# Table of Contents

# Python Types and Objects

**Shalabh Chaturvedi**


## About This Book

Explains different Python *new−style* objects, starting with `<type 'type'>` and `<type 'object'>`, and going all the way to user defined classes and instances. New−style implies Python version 2.2 and up. 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: 1.12
Links: Latest version | Other formats | Discuss
Author: shalabh@cafepy.com

## Before You Begin

Some points you should note:

- This book covers *only* the new−style objects (introduced with Python 2.2). Examples are valid for Python 2.3.3.
- This book is not for absolute beginners. It is for people who already know Python (some Python at least), 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

## 1.1. The Object Within

In the pre 2.2 world, we essentially had three kinds of *objects* to play with − we had the:

- types (list, tuple, string, etc.)
- classes (any classes that *we* define)
- instances (the third kind of object)

Types and classes were known not to socialize, and you couldn't create new types. The new (2.2 and later) system moves around many of the old rules, and it is best to start afresh to understand it better. From here on we only discuss the new system.

So what exactly is an object? An object is an axiom in our system − it is the notion of some *thing*. However, we still define an object by saying it has:

- identity (i.e. given two names we can say for sure if they are one and the same object, or not)
- attributes (i.e. we can reach other objects through `objectname.attributename`)
- relationships (these are just attributes, but special because Python knows about them). There are basically two of these (elaborated later):
    - *type* − every object has exactly one *type*.
    - *bases* − some objects may have more or one *bases*.
- name (the name of the object, some may have more than one, some may have none)

This brings us to our first rule.

### Rule 1

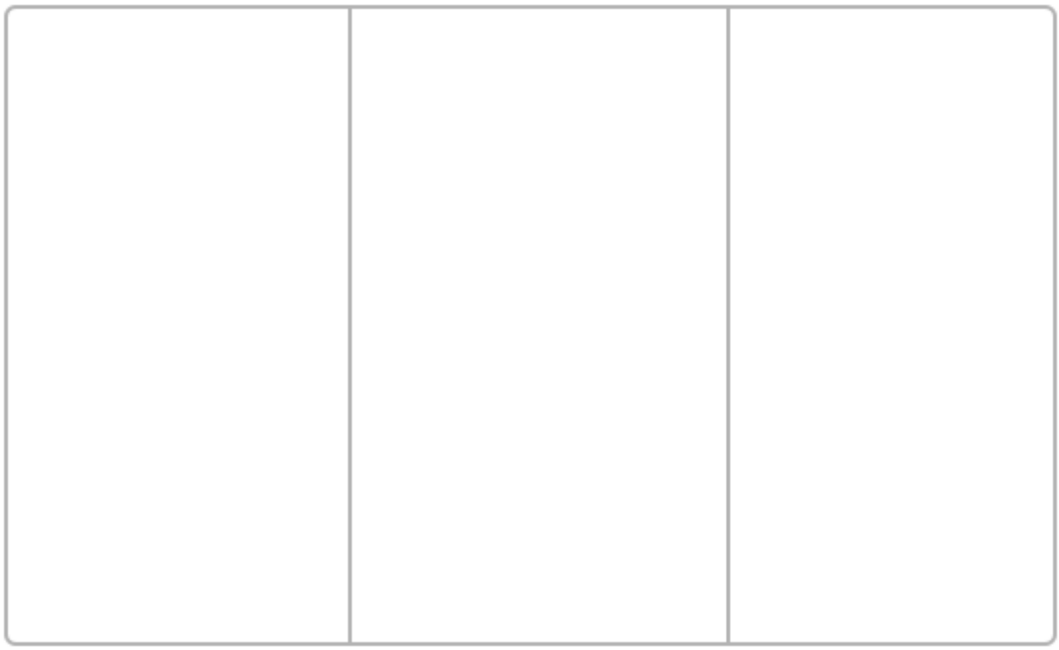
Everything is an object

Not to be taken too seriously, but this rule does make an important point. The `list`, `tuple` and `string` built−ins are objects. Any classes that we define are objects, and of course, instances of those classes are objects as well. Yet, as we will see, all objects are not equal. We keep the notion of objects and relationships at this point and move on.

## 1.2. 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 the time is right. 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.

# 1.3. Relationships

## Can Skim Section

This 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.

While we introduce many different objects, we only use two kinds of relationships (Figure 1.2, Relationships ):

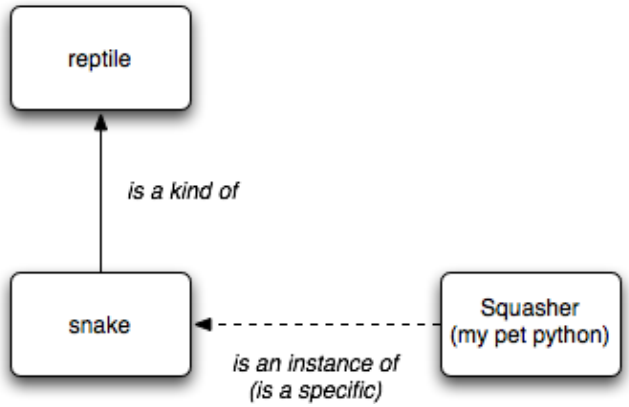
- *is a kind of* (solid line): Known to the OO folks as specialization, this relationship exists between two objects when one (the *subtype*) is a specialized version of the other (the *supertype*). A snake *is a kind of* reptile. It has all the traits of a reptile and some specific traits which identify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

  Terms used: *subtype of*, *supertype of* and *supertype−subtype*.
- *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* and *type−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 1.2. 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'. One might (if one knew the names of the author's other pets) instead come up with 'Squisher', 'Stinger' and 'Hisser'.

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

## Dashed Arrow Up Rule

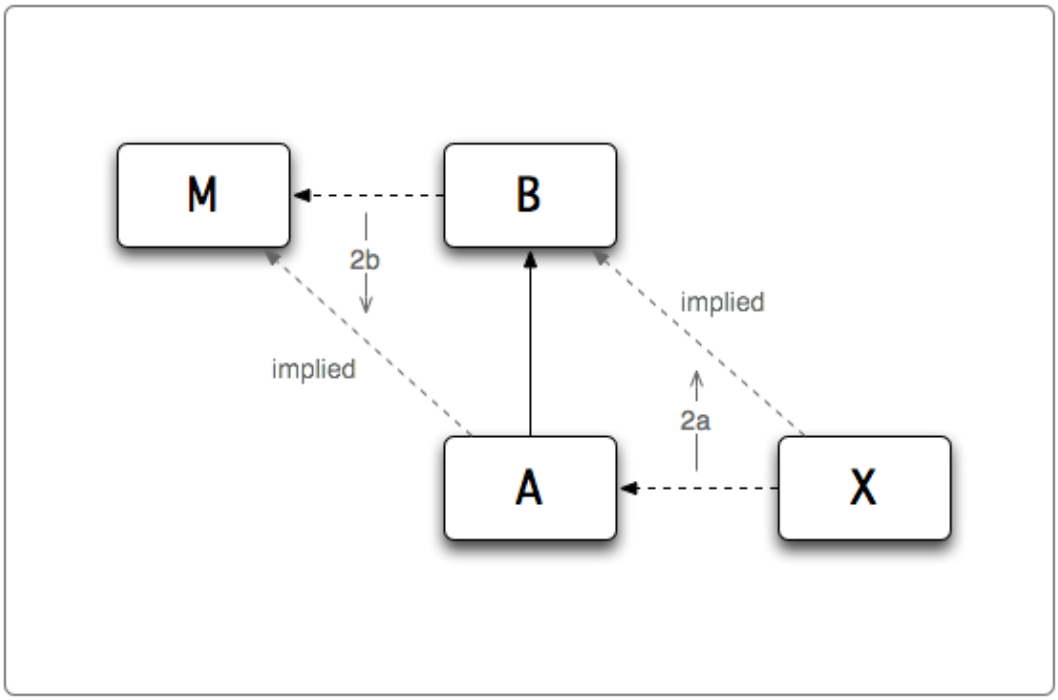
If X is an instance of A, and A is a subtype of B, then X is an instance of B as well.

## Dashed Arrow Down Rule

If B is an instance of M, and A is a subtype 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 1.3,  Transitivity of Relationships  respectively). These properties can be directly derived from the definition of the supertype−subtype relationship.

**Figure 1.3. 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 supertype−subtype relationship.

## Combine Solid Arrows Rule

If A is a subtype of B, and B is a subtype of C, then A is a subtype 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 supertypes 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.

# Chapter 2. Bring In The Objects

## 2.1. The First Objects

Now that we have our objects and relationships, let's get started. We first try to examine two as of yet mysterious objects `<type 'object'>` and `<type 'type'>`.

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

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

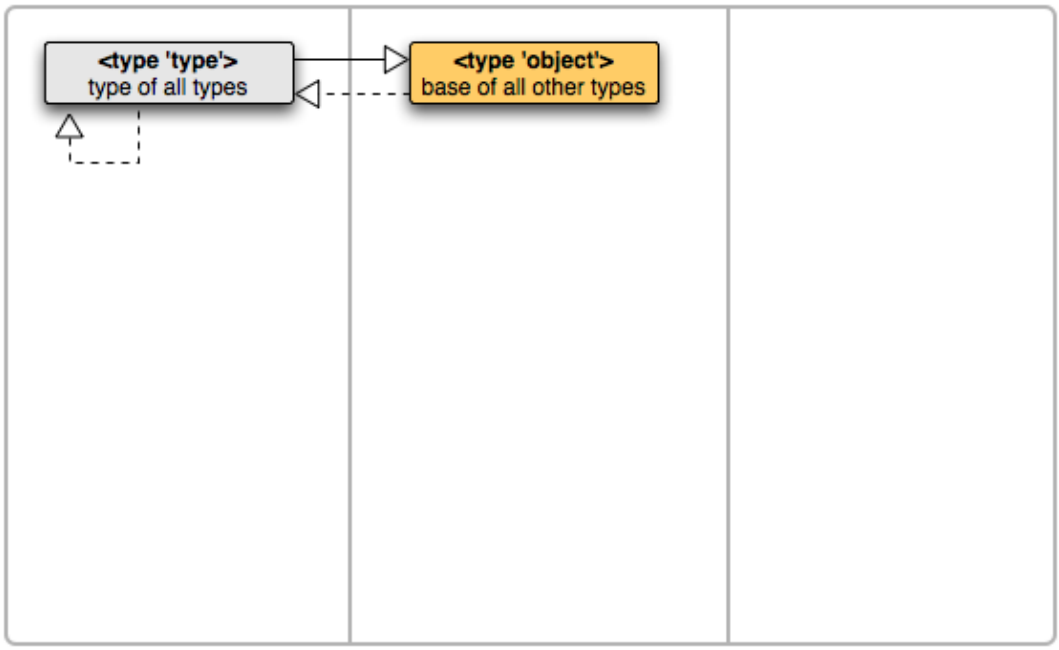
❶❷ The names of the objects within Python.

❸❹ Discovering relationships.

❺

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 subtype 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.

What do we mean by type objects? As we already know, not all objects are equal. In fact, an object can be either a *type object* or a *non−type object*. It takes a certain personality to be the 'type of' another object. Objects with this personality are called *type objects*. Such objects can participate in a type−instance relationship on the type side (as well as instance side). *Non−type objects* are doomed to always participate only on the instance side.

Also, only type objects can be the 'supertype of' another object. Apparently, you need a strong

personality for this as well. In reality, non−type objects are so concrete that it does not make sense for something else to be a subtype. If you still disagree, try to fill in the blank: _____ is a kind of Squasher.

As you guessed, Squasher is a non−type object, and 'snake' is a type object.

*Type objects* are also lovingly called *types*. The strong personality of types gets passed down to subtypes, as a result all subtypes of a type object are types themselves.

To summarize:

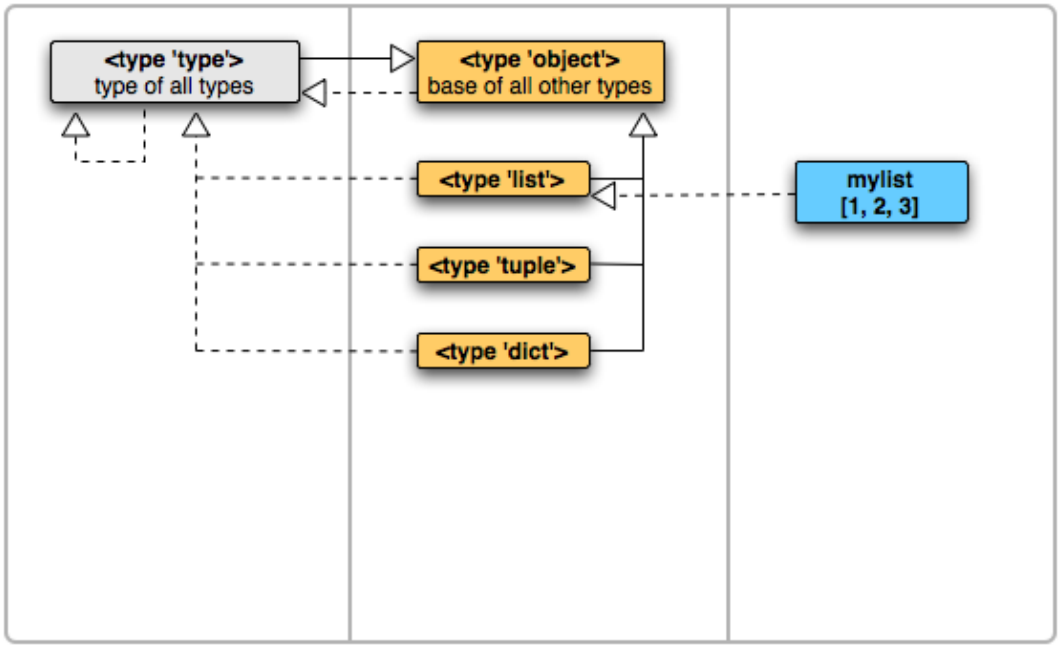
1. `<type 'object'>` is an instance of `<type 'type'>`.
2. `<type 'object'>` is a subtype of no object.
3. `<type 'type'>` is an instance of itself.
4. `<type 'type'>` is a subtype of `<type 'object'>`.
5. There are only two kinds of objects in Python: type objects and non−type objects.

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.

## 2.2. 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 supertype, `<type 'object'>`.

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

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

❼ How refreshing! For the first time an object that has an object other than `<type 'type'>` as its type.

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

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

We are now in a position to better define *type objects*.

### Type Or Non−type Test Rule

If an object is an instance of `<type 'type'>`, then it is a type object. Otherwise, it is a non−type object.

Looking back, we can verify that this is true for all objects we have come across.

## 2.3. New Objects by Subtyping

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. Specifically, we can only create objects that are at the *tail−end* of a relationship (i.e. tail−end of an arrow).

**Example 2.4. Creating new objects by subtyping**

```
class C(object): ❶
    pass ❷

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 object by subtyping an existing type object.

❷ The body of a class definition would usually provide useful methods and more.

❸ Multiple bases are fine too.

❹ Most built−in types can be subtyped (but not all).

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

The term *class* is traditionally used to imply an object created by the `class` statement. However, *classes* are now synonymous with types. Built−in types are usually not referred to as classes. This book prefers using the term *type* for both built−in and user created types.

## 2.4. New Objects by Instantiating

Subtyping is only half the story.

**Example 2.5. Creating new objects by instantiating**

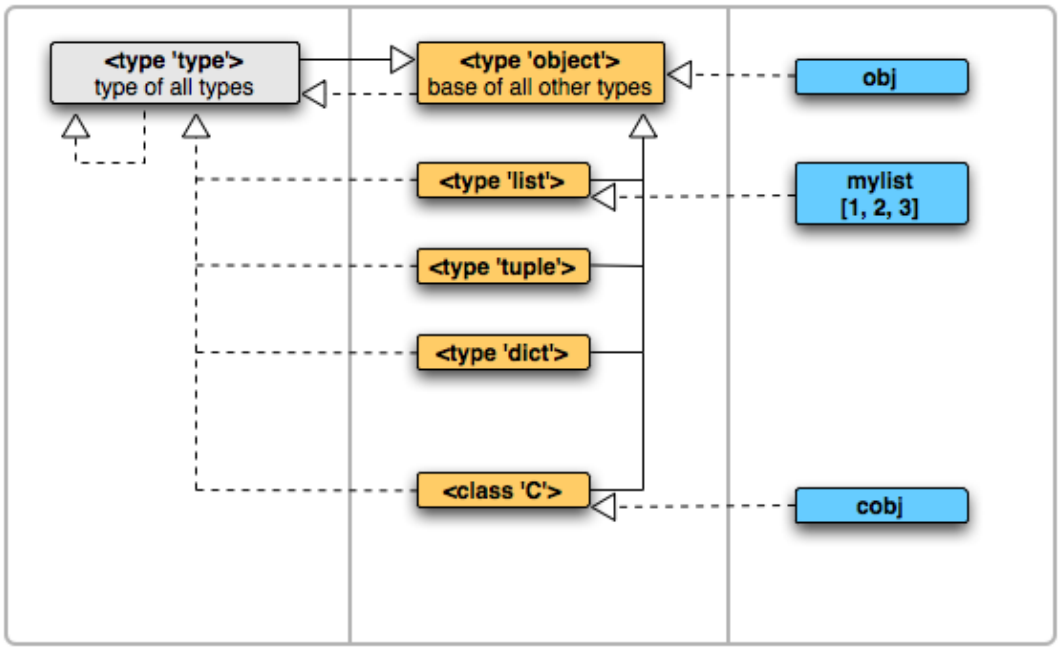
```
obj = object() ❶

cobj = C() ❷

mylist = [1,2,3] ❸
```

❶❷ The call operator (`( )`) creates a new object by instantiating an existing object. The existing object *must* be a type object. Depending on the type object, 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'>`, for example).

After the above exercise, our slate looks quite full.

**Figure 2.3. User Built Objects**




Note that by just subtyping `<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.

## 2.5. 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 object and creates an instance of that object. Specifically it uses the `__new__()` and `__init__()` methods of the type object (discussion of those is outside the scope of this book). In a sense, the type object serves as a factory that can churn out new objects (to each of which it is related as type−instance). This is why every object has a type.

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

**A:** It looks at the base object that you specified, and uses its type as the type for the new object. In the example Example 2.4, Creating new objects by subtyping , `<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 subtypes of `<type 'object'>` (and their subtypes, 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 subtype 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 subtype 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:** 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 subtype of the type of each base.

**Q:** When do I need to specify the `__metaclass__` to use?

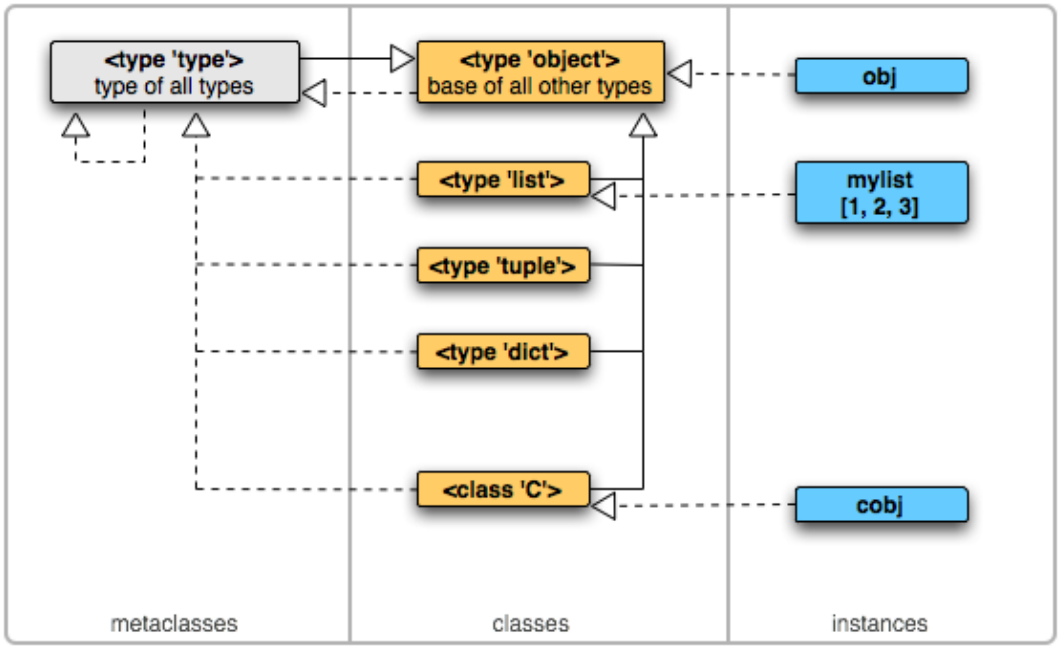
**A:** Never (as long as you're asking this question anyway :)

# Chapter 3. Wrap Up

## 3.1. 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*.

Some interesting points to note:

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 subtyped.
4. Dashed line arrow heads are not allowed rightmost space. These objects are too concrete to be instantiated.
5. Left two spaces contain type objects. Rightmost space contains non−type objects.
6. If we created a new object by subtyping `<type 'type'>` it would be in the leftmost space, and would also be both a subtype and instance of `<type 'type'>`.

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

# 3.2. Summary

To summarize all that has been said:

- There are two kinds of objects in Python:
    1. *Type objects* − can create instances, can be subtyped.
    2. *Non−type objects* − cannot create instances, cannot be subtyped.
- `<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 supertypes of the object. It is empty only for `<type 'object'>`.
- To create a new object using subtyping, 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.
- `issubtype(A,B)` (testing for supertype−subtype relationship) returns `True` iff:
    1. `B` is in `A.__bases__`, or
    2. `issubtype(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. `issubtype(A.__class__,B)` is true.
- Squasher is really a python. (Okay, that wasn't mentioned before, but now you know.)

# 3.3. 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 subtyped.
❻ Some cannot.
❼ More types than you can shake a stick at.

# 3.4. 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.

# 3.5. 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. Specifying only classic classes as bases also creates a classic class. Specifying both classic and new−style classes as bases create a new−style type.
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`.
❹❺ 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.

That's all, folks!

# End Matter

## Related Documentation

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

[pep−253] Subtyping 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 scons [2].

---

[1] http://www.omnigroup.com/
[2] http://www.scons.org/