Table of Contents

Python Types and Objects **1**
- Shalabh Chaturvedi 1
- Before You Begin 2
- Chapter 1. Basic Concepts 2
- The Object Within 2
- A Clean Slate 4
- Relationships 5
- Chapter 2. Bring In The Objects 5
- The First Objects 5
- More Built-in Types 8
- New Objects by Subclassing 9
- New Objects by Instantiating 9
- It's All Instantiation, Really 10
- Chapter 3. Wrap Up 12
- The Python Objects Map 12
- Summary 13
- More Types to Play With 13
- What's the Point, Anyway? 14
- Classic Classes 14
- Chapter 4. Stuff You Should Have Learnt Elsewhere 15
- Object-Oriented Relationships 15
- Related Documentation 18
- Colophon 18

Python Types and Objects

Shalabh Chaturvedi

Copyright © 2005-2009 Shalabh Chaturvedi

All Rights Reserved.

About This Book

Explains Python new-style objects:

- what are `<type 'type'>` and `<type 'object'>`
- how user defined classes and instances are related to each other and to built-in types
- what are metaclasses

New-style implies Python version 2.2 and upto and including 3.x. There have been some behavioral changes during these version but all the concepts covered here are valid. The system described is sometimes called the Python *type system*, or the *object model*.

This book is part of a series:

1. Python Types and Objects [you are here]
2. Python Attributes and Methods

This revision:
Discuss | Latest version | Cover page
Author: shalabh@cafepy.com

Table of Contents

Before You Begin
1. Basic Concepts
    The Object Within
    A Clean Slate
    Relationships
2. Bring In The Objects
    The First Objects
    More Built-in Types
    New Objects by Subclassing
    New Objects by Instantiating
    It's All Instantiation, Really
3. Wrap Up
    The Python Objects Map
    Summary
    More Types to Play With
    What's the Point, Anyway?
    Classic Classes
4. Stuff You Should Have Learnt Elsewhere
    Object-Oriented Relationships
Related Documentation

List of Figures

1.1. A Clean Slate
2.1. Chicken and Egg
2.2. Some Built-in Types
2.3. User Built Objects
3.1. The Python Objects Map
4.1. Relationships
4.2. Transitivity of Relationships

List of Examples

1.1. Examining an integer object
2.1. Examining <type 'object'> and <type 'type'>
2.2. There's more to <type 'object'> and <type 'type'>
2.3. Examining some built-in types
2.4. Creating new objects by subclassing
2.5. Creating new objects by instantiating
2.6. Specifying a type object while using class statement
3.1. More built-in types
3.2. Examining classic classes

Before You Begin

Some points you should note:

- This book covers the new-style objects (introduced a long time ago in Python 2.2). Examples are valid for Python 2.5 and all the way to Python 3.x.
- This book is not for absolute beginners. It is for people who already know Python (even a little Python) and want to know more.
- This book provides a background essential for grasping *new-style* attribute access and other mechanisms (descriptors, properties and the like). If you are interested in only attribute access, you could go straight to Python Attributes and Methods, after verifying that you understand the Summary of this book.

Happy pythoneering!

Chapter 1. Basic Concepts

The Object Within

So what exactly is a Python object? An object is an axiom in our system - it is the notion of some *entity*. We still define an object by saying it has:

- Identity (i.e. given two names we can say for sure if they refer to one and the same object, or not).
- A value - which may include a bunch of attributes (i.e. we can reach other objects through objectname.attributename).
- 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.
- One or more *bases*. Not all objects have bases but some special ones do. A base is similar

to a super-class or base-class in object-oriented lingo.

If you are more of the 'I like to know how the bits are laid out' type as opposed to the 'I like the meta abstract ideas' type, it might be useful for you to know that each object also has a specific location in main memory that you can find by calling the `id()` function.

The *type* and *bases* (if they exist) are important because they define special relationships an object has with other objects. Keep in mind that the types and bases of objects just other objects. This will be re-visited soon.

You might think an object has a name but the name is not really part of the object. The name exists outside of the object in a namespace (e.g. a function local variable) or as an attribute of another object.

Even a simple object such as the number `2` has a lot more to it than meets the eye.

Example 1.1. Examining an integer object

```
>>> two = 2 ❶
>>> type(two)
<type 'int'> ❷
>>> type(type(two))
<type 'type'> ❸
>>> type(two).__bases__
(<type 'object'>,) ❹
>>> dir(two) ❺
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
 '__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
 '__floordiv__', '__format__', '__getattribute__', '__getnewargs__',
 '__hash__', '__hex__', '__index__', '__init__', '__int__', '__invert__',
 '__long__', '__lshift__', '__mod__', '__mul__', '__neg__', '__new__',
 '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
 '__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
 '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
 '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
 '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
 'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

❶ Here we give an integer the name `two` in the current namespace.

❷ The type of this object is `<type 'int'>`. This `<type 'int'>` is another object, which we will now explore. Note that this object is also called just `int` and `<type 'int'>` is the printable representation.

❸ Hmm.. the type of `<type 'int'>` is an object called `<type 'type'>`.

❹ Also, the `__bases__` attribute of `<type 'int'>` is a tuple containing an object called `<type 'object'>`. Bet you didn't think of checking the `__bases__` attribute ;).

❺ Let's list all the attributes present on this original integer object - wow that's a lot.

You might say "What does all this mean?" and I might say "Patience! First, let's go over the first rule."

Rule 1

Everything is an object

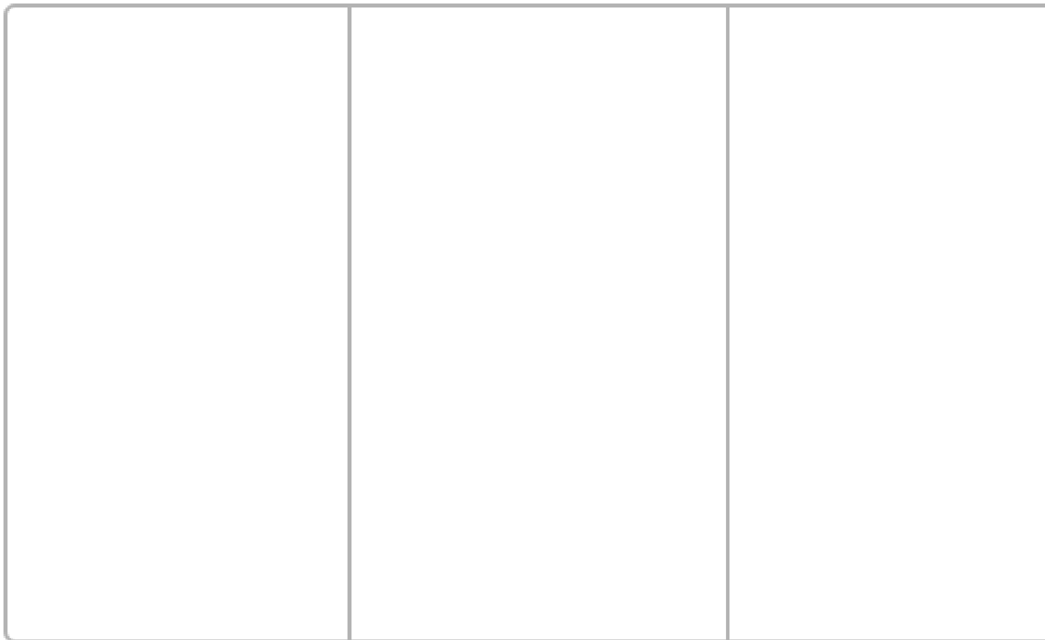
The built-in `int` is an object. This doesn't mean that just the numbers such as `2` and `77` are objects (which they are) but also that there is another object called `int` that is sitting in memory right beside the actual integers. In fact all integer objects are pointing to `int` using their `__class__` attribute saying "that guy really knows me". Calling `type()` on an object just returns the value of the `__class__` attribute.

Any classes that we define are objects, and of course, instances of those classes are objects as well. Even the functions and methods we define are objects. Yet, as we will see, all objects are not equal.

A Clean Slate

We now build the Python object system from scratch. Let us begin at the beginning - with a clean slate.

Figure 1.1. A Clean Slate



You might be wondering why a clean slate has two grey lines running vertically through it. All will be revealed when you are ready. For now this will help distinguish a slate from another figure. On this clean slate, we will gradually put different objects, and draw various relationships, till it is left looking quite full.

At this point, it helps if any preconceived object oriented notions of classes and objects are set aside, and everything is perceived in terms of objects (*our* objects) and relationships.

Relationships

As we introduce many different objects, we use two kinds of relationships to connect. These are the *subclass-superclass* relationship (a.k.a. specialization or inheritance, *"man is an animal"*, etc.) and the *type-instance* relationship (a.k.a instantiation, *"Joe is a man"*, etc.). If you are familiar with these concepts, all is well and you can proceed, otherwise you might want to take a detour through the section called “Object-Oriented Relationships”.

Chapter 2. Bring In The Objects

The First Objects

We examine two objects: `<type 'object'>` and `<type 'type'>`.

Example 2.1. Examining `<type 'object'>` and `<type 'type'>`

```
>>> object ❶
<type 'object'>
>>> type ❷
<type 'type'>
>>> type(object) ❸
<type 'type'>
>>> object.__class__ ❹
<type 'type'>
>>> object.__bases__ ❺
()
>>> type.__class__ ❻
<type 'type'>
>>> type.__bases__ ❼
(<type 'object'>,)
```

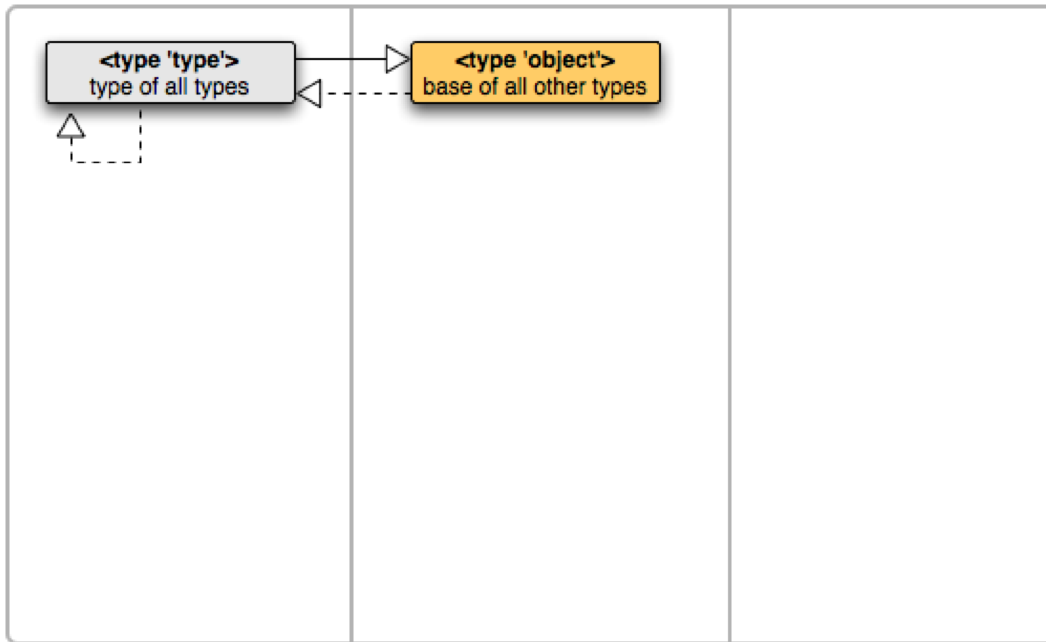
❶ ❷ The names of the two primitive objects within Python. Earlier `type()` was introduced as a way to find the type of an object (specifically, the `__class__` attribute). In reality, it is both - an object itself, and a way to get the type of another object.

❸ ❹ ❺ Exploring `<type 'object'>`: the type of `<type 'object'>` is `<type 'type'>`. We also use the `__class__` attribute and verify it is the same as calling `type()`.

❻ ❼ Exploring `<type 'type'>`: interestingly, the type of `<type 'type'>` is itself! The `__bases__` attribute points to `<type 'object'>`.

Let's make use of our slate and draw what we've seen.

Figure 2.1. Chicken and Egg



These two objects are primitive objects in Python. We might as well have introduced them one at a time but that would lead to the chicken and egg problem - which to introduce first? These two objects are interdependent - they cannot stand on their own since they are defined in terms of each other.

Continuing our Python experimentation:

Example 2.2. There's more to `<type 'object'>` and `<type 'type'>`

```
>>> isinstance(object, object) ❶
True
>>> isinstance(type, object) ❷
True
```

❶ Whoa! What happened here? This is just Dashed Arrow Up Rule in action. Since `<type 'type'>` is a subclass of `<type 'object'>`, instances of `<type 'type'>` are instances of `<type 'object'>` as well.

❷ Applying both Dashed Arrow Up Rule and Dashed Arrow Down Rule, we can effectively reverse the direction of the dashed arrow. Yes, it is still consistent.

If the above example proves too confusing, ignore it - it is not much use anyway.

Now for a new concept - *type objects*. Both the objects we introduced are type objects. So what do we mean by type objects? Type objects share the following traits:

- They are used to represent abstract data types in programs. For instance, one (user defined) object called `User` might represent all users in a system, another once called `int` might represent all integers.

- They can be *subclassed*. This means you can create a new object that is somewhat similar to exsiting type objects. The existing type objects become bases for the new one.
- They can be *instantiated*. This means you can create a new object that is an instance of the existing type object. The existing type object becomes the `__class__` for the new object.
- The type of any type object is `<type 'type'>`.
- They are lovingly called *types* by some and *classes* by others.

Yes you read that right. Types and classes are really the same in Python (disclaimer: this doesn't apply to old-style classes or pre-2.2 versions of Python. Back then types and classes had their differences but that was a long time ago and they have since reconciled their differences so let bygones be bygones, shall we?). No wonder the `type()` function and the `__class__` attribute get you the same thing.

The term *class* was traditionally used to refer to a class created by the `class` statement. Built-in types (such as `int` and `string`) are not usually referred to as classes, but that's more of a convention thing and in reality types and classes are exactly the same thing. In fact, I think this is important enough to put in a rule:

Class is Type is Class

The term *type* is equivalent to the term *class* in all version of Python >= 2.3.

Types and (er.. for lack of a better word) non-types (ugh!) are both objects but only types can have subcasses. Non-types are concrete values so it does not make sense for another object be a subclass. Two good examples of objects that are not types are the integer `2` and the string `"hello"`. Hmm.. what does it mean to be a subclass of `2`?

Still confused about what is a type and what is not? Here's a handy rule for you:

Type Or Non-type Test Rule

If an object is an instance of `<type 'type'>`, then it is a type. Otherwise, it is not a type.

Looking back, you can verify that this is true for all objects we have come across, including `<type 'type'>` which is an instance of itself.

To summarize:

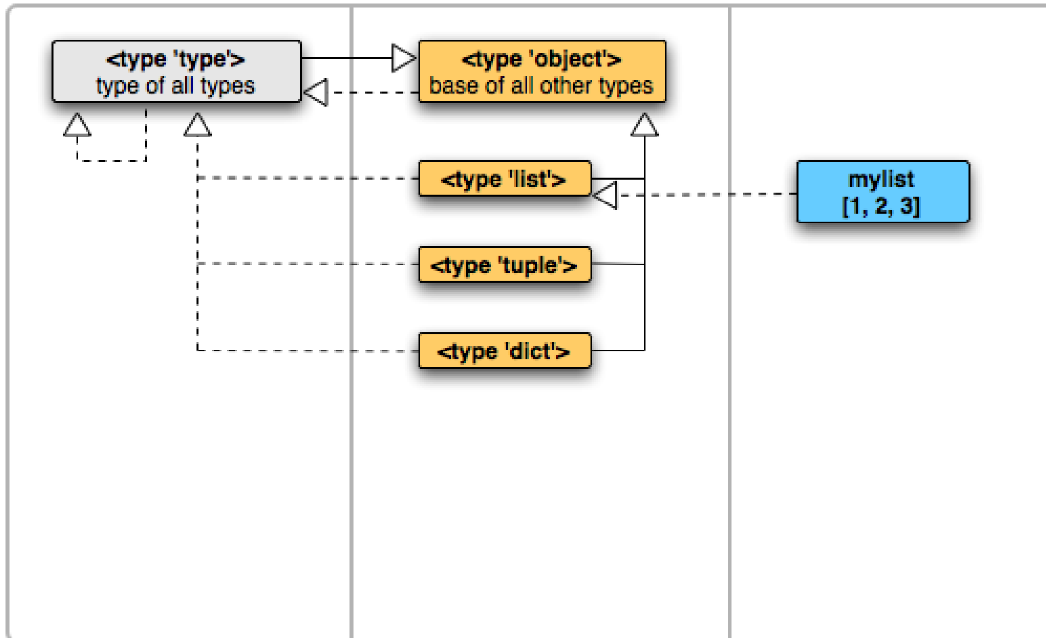
1. `<type 'object'>` is an instance of `<type 'type'>`.
2. `<type 'object'>` is a subclass of no object.
3. `<type 'type'>` is an instance of itself.
4. `<type 'type'>` is a subclass of `<type 'object'>`.
5. There are only two kinds of objects in Python: to be unambiguous let's call these *types* and *non-types*. Non-types could be called instances, but that term could also refer to a type, since a type is always an instance of another type. Types could also be called classes, and I do call them classes from time to time.

Note that we are drawing arrows on our slate for only the *direct* relationships, not the implied ones (i.e. only if one object is another's `__class__`, or in the other's `__bases__`). This make economic use of the slate and our mental capacity.

More Built-in Types

Python does not ship with only two objects. Oh no, the two primitives come with a whole gang of buddies.

Figure 2.2. Some Built-in Types



A few built-in types are shown above, and examined below.

Example 2.3. Examining some built-in types

```
>>> list ❶
<type 'list'>
>>> list.__class__  ❷
<type 'type'>
>>> list.__bases__  ❸
(<type 'object'>,)
>>> tuple.__class__, tuple.__bases__ ❹
(<type 'type'>, (<type 'object'>,))
>>> dict.__class__, dict.__bases__ ❺
(<type 'type'>, (<type 'object'>,))
>>>
>>> mylist = [1,2,3] ❻
>>> mylist.__class__ ❼
<type 'list'>
```

❶ The built-in `<type 'list'>` object.

❷ Its type is `<type 'type'>`.

❸ It has one base (a.k.a. superclass), `<type 'object'>`.

❹ ❺ Ditto for `<type 'tuple'>` and `<type 'dict'>`.

❻ This is how you create an instance of `<type 'list'>`.

❼ The type of a list is `<type 'list>`. No surprises here.

When we create a tuple or a dictionary, they are instances of the respective types.

So how can we create an *instance* of `mylist`? We cannot. This is because `mylist` is a not a type.

New Objects by Subclassing

The built-in objects are, well, *built into* Python. They're there when we start Python, usually there when we finish. So how can we create new objects?

New objects cannot pop out of thin air. They have to be built using existing objects.

Example 2.4. Creating new objects by subclassing

```
# In Python 2.x:
class C(object): ❶
    pass

# In Python 3.x, the explicit base class is not required, classes are
# automatically subclasses of object:
class C: ❷
    pass

class D(object):
    pass

class E(C, D): ❸
    pass

class MyList(list): ❹
    pass
```

❶ The `class` statement tells Python to create a new type by subclassing an existing type.

❷ Don't do this in Python 2.x or you will end up with an object that is an old-style class, everything you read here will be useless and all will be lost.

❸ Multiple bases are fine too.

❹ Most built-in types can be subclassed (but not all).

After the above example, `C.__bases__` contains `<type 'object'>`, and `MyList.__bases__` contains `<type 'list'>`.

New Objects by Instantiating

Subclassing is only half the story.

Example 2.5. Creating new objects by instantiating

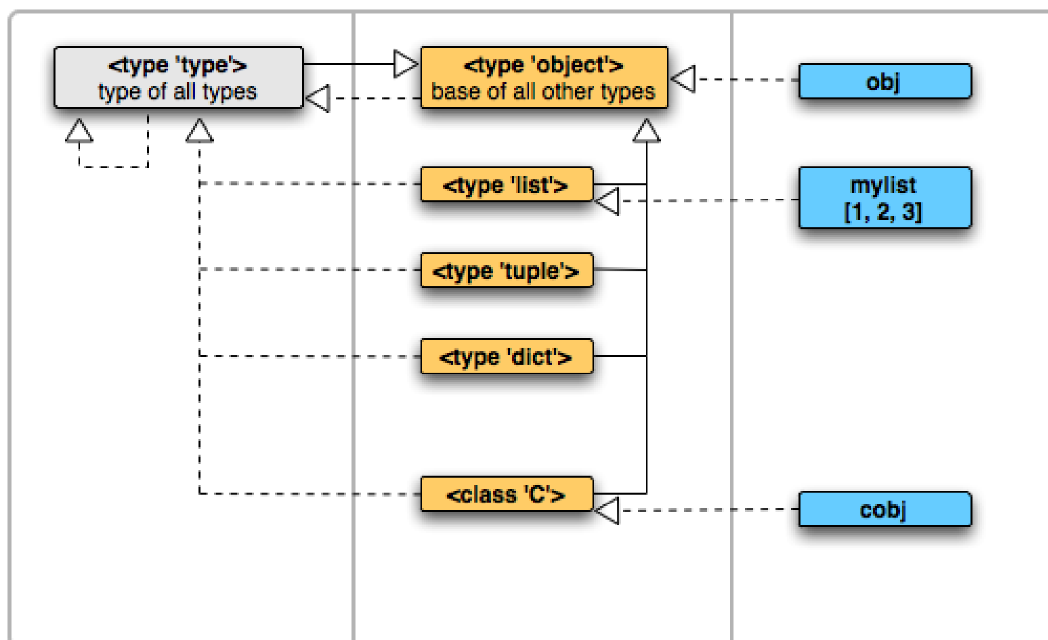
```
obj = object() ❶

cobj = C() ❷

mylist = [1,2,3] ❸
```

❶ ❷ The call operator (`()`) creates a new object by instantiating an existing object. The existing object *must* be a type. Depending on the type, the call operator might accept arguments.

❷ Python syntax creates new objects for some built-in types. The square brackets create an instance of `<type 'list'>`; a numeric literal creates an instance of `<type 'int'>`.

After the above exercise, our slate looks quite full.

Figure 2.3. User Built Objects



Note that by just subclassing `<type 'object'>`, the type `C` automatically is an instance of `<type 'type'>`. This can be verified by checking `C.__class__`. Why this happens is explained in the next section.

It's All Instantiation, Really

Some questions are probably popping up in your head at this point. Or maybe they aren't, but I'll answer them anyway:

Q: How does Python *really* create a new object?

A:

Internally, when Python creates a new object, it always uses a type and creates an instance of that object. Specifically it uses the `__new__()` and `__init__()` methods of the type (discussion of those is outside the scope of this book). In a sense, the type serves as a factory that can churn out new objects. The type of these manufactured objects will be the type object used to create them. This is why every object has a type.

Q: When using instantiation, I specify the type, but how does Python know which type to use when I use subclassing?

A: It looks at the base class that you specified, and uses its type as the type for the new object. In the example Example 2.4, “Creating new objects by subclassing” , `<type 'type'>` (the type of `<type 'object'>`, the specified base) is used as the type object for creating `C`.

A little thought reveals that under most circumstances, any subclasses of `<type 'object'>` (and their subclasses, and so on) will have `<type 'type'>` as their type.

Advanced Material Ahead

Advanced discussion ahead, tread with caution, or jump straight to the next section.

Q: Can I instead specify a type object to use?

A: Yes. One option is by using the `__metaclass__` class attribute as in the following example:

Example 2.6. Specifying a type object while using `class` statement

```
class MyCWithSpecialType(object):
    __metaclass__ = SpecialType
```

Now Python will create `MyCWithSpecialType` by instantiating `SpecialType`, and not `<type 'type'>`.

Q: Wow! Can I use any type object as the __metaclass__?

A: No. It must be a subclass of the type of the base object. In the above example:

- Base of `MyCWithSpecialType` is `<type 'object'>`.
- Type of `<type 'object'>` is `<type 'type'>`.
- Therefore `SpecialType` must be a subclass of `<type 'type'>`.

Implementation of something like `SpecialType` requires special care and is out of scope for this book.

Q: What if I have multiple bases, and don't specify a `__metaclass__` - which type object will be used?

A: Good Question. Depends if Python can figure out which one to use. If all the bases have the same type, for example, then that will be used. If they have different types that are not related, then Python cannot figure out which type object to use. In this case specifying a `__metaclass__` is required, and this `__metaclass__` must be a subclass of the type of each base.

When should I use a `__metaclass__`?

Q:

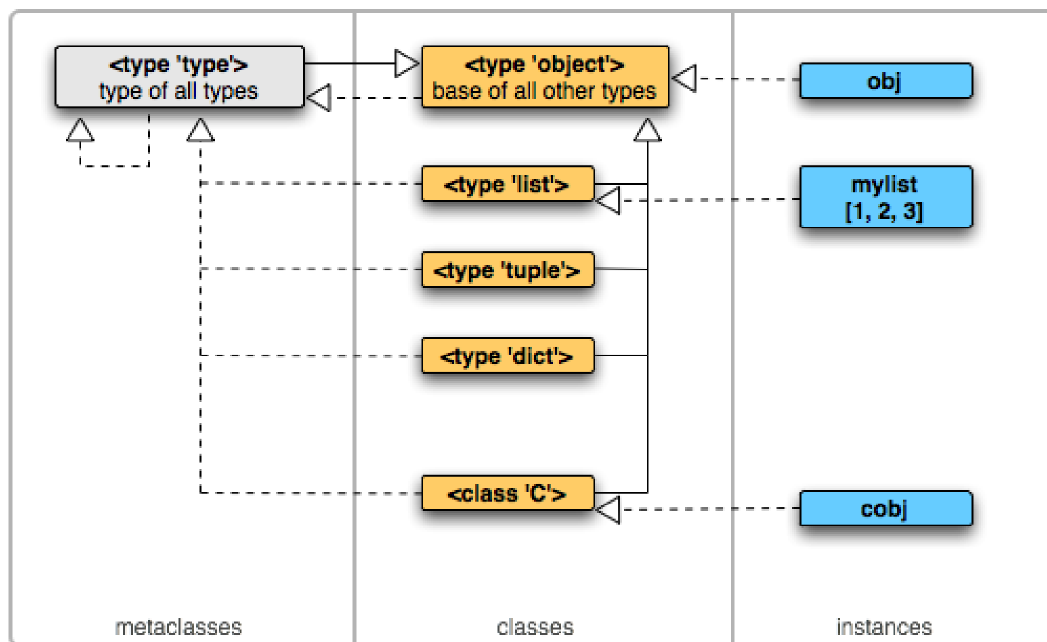
A: Never (as long as you're asking this question anyway :)

Chapter 3. Wrap Up

The Python Objects Map

We really ended up with a map of different kinds of Python objects in the last chapter.

Figure 3.1. The Python Objects Map



Here we also unravel the mystery of the vertical grey lines. They just segregate objects into three spaces based on what the common man calls them - *metaclasses*, *classes*, or *instances*.

Various pedantic observations of the diagram above:

1. Dashed lines cross spacial boundaries (i.e. go from object to *meta*-object). Only exception is `<type 'type'>` (which is good, otherwise we would need another space to the left of it, and another, and another...).
2. Solid lines do not cross space boundaries. Again, `<type 'type'>` -> `<type 'object'>` is an exception.
3. Solid lines are not allowed in the rightmost space. These objects are too concrete to be subclassed.
4. Dashed line arrow heads are not allowed rightmost space. These objects are too concrete to be instantiated.
5. Left two spaces contain types. Rightmost space contains non-types.
6. If we created a new object by subclassing `<type 'type'>` it would be in the leftmost space, and would also be both a subclass and instance of `<type 'type'>`.

Also note that <type 'type'> is indeed a type of all types, and <type 'object'> a superclass of all types (except itself).

Summary

To summarize all that has been said:

- There are two kinds of objects in Python:
    1. *Type objects* - can create instances, can be subclassed.
    2. *Non-type objects* - cannot create instances, cannot be subclassed.
- <type 'type'> and <type '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 <type 'object'>.
- To create a new object using subclassing, we use the class statement and specify the bases (and, optionally, the type) of the new object. This always creates a type object.
- To create a new object using instantiation, we use the call operator (()) on the type object we want to use. This may create a type or a non-type object, depending on which type object was used.
- Some non-type objects can be created using special Python syntax. For example, [1, 2, 3] creates an instance of <type 'list'>.
- Internally, Python *always* uses a type object to create a new object. The new object created is an instance of the type object used. Python determines the type object from a class statement by looking at the bases specified, and finding their types.
- issubclass(A,B) (testing for superclass-subclass relationship) returns True iff:
    1. B is in A.__bases__, or
    2. issubclass(Z,B) is true for any Z in A.__bases__.
- isinstance(A,B) (testing for type-instance relationship) returns True iff:
    1. B is A.__class__, or
    2. issubclass(A.__class__,B) is true.
- Squasher is really a python. (Okay, that wasn't mentioned before, but now you know.)

More Types to Play With

The following example shows how to discover and experiment with built-in types.

Example 3.1. More built-in types

```
>>> import types ❶
>>> types.ListType is list ❷
True
>>> def f(): ❸
...     pass
...
>>> f.__class__ is types.FunctionType ❹
True
>>>
>>> class MyList(list): ❺
...     pass
...
```

```
>>> class MyFunction(types.FunctionType): ❻
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: type 'function' is not an acceptable base type
>>> dir(types) ❼
['BooleanType', 'DictProxyType', 'DictType', ..]
```

❶ The `types` module contains many built-in types.
❷ Some well known types have another name as well.
❸ `def` creates a function object.
❹ The type of a function object is `types.FunctionType`
❺ Some built-in types can be subclassed.
❻ Some cannot.
❼ More types than you can shake a stick at.

What's the Point, Anyway?

So we can create new objects with any relationship we choose, but what does it buy us?

The relationships between objects determine how attribute access on the object works. For example, when we say `objectname.attributename`, which object do we end up with? It all depends on `objectname`, its type, and its bases (if they exist).

Attribute access mechanisms in Python are explained in the second book of this series: Python Attributes and Methods.

Classic Classes

This is a note about *classic* classes in Python. We can create classes of the old (pre 2.2) kind by using a plain class statement.

Example 3.2. Examining classic classes

```
>>> class ClassicClass: ❶
...     pass
...
>>> type(ClassicClass) ❷
<type 'classobj'>
>>> import types
>>> types.ClassType is type(ClassicClass) ❸
True
>>> types.ClassType.__class__ ❹
<type 'type'>
>>> types.ClassType.__bases__ ❺
(<type 'object'>,)
```

❶ A class statement specifying *no* bases creates a classic class Remember that to create a new-style class you must specify `object` as the base (although this is not required in Python 3.0 since new-style classes are the default). Specifying only classic classes as bases also creates a classic class. Specifying both classic and new-style classes as bases create a new-style class.

❷ Its type is an object we haven't seen before (in this book).

❸ The type of classic classes is an object called `types.ClassType`.

❹ ❺ It looks and smells like just another type object.

The `types.ClassType` object is in some ways an alternative `<type 'type'>`. Instances of this object (classic classes) are types themselves. The rules of attribute access are different for classic classes and new-style classes. The `types.ClassType` object exists for backward compatibility and may not exist in future versions of Python. Other sections of this book should not be applied to classic classes.

Comment on this book here: discussion page. I appreciate feedback!

That's all, folks!

Chapter 4. Stuff You Should Have Learnt Elsewhere

Object-Oriented Relationships

Can Skim Section

This oddly placed section explains the *type-instance* and *supertype-subtype* relationships, and can be safely skipped if the reader is already familiar with these OO concepts. Skimming over the rules below might be useful though.

While we introduce many different objects, we only use two kinds of relationships (Figure 4.1, “Relationships”):

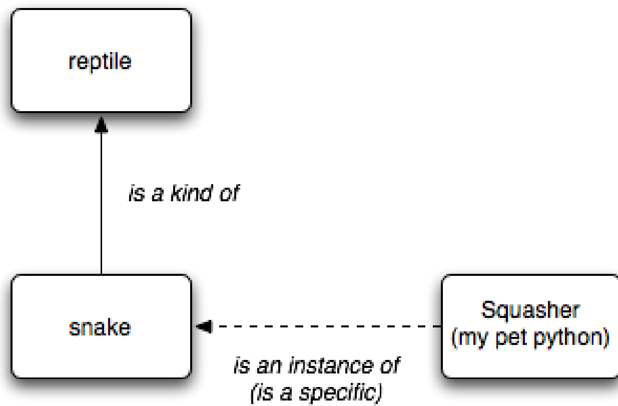
- *is a kind of* (solid line): Known to the OO folks as specialization, this relationship exists between two objects when one (the *subclass*) is a specialized version of the other (the *superclass*). A snake *is a kind of* reptile. It has all the traits of a reptile and some specific traits which identify a snake.

  Terms used: *subclass of*, *superclass of* and *superclass-subclass*.
- *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*, *type-instance* and *class-instance*.

Note that in plain English, the term *'is a'* is used for both of the above relationships. *Squasher is a snake* and *snake is a reptile* are both correct. We, however, use specific terms from above to avoid any confusion.

Figure 4.1. Relationships



We use the solid line for the first relationship because these objects are *closer* to each other than ones related by the second. To illustrate - if one is asked to list words similar to 'snake', one is likely to come up with 'reptile'. However, when asked to list words similar to 'Squasher', one is unlikely to say 'snake'.

It is useful at this point to note the following (independent) properties of relationships:

Dashed Arrow Up Rule

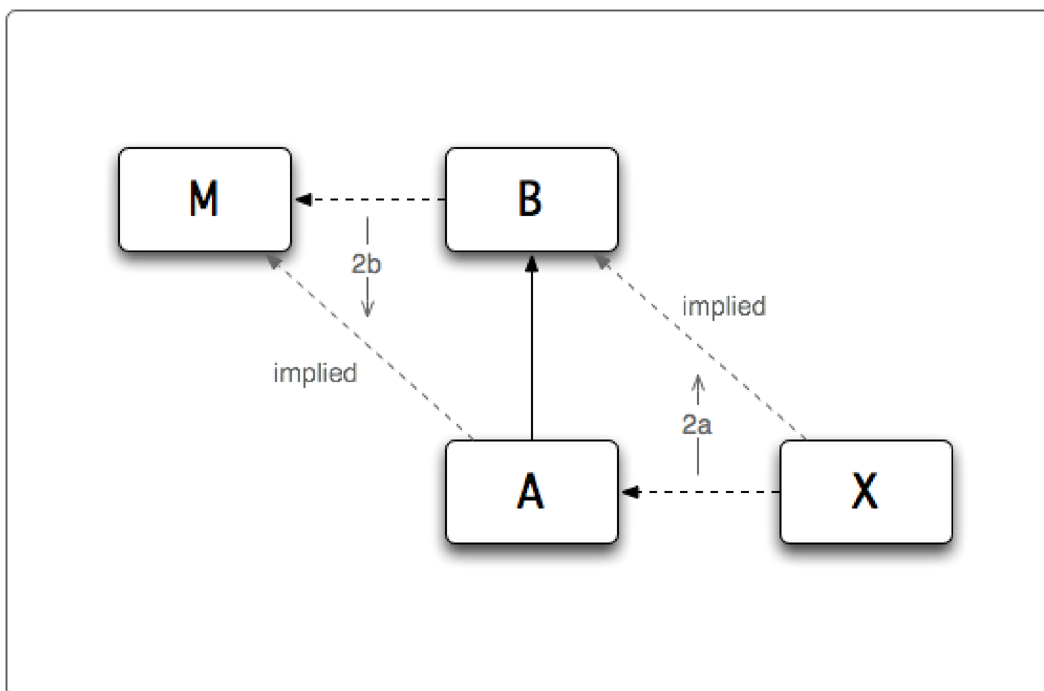
If X is an instance of A, and A is a subclass of B, then X is an instance of B as well.

Dashed Arrow Down Rule

If B is an instance of M, and A is a subclass of B, then A is an instance of M as well.

In other words, the head end of a dashed arrow can move up a solid arrow, and the tail end can move down (shown as 2a and 2b in Figure 4.2, “Transitivity of Relationships” respectively). These properties can be directly derived from the definition of the superclass-subclass relationship.

Figure 4.2. Transitivity of Relationships



Applying Dashed Arrow Up Rule, we can derive the second statement from the first:

1. Squasher is an instance of snake (or, the type of Squasher is snake).
2. Squasher is an instance of reptile (or, the type of Squasher is reptile).

Earlier we said that an object has exactly one type. So how does Squasher have two? Note that although both statements are correct, one is more correct (and in fact subsumes the other). In other words:

- `Squasher.__class__` is `snake`. (In Python, the `__class__` attribute points to the type of an object).
- Both `isinstance(Squasher, snake)` and `isinstance(Squasher, reptile)` are true.

A similar rules exists for the superclass-subclass relationship.

Combine Solid Arrows Rule

If A is a subclass of B, and B is a subclass of C, then A is a subclass of C as well.

A snake is a kind of reptile, and a reptile is a kind of animal. Therefore a snake is a kind of animal. Or, in Pythonese:

- `snake.__bases__` is `(reptile,)`. (The `__bases__` attribute points to a tuple containing superclasses of an object).
- Both `issubclass(snake, reptile)` and `issubclass(snake, animal)` are true.

Note that it is possible for an object to have more than one base.

Related Documentation

[descrintro] Unifying types and classes in Python 2.2. Guido van Rossum.

[pep-253] Subclassing Built-in Types. Guido van Rossum.

Colophon

This book was written in DocBook XML. The HTML version was produced using DocBook XSL stylesheets and `xsltproc`. The PDF version was produced using `htmldoc`. The diagrams were drawn using OmniGraffe [1]. The process was automated using Paver [2].

---

[1] http://www.omnigroup.com/
[2] http://www.blueskyonmars.com/projects/paver/

Feedback

Comment on this book here: discussion page. I appreciate feedback!