Pythonic Theory

Version 1.3

By Kirby Urner
Thursday, July 15, 2004

Python is an "easy" programming language to start with, when learning to program. I put the word "easy" in quotes, because programming may be fairly characterized as a difficult activity. However, in relative terms, Python is fairly easy, compared to many other languages, and studying Python will get you up to speed on many of the concepts that drive contemporary languages.

Many computer languages have the concept of types. Types would include integers, floating point numbers, character strings and booleans.

Clearly integers, rational numbers, and complex numbers, are different types of *number*. In other words, integers comprise a different set than the rationals, even though all integers are also rational (but not all rationals are integers).

However, "number" is too restrictive a term for everything we might want to typify in a computer language. For example, character strings aren't numbers. So we need a more general word, which *extends* this idea of "different types of number" but to a much broader range of types.

Let's agree to speak of types *of objects*. Numbers are objects, and so are character strings, and so are all manner of objects, already defined or waiting to be defined by the programmer.

Now that we're thinking about types of object, it makes sense to explore how objects may convert to objects of another type. Additionally, objects may combine or interact according to various recipes, to yield results of the same or different type.

For example, when we add, subtract, or multiply two integers, the result is always an integer (this property is known as *closure*, and obtains only with respect to some operations). When we divide two integers, however, as in the expression 3/2, the result may be expressed as 1.5, which is not an integer, but a floating point number. In other words, an integer divided by an integer may result in a number of a different type (not an integer).

When Python was first defined, Guido van Rossum, its inventor, decided to make integers closed under division. This is called integer division, and the result is always another integer. For example, 3/2 would yield 1, because 2 goes into 3 once and no more. Fractional parts are not considered. 2/3 would yield 0, because 3 does not go into

2 at all. To get a floating point answer, at least one of the two arguments had to be a floating point.

Later, Guido decided this was a mistake and changed how the division operator worked. 2/3 would now return a floating point number. The change was made because it was too difficult to know whether x/y would return an integer or floating point, if you didn't already know what x and y were – and finding this out could mean wading through a lot of code (and even then, you might not know – maybe the user passes in x and y from outside the program).

In the emerging Python, x/y always returns a floating point answer, even if the arguments x and y are both integers. 4/2 returns 2.0, not the integer 2. And 2/4 return 0.5, not 0. However, integer division is still available using the // operator (a double slash). 2//4 evaluates to 0, and 4//2 evaluates to 2, not 2.0 (the former is an integer, the latter a floating point).

Now consider character strings. They're able to make use of the + operator (the plus symbol) in a meaningful way: 'abc' + 'def ' evaluates to 'abcdef '. We call this "concatenation" and most computer languages support it (but not always by means of the plus symbol). On the other hand, there's no obvious interpretation for 'abc' / 'def ' or 'abc' – 'def ', and Python doesn't define these.

But certainly we may think of useful operations associated with character strings other than concatenation. For example, there's finding a substring within a larger string, making a string upper case, or lower case. Python, and many other computer languages, support these operations and more.

An important distinction between object oriented semantics, and traditional mathematics, is that we think of objects as "containing" the operations they know how to do. In traditional mathematics, the integer 2 is "stupid" in the sense that it's just a dumb number. Addition is something we do with numbers, as in 2 + 2, but we don't think of 2 itself as knowing how to implement addition.

But the semantics around objects is different, in that we think of objects as "encapsulating" their various methods. The string 'abc', by virtue of being a string object, knows how to uppercase itself. And so the Pythonic expression 'abc'.uppercase( ) should be read as invoking the uppercase method "inside" the object 'abc'. Objects contain a wealth of information about all sorts of methods, sort of like cells contain DNA.

However, it might seem wasteful for each and every number, of which there are an infinite number (character strings likewise), to carry around inside of itself some copy of the algorithm for addition (subtraction, multiplication and so on). Probably the reason traditional mathematics localizes addition somewhere outside of the individual numbers is based on this intuitive sense that there's only one addition algorithm, whereas there's no limit on the number of numbers.

Python, like other object oriented languages, is sensitive to our sense of economy. Instead of thinking that every number contains its own copy of the addition method, we think of all numbers as instantiating (giving substance to) a single *class*. It's the *class definition* that formalizes addition, subtraction and so on.

Let's think about animals instead of numbers – mammals in particular. All mammals have an analogous skeletal frame, even though the specific shapes of the bones vary greatly. Hippopotami have ribs, as do whales, as do dogs. Of course many non-mammals have ribs as well (such as snakes and chickens), but lets just stick with the mammals for now.

Every specific mammal *object* will be an *instance* of the mammal *class*. In addition, all dogs will be instances of a narrower *subclass* of mammal, namely the dog class. Everything that dogs have in common, but don't share with the other mammals, might be defined in the dog subclass. Attributes and abilities shared with all mammals might be defined at the mammal level.

To make this more concrete, lets look at some actual Python in action. Here's a simple Mammal class:

```
class Mammal(object):
     def eat(self):  print 'Munch'
     def sleep(self):   print 'Zzzzz'
```

Now lets define the Dog class as a subclass of Mammal:

```
class Dog (Mammal):
     def bark(self):  print 'Bark!'
```

These would be the generic blueprints for a Mammal and Dog class respectively. The fact that our Dog class is a subclass of Mammal is denoted by the appearance of Mammal in parentheses after the word Dog in the class header. What this means in practice is that the Dog class *inherits* the Mammal methods (eat and sleep), plus adds one method all its own (bark).

Notice that Mammal actually inherits from a parent class named "object". The "object" class is native to Python and is the root class definition for what are called new-style classes. New-style classes appeared around Python 2.1 and are a result of the effort to better unify Python's type and class semantics. At some point in the future, all classes will automatically be new-style, and the need to explicitly inherit from 'object' will go away (or so I've been led to expect).

Now that we have some class definitions (which are like blueprints), we're ready to instantiate some actual objects.

```
>>> hippo = Mammal()
```

```
>>> hippo.eat()
Munch
>>> fido = Dog()
>>> fido.sleep()
Zzzzz
>>> fido.bark()
Bark!
```

Notice that fido, our dog object, knows how to sleep, even though the Dog class doesn't specifically define a sleep method. This is because all Dog objects inherit the Mammal class's methods. The hippo, on the other hand, does not have a bark method (nor should it). All it knows how to do, so far, is sleep and eat. If we entered the expression hippo.bark() in the above dialog with the Python interpreter, we'd get an error message.

So, going back to numbers, it's as if we had an Integer class definition somewhere, inside of which we defined methods such as addition and subtraction. An integer object, such as 2, would be like a specific dog (like fido). It knows how to add because every integer object has access to its own class definition, as well as to definitions inherited from *parent classes* (also known as *super* classes).

So now you may be starting to see how an object oriented language revolves around the notion of classes, where classes are roughly equivalent to types. In Python, the semantics have evolved to bring the "class" and "type" concepts much closer together, such that we're now able to pretty much equate the two (as mentioned above, so-called new-style classes were a step forward in this regard).

Let's say there's a kind of number that Python does not natively define. A good example would be a set of integers that add modulo 7. Addition modulo 7 works like this: 3 + 3 = 6, 6 + 6 = 5, 4 + 4 = 1, 2 + 5 = 0. Does that make any sense? The rule is: the result is equivalent to the *integer remainder*, once you do ordinary addition and then divide by 7. So, for example, 4 + 4 = 8 in ordinary addition, and 8 divided by 7 leaves a remainder of 1, which is our answer. 6 + 6 = 12, and 12 / 7 yields a remainder of 5, so 6 + 6 = 5.

Clearly this is a special kind of integer. What Python provides to us, as programmers, is an *extensible type system*, meaning that if the type we want doesn't already exist, we have the means to define it, by writing a class definition. Here's one way this might look:

```
class Modint(object):

   modulus = 7

   def __init__(self, value):
       self.value = value % self.modulus

   def __add__(self, other):
       return Modint(self.value + other.value)
```

```
    def __repr__(self):
        return '%s (mod %s)' % (self.value, self.modulus)
```

The three methods here defined all have a strange-looking form: they have double underlines before and after some word or abbreviation. There's a rather long list of such words in Python, and they stand for pre-defined ways in which object methods may be triggered by the syntax.

For example, the plus symbol (+) will trigger the __add__ method.

The __init__ method, on the other hand, is called the *initializer* or *constructor* and gets triggered automatically when an object is first instantiated, by means of the class name followed by parenthesis containing whatever arguments (if any) that __init__ expects.

__repr__ defines how an object will be *represented*. For example, when interacting with the Python shell, just entering the name of a variable by itself on a line will trigger its representation (its __repr__ method).

To make this clearer, take a look at some Modint objects in action:

```
>>> n1 = Modint(6)  # triggers __init__
>>> n2 = Modint(6)  # triggers __init__
>>> n1 + n2         # triggers __add__ and __repr__
5 (mod 7)
>>> n3 = Modint(4)
>>> n3 + n3
1 (mod 7)
>>> n1 + n3
3 (mod 7)
```

We're getting the answers we wanted.

Python furnishes the % operator. x % y returns the remainder when x is divided by y, i.e. 22 % 7 returns 1 and 14 % 7 returns 0. Our class simply builds an addition method (__add__) around this operator. It does ordinary addition (e.g. 6 + 6 = 12), but then passes the sum to Modint, which triggers __init__, the constructor. Within the constructor, 12 gets turned into 5 by means of the expression: self.value = value % self.modulus.

And what is this 'self' we keep seeing? 'self' is a reference to "this object" (whatever object we're working in at the moment). 'self' differentiates one object from another of the same class. In writing self.value = value % self.modulus, we're *binding* an arithmetic result to a specific object (*this* one).

The variable named ‘value’ to the right of the equal sign is simply what got passed in as an argument to __init__. It’s about to go out of scope (to become inoperative), but we have it long enough to do a modulo operation on it, and to get a result. It’s this result that we keep, by binding it to self.value, *self* being the persistent nucleus of any object. self.value persists even after __init__ is finished, as a permanent attribute of our newly instantiated object.

Later, when we add two Modint-type numbers, we’ll be able to retrieve self.value and, because *other* is another Modint (or had better be, if we don’t want an error), we’ll have access to other.value as well (other contains a *self* of its own, which contains *value*).

In other words, every specific Modint has a self. Within this self, our *value* will be stored (we haven’t gotten to dictionaries yet, but when we do, you’ll come to think of self as a kind of dictionary, a container for __dict__).

So we’ve extended the built in type system by using the ordinary integers and operators supplied to us, and organizing them into a class definition for Modint objects.

```
>>> type(n3)
<class '__main__.Modint'>

>>> type(fido)
<class '__main__.Dog'>

>>> type(hippo)
<class '__main__.Mammal'>
```

What the above discloses is that the built-in *type* method, applied to any object, will return what type of object its argument is, along with a prefix identifying in which module the class definition occurs (we have yet to discuss modules – basically they’re the .py files on disk). The interactive shell, which we’ve so far been using, is automatically named __main__ (another of those “special” Python words, with the double underlines).

Up to this point, we’ve used various types of number, plus character strings, as our representative *types*. However, Python natively supplies a much richer variety of indigenous types of the kind programmers have come to value. Among these types are: lists, tuples, dictionaries and sets. Tuples are a lot like lists, and sets are a lot like dictionaries, so we should begin by looking at lists and dictionaries (see Python documentation for more on tuples and sets).

A list is a collection of objects. Syntactically, it’s represented by a pair of square brackets in Python, with the list elements separated by commas. For example:

```
>>> mylist = ['a', 3, ['555', '333'], 10.77 ]
>>> mylist[0]
'a'
```

```
>>> mylist[2]
['555', '333']
>>> mylist[2][1]
'333'
```

What you see here is mylist (a random variable name I made up) being assigned a list of elements. The elements need not all be of the same type. Here we find a mixture of types, including another list. Lists may be elements of other lists. Note that `'555'` is a character string, not a number, thanks to the quotes (single or double quotes would have the same meaning in this context).

Next what you see is retrieval of list elements by index. List elements are addressable using integers, starting with 0. The element with an address of 0 is what we would call the first element of the list – a possible source of confusion at first. On the last line, you'll see two square-bracket-triggered *__getitem__* operations, one right after the other. The first retrieves `['555', '333']`, and the second extracts `'333'` from within that two-element list.

Yes, __getitem__ is one of those Pythonic method names and, behind the scenes, is what gets triggered by square bracket subscript notation. In other words, mylist[2] is another way of invoking the method __getitem__ common to all list objects, passing it the argument 2. To test this, we rewrite the last two lines above using this relatively unwieldy (but nevertheless legal and intelligible) Python syntax:

```
>>> mylist.__getitem__(2)
['555', '333']
>>> mylist.__getitem__(2).__getitem__(1)
'333'
```

Based on the Modint example wherein we defined __add__ and __repr__, you might guess that defining __getitem__ inside a class of our own design would be a way to provide our own behavior for subscript notation vis-à-vis objects instantiated from that class. You would be correct.

Although this is somewhat perverse, let's define a Cat class, inheriting from Mammal, which implements a __getitem__ method.

```
class Cat(Mammal):
     def __getitem__(self, howmany):
          print 'Meow ' * howmany

>>> somecat = Cat()
>>> somecat[4]
Meow Meow Meow Meow
>>> somecat.sleep()  # just to show Mammal methods work too
Zzzzz
```

Now that's rather odd, to have square bracket syntax trigger meowing. But the point is to show how Python provides these hooks giving the programmer ways to *overload operators*.

We've already seen an example of overloading the plus symbol (+). In the Modint class, we defined __add__, and that effectively turned + into a "modulo addition" operator. In the Cat class, we've turned subscript notation (square brackets with an enclosed integer), into a trigger for repeated meowing.

Notice, by the way, the strings may not understand subtraction and division, but they do give meaning to multiplication. *somestring* * *n* is equivalent to concatenating *somestring* with itself *n* times. That makes sense, because multiplication is generally conceived of as "repeated addition". So if it makes sense to "add" strings (in the sense of concatenating them), then it should make sense to "multiply" them (in the sense of concatenating them repeatedly). And so you might think of some String class as defining __add__ and __mul__ internally – just as you might in some class of your own.

I haven't exhausted what list objects are able to do. For example, mylist.sort( ) and mylist.reverse( ) will sort the elements or reverse them in place respectively. Unlike many list methods, these two don't return any results. The operate on the list itself, changing the ordering of the elements. String methods, on the other hand, always return new string objects (we say lists are mutable – changeable -- but strings are not).

We'll come back to lists later, but let's move on to dictionaries, to give a sense of how these differ. Dictionaries are collections of elements, like lists, but they're accessed not by integer indexes, but by "keys" – where keys are any immutable object, such as integers or strings. In other words, a dictionary consists of "key/value" pairs, where values are "looked up" by key, independently of any ordering.

```
>>> adict = {'fido':Dog(), 'kitty':Cat(), 'piggy':Mammal()}
>>> adict['fido']
<__main__.Dog object at 0x00AD94D0>
>>> adict['kitty'][4]
Meow Meow Meow Meow
>>> adict['piggy'].eat()
Munch
```

Here I've paired a set of strings (names) with new objects. I then retrieve the objects using the keys (the strings). adict['fido'] looks a lot like the subscript notation used with lists, and it is, but instead of integers for indexes, we use any immutable key we like (integers would work too, in some other dictionary, but would be independent of any ordering – a dictionary doesn't have a "first" or "last" key/value pair).

adict['fido'] dumps out some information about what sort of object we've retrieved. It's not a very pretty representation of a dog object, because we never bothered to write

behavior for a __repr__ method in Mammal or Dog, and so we inherit the generic behavior from the parent to both Dog and Mammal: object.

adict['kitty'] would return our Cat object, but the expression is adict['kitty'][4], meaning we're immediately triggering the object's __getitem__ method, which nets us a lot of meowing.

adict['piggy'] retrieves a Mammal object (which we presume might be a pig but who knows for sure), while the expression adict['piggy'].eat( ) triggers the Mammal class's eat method. Dog and Cat objects also know how to eat, since they inherit from Mammal.

Now suppose we wanted to retrieve these mammals in reverse alphabetical order. The dictionary doesn't really have this concept (given the term "dictionary" you might think that it would, but this is a more primitive kind of dictionary, also known as a hash table).

However, the keys( ) method will return a list of any dictionary's keys. Once we have a list, we know we can sort it and reverse it. Then we'll be able to loop through these keys in the desired order, pulling mammal objects accordingly:

```
>>> thekeys = adict.keys()  # grab the keys
>>> thekeys
['kitty', 'fido', 'piggy']
>>> thekeys.sort()          # alpha sort in place
>>> thekeys
['fido', 'kitty', 'piggy']
>>> thekeys.reverse()       # now reverse in place
>>> thekeys
['piggy', 'kitty', 'fido']
>>> for key in thekeys:     # iterate thru keys for values
       print adict[key]


<__main__.Mammal object at 0x00AD9AD0>
<__main__.Cat object at 0x00AD9AB0>
<__main__.Dog object at 0x00AD9ED0>
```

Notice that the keys( ) method isn't required to return the keys in any particular order – so long as they're all there, this method has done its assigned task.

So to get our keys in reverse alphabetical order, we first want to sort the list in place, then reverse it. Our list assigned to the variable thekeys gets rearranged with each step. Lastly, we're ready to employ a *loop structure*, making the variable *key* become each element of thekeys (a list, not a dictionary) in turn. This variable *key* (we could have called it anything, but *key* makes sense) is now used to invoke the dictionary's __getitem__ method, i.e. we use subscript syntax to pull our objects out of the dictionary.

The retrieved objects dump their default __repr__ output (inherited from object) to the screen.

Remember awhile back we talked about turning objects of one type into objects of another? Well, a list of two-element lists may be turned into a dictionary, by feeding it to the generic dict type:

```
>>> alist = [['fido',Dog()], ['kitty',Cat()]]
>>> newdict = dict(alist)

>>> for item in newdict.items():
        print item[0] + " : " + repr(item[1])


kitty : <__main__.Cat object at 0x00AD9790>
fido : <__main__.Dog object at 0x00AD95B0>
```

dict(alist) actually has a somewhat different flavor than we've encountered so far. This is because dict is the generic, built-in dictionary type, which all dictionaries inherit from. And yet we're able to pass a list (of lists) to this generic dict type as an argument.

In the case of dict(listofpairs), the syntax is triggering one of those "special methods," in this case __call__. This is distinct from __init__, which is triggered as an object is being created.

Ordinary functions, outside of any classes, are examples of "callables" (objects that support __call__ syntax). For example:

```
>>> def f(x):  return x*x

>>> callable(f)
True
>>> f.__call__(10)
100
>>> f(10)
100
```

f(x) is just some random function (it multiplies its numeric argument by itself). In that it executes when triggered by parentheses (in this case containing an argument – but other functions might not take arguments), it's "callable," and asking if it's callable (using the built-in callable function) returns a True.

It's awkward, but legal, to write f.__call__(10), to trigger f with an argument of 10. But of course the ordinary way to write this would be f(10), as shown on the last line.

So dict, a generic type, supports call, i.e. is callable:

```
>>> dict
<type 'dict'>

>>> callable(dict)
True

>>> dict.__call__
<method-wrapper object at 0x00AD9F30>

>>> dict.__call__(alist)
{'kitty': <__main__.Cat object at 0x00AD9790>, 'fido':
<__main__.Dog object at 0x00AD95B0>}

>>> dict(alist)
{'kitty': <__main__.Cat object at 0x00AD9790>, 'fido':
<__main__.Dog object at 0x00AD95B0>}
```

In general, generic types that support __call__ (are callable), use the associated method to turn inputs into objects of the type in question. The generic integer object (int) turns floating point numbers into integers. The generic string object (str), turns numbers into strings. The generic list object (list) turns any iterable (anything one might iterate over, such as a string), into a list.

```
>>> int(10.3)
10
>>> callable(str)
True

>>> str(4444)
'4444'

>>> list('abcd')
['a', 'b', 'c', 'd']

>>> str           # supplied by Python
<type 'str'>

>>> type(str)     # str is one of several built-in types
<type 'type'>

>>> type(Mammal)  # Mammal is a user-defined type
<type 'type'>
```

And so there you have it. We've been looking at Python fundamentals, which is a somewhat abstract and theoretical way of exploring the language's design. The point of

this investigation has been to develop what we mean by such terms as “class” “object” and “type”.

What Python makes “easy” (relatively) is a way of thinking in which programmers start with built-in or indigenous types, and then extend the type system by means of class definitions, to include whatever additional types a given programming task might suggest.

We initially grasp this notion of “types” by thinking of different types of number, and then realize that “number” is too narrow a focus, for the concept of “type” we need. So in place of “number” we think of “objects”. And these objects, unlike the traditional idea of numbers, are considered to contain information about their operations, as well as information about how to convert between types. The information is centralized in class definitions, which objects of the same type all share.

Each object also has a “self”, which is able to store information unique to that object (such as a name or value). The “self” is a reference to a specific object. When we invoke a method or try to retrieve a value from an object, it consults its self first, then looks to the class definition for methods bound to self (the implicit first argument, unless it’s a class or static method – notions we’ve yet to develop). Then, if the method still isn’t found, it begins to search the parents of the class, i.e. those classes from which the object’s class inherits.