [7]:
```
def f():
    """My name is f."""
    pass
print(type(f))
print(type(type(f)))
print(type(f).__bases__)
print(f.__doc__)
```

```
<class 'function'>
<class 'type'>
(<class 'object'>,)
My name is f.
```

The power of `type`

1. Everything is an object in Python, and they are all either instances of classes or instances of metaclasses.
2. `type` is the metaclass Python uses to create (i.e. *instantiate*) all classes and metaclasses, **including** `type` itself.
3. `type` is actually its own metaclass. This is not something you could reproduce in pure Python, and is done by cheating a little bit at the implementation level.

What is **object**?

1. object is the class that **all** classes inherit from.
2. All classes **including** object are subclasses of themselves.
3. All classes **including** object are subclasses of object. object.__bases__ is an empty tuple.
4. All classes **except** object will have object in __bases__ in a class in their inheritance hierarchy.

Kinds of objects

- There are two kinds of objects in Python:
  1. *Type objects* - can create instances, can be subclassed. e.g. `type`, `object`, `int`, `str`, `list`.
  2. *Non-type objects* - cannot create instances, cannot be subclassed. e.g. `1`, `"hello"`, `[1, 2, 3]`.
- `type` and `object` are two primitive objects of the system.
- `objectname.__class__` exists for every object and points the type of the object.
- `objectname.__bases__` exists for every type object and points the superclasses of the object. It is empty only for `object`.


Okay, any ideas on how to make women more interested in us?
Java
C++
python
Make more exceptions?
C++
Redefine our methods?
python
Stop treating them like objects?
ANSI
C
szym1000 / 9GAG
Java
ANSI
C

Recap

1. All classes and metaclasses **including** `object` are *subclasses* of `object`.
2. All classes and metaclasses **including** `type` are *instances* of `type`.
3. All objects **including** `object` are *instances* of `object`.

In [8]:
```
issubclass(type, object) # Recap rule #1
```

Out[8]:
```
True
```

In [9]:
```
issubclass(object, object) # Recap rule #1
```

Out[9]:
```
True
```

In [10]:
```
issubclass(object, type) # Recap rule #1
```

Out[10]:
```
False
```

In [11]:
```
isinstance(object, type) # Recap rule #2
```

Out[11]:
```
True
```

In [12]:
```
isinstance(type, type) # Recap rule #2
```

Out[12]:
```
True
```

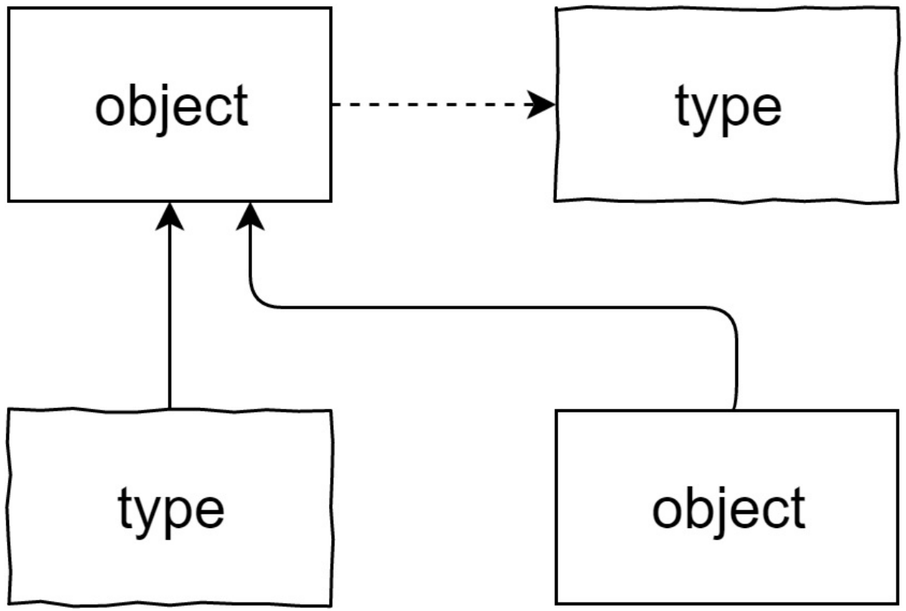
In [13]:
```
isinstance(type, object) # Recap rule #3
```

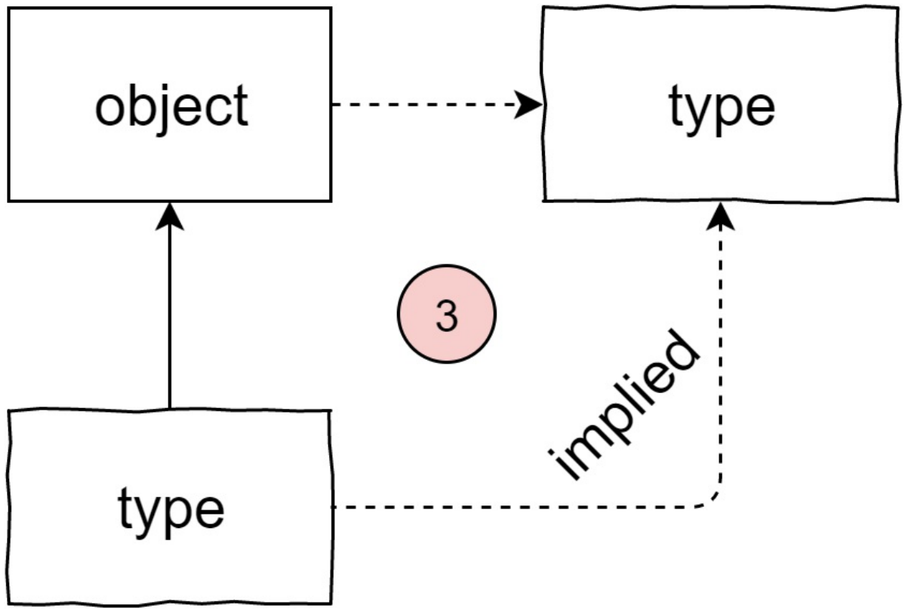
Out[13]:
```
True
```

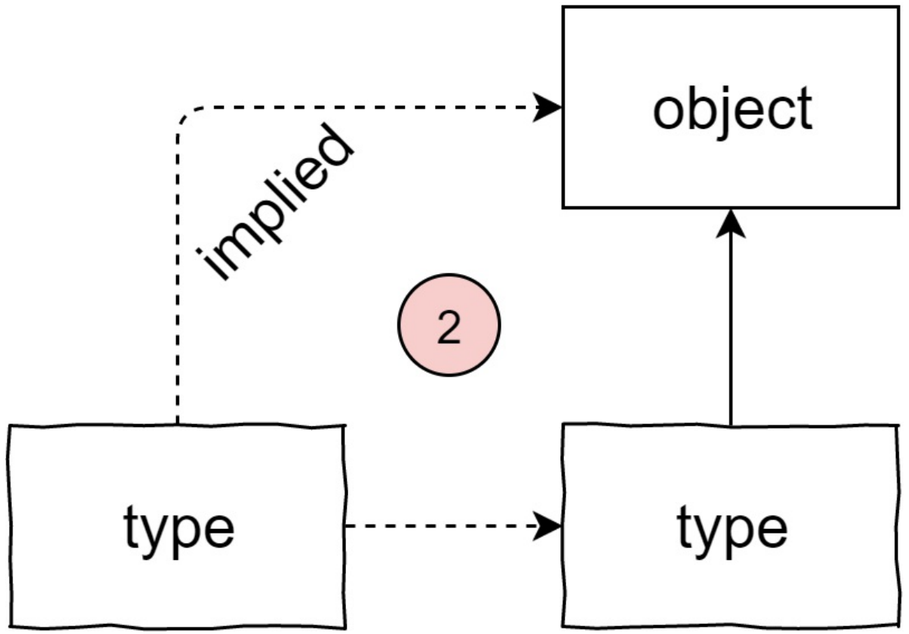
In [14]:
```
isinstance(object, object) # Recap rule #3
```

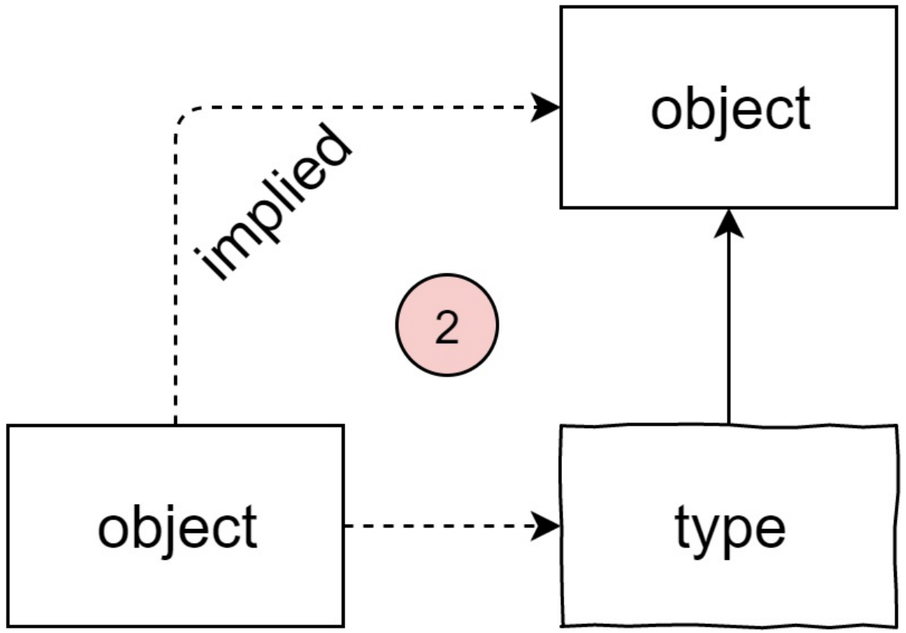
Out[14]:
```
True
```

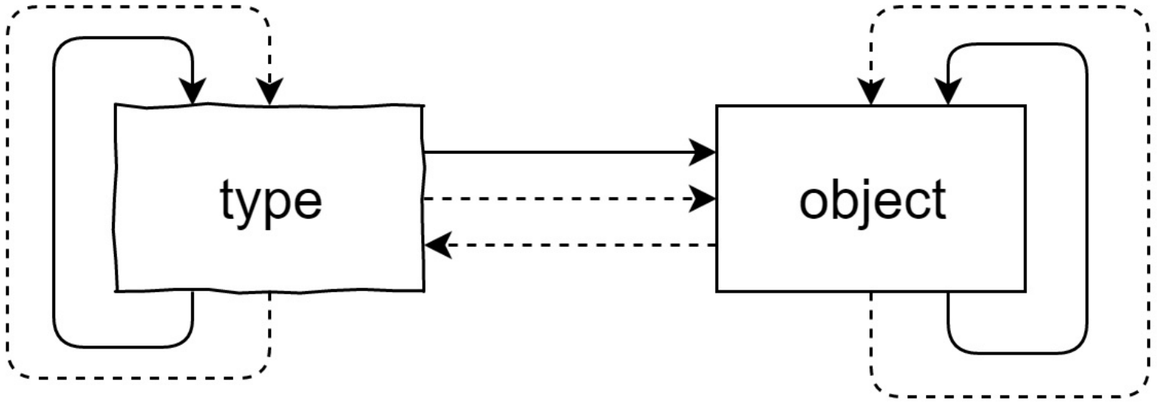

DAFUQ?
memegenerator.net


object
type
type
object


object
type
3
implied
type

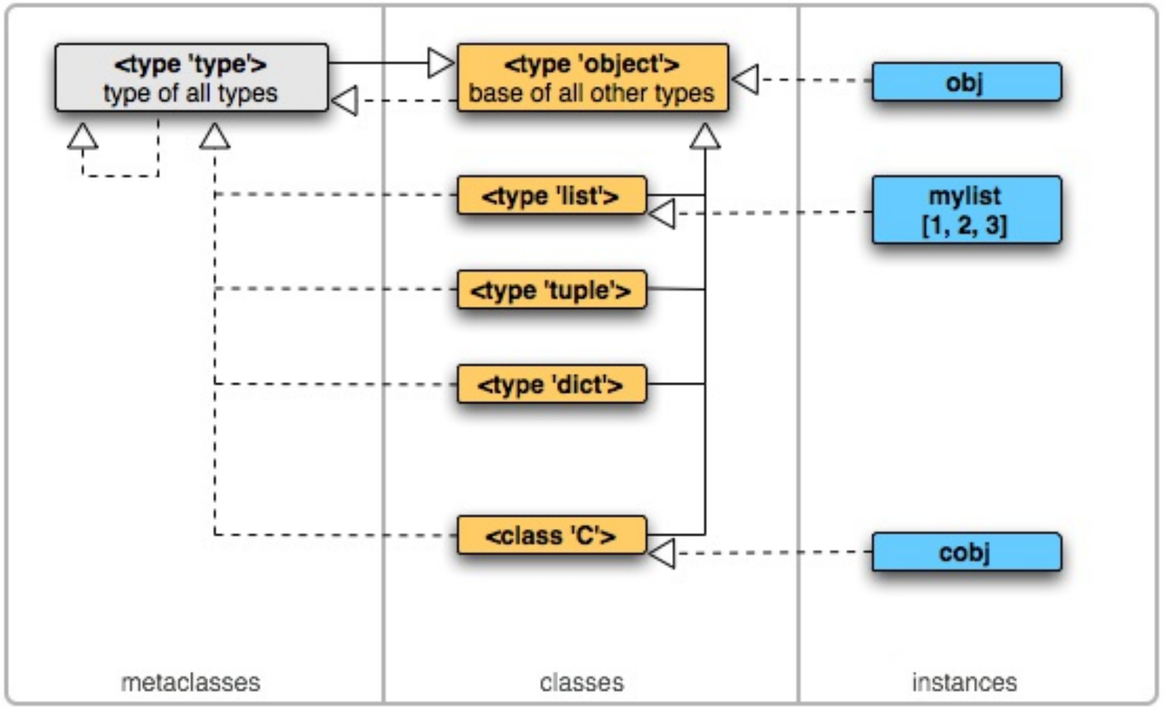

object
implied
2
type
type


object
implied
2
object
type

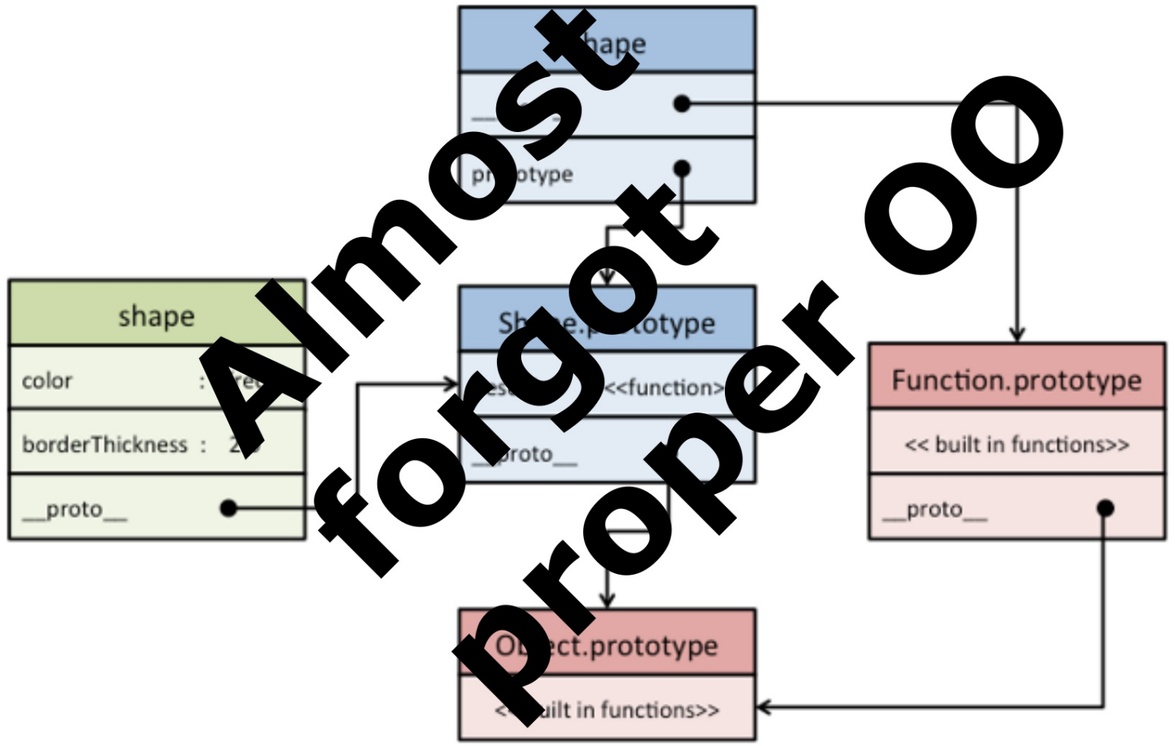


In [15]:
```
from IPython.display import Image, display
display(Image(url='figures/mind_blown.gif', width=400))
```

The Python Objects Map



Rant: JavaScript



Almost forgot proper OO

Rant: Ruby

Ruby inherited the Perl philosophy of having more than one way to do the same thing. I inherited that philosophy from Larry Wall, who is my hero actually. Y. Matsumoto, creator of Ruby.

Matz chose to sacrifice first-class functions just so he could make parentheses ***optional****.*

Methods are a fundamental part of Ruby's syntax, but they are not values that Ruby programs can operate on. That is, ***Ruby's methods are not objects*** *in the way that strings, numbers, and arrays are. From [The Ruby Programming Language Book](#).*

Anonymous 12/26/14(Fri)19:51:11 No.45793352 >>45793379 >>45794765 >>45794788 >>45796081 >>45799058 >>45800174

why the fuck start ups only want ruby on rails developers ?

Anonymous 12/26/14(Fri)22:01:48 No.45794867 >>45794890 >>45798725 >>45798760 >>45799182 >>45799190 >>45799231 >>45800065

It's hipster technology. They create an internet startup with Rails or Node.js on top of MongoDB. They have a .io domain with a name that normally ends in -er but they removed the 'e'. They code in Sublime Text on their Macbooks while sitting in Starbucks. When people ask about their work they refer to themselves as a "code artisan".

Thank you!


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF CAR'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH ~~PERL~~ Python.