Object-oriented programming with Python

An introduction into OOP and design patterns with Python

Max Leuthäuser
s7060241@mail.zih.tu-dresden.de
TU Dresden

Abstract

While programming itself is a relatively young discipline, you may find it surprising that object-oriented programming (OOP from now on) actually goes back as far as the 1960's. Simula[1] is considered to be the first object-oriented programming language. This paper tries to dive quickly into OOP with Python and give a small introduction into important basics. Later we find more advanced techniques with design patterns. This paper assumes a minimal knowledge of Python syntax. If you know how to create the basic data-types and call functions, but want to know more about objects, classes, design patterns and advanced built-in language extensions you may find this helpful.

Introduction

OOP itself is a controversial paradigm, in fact it is not even clear what a strict definition of OOP *is*. Actually some people think that OOP has nearly had its days, and we need to find the post OOP paradigm[2]. So it is not very surprising that others like Paul Graham[3] think it was never necessary in the first place. But in general OOP is basically a standard. Most lines of code nowadays are written with object-oriented (and semi object orientated) languages[4]. It is not the task of this paper to underline all the advantages of Python now, but some basic facts are important. The whole design philosophy of Python encourages a clean programming style. Its basic datatypes and system of namespaces makes it easier to write elegant, modular code[5]. These factors and the unique block structure by indentation rules are very helpful for reading and understanding code. Actually the basic principles of object-oriented programming are relatively easy to learn. Like other aspects of Python they are nicely implemented and well thought out.

The remaining part of this paper is organized as follows: The next two sections introduce the object orientated programming paradigm and its basics - especially with Python. After this some important facts with inheritance and additional language extensions are discussed. This is concluded with some simple measurements and some broader observations on using design patterns in Python. This paper is full of examples written in Python and C++ and/or Java. Feel free to compile them, they may help to understand what was meant in the text.

[1] http://en.wikipedia.org/wiki/Simula

[2] http://www.americanscientist.org/Issues/Comsci03/03-03Hayes.html

[3] http://www.paulgraham.com/noop.html

[4] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[5] Python's system of modules and packages as well - these are related to namespaces, and are the envy of other languages.

1. The Basics

As mentioned above you should already be familiar with basics like the following ones, so this is simply a small summary to repeat and keep things in mind. Advanced users might skip this and continue reading the "Advanced language extensions" part or dive directly into "Design Patterns with Python".

1.1 What is an object?

We are in the domain of programming - so an object is a concept. Calling elements of our programs objects is a metaphor - a useful way of thinking about them. In Python the basic elements of programming are things like strings, dictionaries, integers, functions, and so on. They are all objects[6].

1.2 A small reference to procedural programming

Using a procedural style of programming, programs are often split into reusable "chunks" called functions and methods. The resulting code is easier to maintain. These chunks are callable at different places in your program so improving one method or function will enhance the performance on several places. Another important fact to keep in mind when talking about OOP comes from procedural programming as well. It maintains a strict separation between your code and your data. You have variables, which contain your data, and procedures. You pass your variables to your procedures - which act on them and perhaps modify them. If a function wants to modify the content of a variable by passing it to another function it needs access to both the variable and the function it's calling. If you are performing complex operations this could be lots of variables and lots of functions.

1.3 Talking about python basics

Traditionally essential elements of OOP are encapsulation and message passing. The Python equivalents are *namespaces* and *methods*. Python does not subscribe to protecting the code from the programmer, like some of the more BSD languages. Python does encapsulate objects as a single namespace but it is a translucent encapsulation. The following introductions are visualized using the string object as an example.

1.3.1 Using objects

As already mentioned above it turns out that lots of operations are common to objects of the same type. For example most languages have built-in ways to create a lowercase version of a string. Actually there are a lot of standard operations to interact with strings. Building an uppercase version, splitting and concatenating, and so on. In an object-oriented language we can build these in as properties of the string object. In Python we call these methods. *Every*

[6] Like Smalltalk - which is sometimes spoken of as the archetypal object-oriented language.

string object has a standard set of methods. This is true for all kinds of objects in Python. Consider the following example (code taken from [1]):

```
original_string = ' some text '

# remove leading and trailing whitespace
string1 = original_string.strip()

# make uppercase
string2 = string1.upper()
print string2
> SOME TEXT

# make lowercase
string2.lower() == string1
> True
```

Python uses the dot syntax (like Java or C#) to access attributes of objects. The statement *string2.lower()* means call the lower method of the object *string2*. This method returns a new *string* - the result of calling the method. So every string is actually a string object - and has all the methods of a string object. In Python terminology we say that all strings are of the string type. In the object model the functions (methods) and other attributes that are associated with a particular kind of object become part of the object. The data and the functions for dealing with it are no longer separate - but are bound together in one object.

1.3.2 Creating objects

In Python there is a blueprint string object called the string type. Its actual name is *str*. It has all the methods and properties associated with the string as already described above. Each single time a new string is created, the blueprint is used to create a new object with all the properties of the blueprint. All the built-in datatypes have their own blueprint, e.g. the integer (*int*), the float (*float*), booleans (*bool*), lists (*list*), dictionaries (*dict*), and some more. For this datatypes it is possible to use the standard Python syntax to create them or we can use the blueprint itself - the type.

```
# create a dictionary the normal way
a_dict = {
    'key' : 'value',
    'key2' : 'value2'
    }
# use 'dict' to create one
list_of_tuples = [('key', 'value'),
                  ('key2', 'value2')]
a_dict_2 = dict(list_of_tuples)
#
print a_dict == a_dict_2
True
print type(a_dict)
<type 'dict'>
print type(a_dict_2)
<type 'dict'>
```

a_dict_2 was created by passing a list of tuples to *dict*. Very basic, but it illustrates that new objects are created from blueprints. These objects have all the methods defined in the blueprint. The new object is called an *instance* and the process of creating it is called instantiation (meaning creating an instance). For the built-in datatypes the blueprint is known as the type of the object. You can test the type of an object by using the built-in function type.[7]

[7] Actually type is not a function - it is a type. Exactly the type of types (type(type) is type). It is used as function here.

Last thing in this section is making use of some simple *dictionary* methods.

```
a_dict = {
    'key' : 'value',
    'key2' : 'value2'
    }
a_dict_2 = a_dict.copy()
print a_dict == a_dict_2
True
a_dict.clear()
print a_dict
{}
print a_dict.clear
<built-in method clear of dict object at 0
    x0012E540>
print type(a_dict.clear)
<type 'builtin_function_or_method'>
```

The clear method of *a_dict* was used by calling *a_dict.clear()*. When we printed *clear*, instead of calling it, we can see that it is just another object. It is a method object of the appropriate type.

1.3.3 Using functions as objects

To demonstrate that functions can be objects consider the following code. You may have written something like this before in some other language:

```
if value == 'one':
    # do something
    function1()
elif value == 'two':
    # do something else
    function2()
elif value == 'three':
    # do something else
    function3()
```

Other languages (e.g. Java) have a construct called *switch* that makes writing code like that a bit easier. In Python we can achieve the same thing (in less lines of code) using a dictionary of functions. As an example, suppose we have three functions. You want to call one of the functions, depending on the value in a variable called *choice*.

```
def function1():
    print 'You chose one.'
def  function2():
    print 'You chose two.'
def  function3():
    print 'You chose three.'

# switch is our dictionary of functions
switch = {
    'one': function1,
    'two': function2,
    'three': function3,
    }

# choice can either be 'one', 'two', or '
    three'
choice = raw_input('Enter one, two, or three
    :')

# call one of the functions
try:
    result = switch[choice]
except KeyError:
    print 'I did not understand your choice.
        '
else:
    result()
```

The point of interest is: *result = switch[choice]*. The *switch[choice]* returns one of our function objects (or raises a *KeyError*). Finally *result()* is called which does the trick.

1.4 Making use of user defined classes

Much more interesting is that we can create our own blueprints. These are called classes. We can define our own class of object - and from this create as many instances of this class as we want. All the instances will be different - depending on what data they are given when they are created. They will all have the methods (and other properties) from the blueprint. Take a look at the following simple example. We define our own class using the *class* keyword. Methods are defined like functions - using the *def* keyword. They are indented to show that they are inside the class.

```
class OurClass(object):
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def printargs(self):
        print self.arg1
        print self.arg2
```

Now it is easy to create an own instance:

```
instance = OurClass('arg1', 'arg2')
print type(instance)
<class 'OurClass'>
instance.printargs()
arg1
arg2
```

The arguments *arg1* and *arg2* are passed to the constructor as arguments. Later they are printed using the method *printargs()*. To get a deeper understanding what happened here we need to know more about the constructor and the things which are related to this.

1.4.1 The __init__ method

The __init__ method (init for initialise) is called when the object is instantiated. Instantiation is done by (effectively) calling the class. It is also known a the *constructor* as mentioned above. This is very similar to Java where the constructor has always the name as the class itself followed by the list of the arguments surrounded by brackets. The only thing leftover now is the confusing self parameter.

1.4.2 Understanding the self parameter

The arguments accepted by the __init__ method (known as the method signature) are:

```
def __init__(self, arg1, arg2):
```

This is a little bit confusing. When we initiate the object we actually pass only two arguments:

```
instance = OurClass('arg1', 'arg2')
```

So where does the extra argument comes from? When we access attributes of an object we do it by name (or by reference). Here *instance* is a reference to our new object. We access the *printargs* method of the instance object using *instance.printargs*. In order to access object attributes from within the __init__ method we need a reference to the object within the method of the object itself. Whenever a method is called, a reference to the main object is passed as the first argument. By convention you always call this first argument to your methods *self*.

1.5 Conclusions for this section

So what are the advantages of using OOP and what is so useful using objects?

Objects combine data and the methods used to work with the data. So it is possible to wrap complex processes - but present a simple interface to them. How these processes are done inside the object becomes a mere implementation detail. Anyone using the object only needs to know about the methods and attributes. This is the real principle of encapsulation. Other parts of your application (or even other programmers) can use your classes and their public methods - but you can update the object without breaking the interface they use.

You can also pass around objects instead of data. This is one of the most useful aspects of OOP. Once you have a reference to the object you can access any of the attributes of the object. If you need to perform a complex group of operations as part of a program you could probably implement it with procedures and variables. You might either need to use several global variables for storing states (which are slower to access than local variables and not good if a module needs to be reusable within your application) - or your procedures might need to pass around a lot of variables.

If you implement a single class that has many attributes representing the state of your application, you only need to pass around a reference to that object. Any part of your code that has access to the object can also access its attributes.

The main advantage of objects is that they are a useful metaphor. It fits in with the way we think. In real life objects have certain properties and interact with each other. The more our programming language fits in with our way of thinking, the easier it is to use it to think creatively [1].

To go back to Python itself the "everything is an object" mentality should be mentioned.

2. Everything is an object

As you now know already from above everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute *__doc__*, which returns the doc string (short usage documentation of the function) which is defined in the function's source code. The *sys* module is an object having an attribute called path. And so forth.

In Python, the definition for "object" is quite weak; some objects have neither attributes nor methods and not all objects are subclassable. But everything is an object in the sense that it can be assigned to a variable or passed as an argument to a function.

2.1 First class functions

Simply spoken first class functions allow run-time creation of functions from functions. It do not have to be members of some class. The exact definition is something like this[8]: In computer science, a programming language is said to support first-class functions (also called function literals, function types) if it treats functions as first-class objects. Specifically, this means that the language supports constructing new functions during the execution of a program, storing them in data structures, passing them as arguments to other

[8] http://en.wikipedia.org/wiki/First-class_function

functions, and returning them as the values of other functions. This concept does not cover any means external to the language and program (meta programming), such as invoking a compiler or an eval function to create a new function.

2.2 Meta programming with on-the-fly method patching

On-the-fly method patching, also known as *bind unbound methods* is some kind of attaching any method (which is actually a method object) at a specific class in the way of meta programming. At this part I do not want to make use of some fancy explanation or definition[9]. A simple example should visualize the functionality:

We simply create a class

```
class A:
        b = 3
```

and create an instance.

```
a = A()
print a.b
3
```

Now we create a first-class function.

```
def f(self, a):
        return a + self.b
```

Finally we install that function in the class.

```
A.fred = f
```

So calling that function on the instance of A() does the trick.

```
print a.fred(4)
7
```

3. Inheritance

3.1 Single inheritance

Of course Python supports inheritance and even a limited form of multiple inheritance as well. The syntax looks basically like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

The name *BaseClassName* must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName
    ):
```

Execution of a derived class definition is actually the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class [2].

Instantiation of derived classes is simple: *DerivedClassName()* creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it.

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call *BaseClassName.methodname(self, arguments)*. This is occasionally useful to clients as well[10].

Python has two built-in functions that work with inheritance [2]:

- Use *isinstance()* to check an instance's type: *isinstance(obj, int)* will be True only if *obj.__class__* is int or some class derived from int.
- Use *issubclass()* to check class inheritance: *issubclass(bool, int)* is True since bool is a subclass of int. However, *issubclass(unicode, str)* is False since unicode is not a subclass of str (they only share a common ancestor, basestring).

3.2 Small insertion - private variables

"Private" instance variables that cannot be accessed except from inside an object do not exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice [2].

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling [11]. Any identifier of the form __spam (at least two leading underscores, at most one trailing underscore) is textually replaced with _classname__spam, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger. These rules are comparable to the Java keyword *private* but are completely voluntarily and are the responsibility of of the programmer.

3.3 Multiple Inheritance

A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    ...
    <statement-N
```

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in *DerivedClassName*, it is searched in *Base1*, then (recursively) in the base classes of *Base1*, and only if it is not found there, it is searched in *Base2*, and so on.

[9] For more information see:
http://bitworking.org/news/Python_isnt_Java_without_the_compile

[10] Note that this only works if the base class is accessible as BaseClassName in the global scope.

[11] http://en.wikipedia.org/wiki/Name_mangling

For new-style classes[12], the method resolution order changes dynamically to support cooperative calls to *super()*. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages. (The *super()* keyword is explained in the next section.)

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where one of the parent classes can be accessed through multiple paths from the bottommost class - this is known as the *diamond problem*). For example, all new-style classes inherit from object, so any case of multiple inheritance provides more than one path to reach object. To keep the base classes from being accessed more than once, the dynamic algorithm serialises the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance[13].

3.4 The super() function

Definition from [3]: This function returns a proxy object that delegates method calls to a parent or sibling class of type. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by getattr()[14] except that the type itself is skipped.

To understand why this is useful consider the following two use cases:

- In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of *super* in other programming languages.
- If you want to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement "diamond diagrams" where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case[15]. [3]

In both cases a typical superclass looks something like this:

```
class C(B):
    def method(self, arg):
        super(C, self).method(arg)
```

There are two important things to mention:

- *super()* is implemented as part of the binding process for explicitly dotted attribute lookups such as *super().__getitem__(name)*. It does so by implementing its own *__getattribute__()* method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, *super()* is undefined for implicit lookups using statements or operators such as *super()[name]*.
- *super()* is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references.

  This function is new in Python version 2.2 and only usable in new-style classes as mentioned above.

4. Language extensions

Of course it would be possible to use Python as any other object orientated language. But there are some fancy extensions you should know to write more cleaner and readable code in less lines than you would expect. The following subsections should give a small introduction into the simplifications which came from using these language extension.

4.1 Lambda functions

Python supports an interesting syntax that lets you define one-line mini-functions on the fly. Borrowed from Lisp, these so-called *lambda* functions can be used anywhere a function is required.

```
''' 1) The standard way '''
def f(x):
        return x*2
>>> f(3)
6

''' 2) Using lambda as anonymous function
    bound to g '''
g = lambda x: x*2
>>> g(3)
6

''' 3) Using lambda without assigning '''
(lambda x: x*2)(3)
6
```

1. This is the standard way. A function is defined with the keyword *def*.
2. This is a lambda function that accomplishes the same thing as the normal function above it. Note the abbreviated syntax here: there are no parentheses around the argument list, and the return keyword is missing (it is implied, since the entire function can only be one expression). Additionally, the function has no name, but it can be called through the variable it is assigned to.
3. You can use a lambda function without even assigning it to a variable. This may not be the most useful thing in the world, but it just goes to show that a lambda is just an in-line function. [4]

Summarizing, a lambda functions is simply a function which takes any number of arguments and return the value of exactly one single expression. *Please note:* lambda functions are a matter of style. You do not need to use them. They are never required! Only use them in places where you want to encapsulate specific, non-reusable code without littering your code with a lot of little one-line functions. Do not try to squeeze too much into a lambda function; if you need something more complex, define a normal function instead and make it as long as you want.

4.2 List comprehensions

List comprehensions provide a easy way to create lists without making use of special functions. (such as *map()*, *filter()* and/or

[12] Any class which inherits from object. This includes all built-in types like list and dict. Only new-style classes can use Python's newer, versatile features like __slots__, descriptors, properties, and __getattribute__(). More information can be found here: http://docs.python.org/reference/datamodel.html#newstyle

[13] For more detail, see http://www.python.org/download/releases/2.3/mro/.

[14] http://docs.python.org/library/functions.html#getattr

[15] Because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime.

lambda) The resulting list is often more clearer. Each list comprehension consists of an expression followed by a for clause, then zero or more for or if clauses. The result will be a list resulting from evaluating the expression in the context of the *for* and *if* clauses which follow it. If the expression would evaluate to a tuple, it must be parenthesized. Here are some simple, self explaining examples[16]:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]

>>> [3*x for x in vec if x > 3]
[12, 18]

>>> [3*x for x in vec if x < 2]
[]

>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]

>>> [x, x**2 for x in vec] # error - parens
    required for tuples
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax

>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]

>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]

>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]

>>> [vec1[i]*vec2[i] for i in range(len(vec1
    ))]
[8, 12, -54]
```

It gets even more interesting when making use of nested list comprehensions. They are a powerful tool but like all powerful tools they need to be treated carefully, if at all. I do not want to go too deep with this so see the following example to get a basic understanding:

```
mat = [ [1, 2, 3], [4, 5, 6], [7, 8, 9], ]
print [[row[i] for row in mat] for i in [0,
    1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

As you can see this example will swap the rows of the matrix. In general try to read from right to left to avoid apprehension when analysing nesting list comprehensions.

4.3 Closures

If you are already practiced with some functional programming language you might know closures yet but they are sufficiently interesting to many developers which work with nonfunctional languages like Perl or Ruby too. These languages include closures as a feature already. Moreover, since Python 2.1 lexical scope closures are added which provide most of the capabilities of closures in general.

So *what* is a closure, anyway? There was a nice characterization of the concept on the Python newsgroup by Steve Majewski[17]: That is, a closure is something like FP's Jekyll to OOP's Hyde (or perhaps the roles are the other way around). A closure, like an object instance, is a way of carrying around a bundle of data and functionality, wrapped up together.

So closures are an incredibly useful language feature. They let us do clever things that would otherwise take a lot of code, and often enable us to write code that is more elegant and more clear. In Python, closures are read-only affairs; that is, a function defined inside another lexical scope cannot change variables outside of its local scope. To get a better understanding what this mean and to see how a closure is working in Python consider the following example. We want to implement something like a simple counter which prints an increasing number each time we call it. So the naive approach would maybe something like this:

```
def counter():
    count = 0
    def c():
        count += 1
        return count
    return c

g = counter()
print g(), g(), g()
```

You expect now some result like "1 2 3". But actually it fails with this:

```
Traceback (most recent call last):
  File "src/counter.py", line 9, in <module>
    g()
  File "src/counter.py", line 4, in c
    count += 1

UnboundLocalError: local variable 'count'
    referenced before assignment
```

This is now where you can see that the lexical scope cannot change the variables outside of its local scope. In Python 3 there is a new keyword for this: *nonlocal* which solves this problem. But with Python 2.7 we have to be a bit more clever. There exist two simple ways to work around. First is the *array idiom*:

```
def counter():
    curr = [0]

    def c():
        curr[0] += 1
        return curr[0]

    return c

g = counter()
print g(), g(), g()
```

Second is some smarter and much more readable way. It makes use of a function attribute.

```
def counter():
```

[16] Taken from http://docs.python.org/tutorial/datastructures.html#list-comprehensions

[17] Original newsgroup post is not available, but it can be found here: http://www.ibm.com/developerworks/library/l-prog2.html

```
    def c():
        c.count += 1
        return c.count
    c.count = 0
    return c

g = counter()
print g(), g(), g()
```

Both print "1 2 3" as expected.

In summary the use of closures can save you of writing a bunch of lines to solve some problem. But take care of scoping problems.

4.4 Decorator

To understand decorators there are some basics which have to be explained first. Some are outlined in other sections of this paper, others are important consequences.

- Functions are objects in Python.
- A function can be defined inside another function.
- Functions can be assigned to variables.
- Therefore a function can return another function.
- Functions can be passed as parameters to other functions and they can be executed there.

Now we have everything to understand decorators. They are wrappers, meaning they let you execute code before and after the function they decorate without the need to modify the function itself. See the following example which does this manually:

```
def mydecorator(a_function_to_decorate) :
    def wrapper() :
        print "Before the function runs"
        a_function_to_decorate()
        print "After the function runs"
    return wrapper

def stand_alone_function() :
    print "I am a stand alone function, don'
        t you dare modify me"

f_decorated = mydecorator(
    stand_alone_function)
f_decorated()
```

This will generate the following output:

```
Before the function runs
I am a stand alone function, don't you dare
    modify me
After the function runs
```

Now the same thing with the Python decorator syntax:

```
@mydecorator
def another_stand_alone_function() :
    print "Leave me alone"

another_stand_alone_function()
```

Basically this ends with the same output:

```
Before the function runs
Leave me alone
After the function runs
```

That's it. So @*mydecorator* is just a shortcut. In conclusion decorators are just a pythonic variant of the decorator design pattern. Of course you can do some fancy advanced things with decorators - I will not explain them at this point[18]:

- You can cumulate decorators. (order matters)
- You can pass arguments to decorators.
- You can decorate decorators with decorators.

Please keep the following best practices in mind when using decorators:

- They exists only since Python 2.4, therefore be sure that is what your code is running on.
- Decorators slow down the code.
- You can not undecorate a function. So once a function is decorated, it's done. For *all* the code.
- Decorators wrap functions, making them hard to debug.

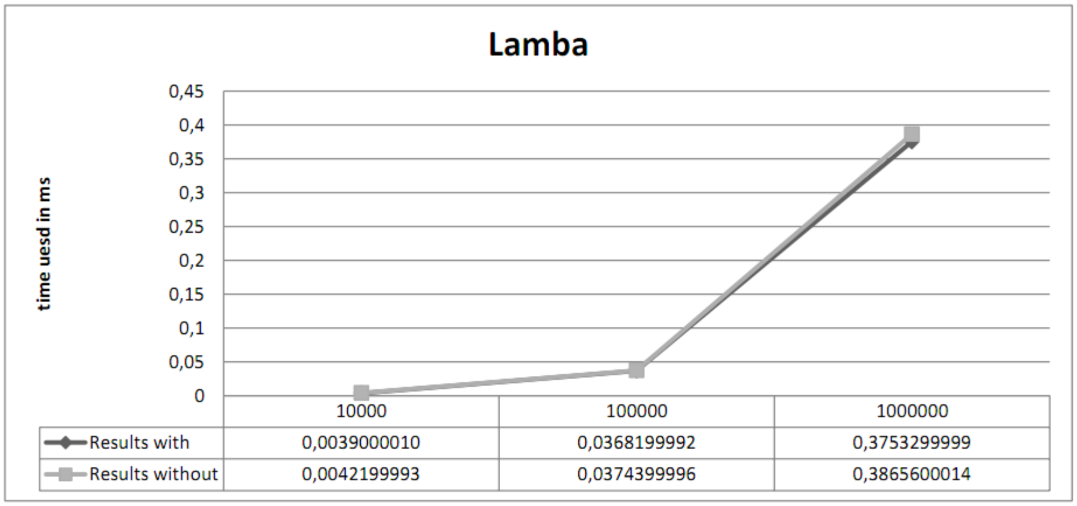
In practice classic uses are extending a function behaviour from an external library (you can not modify it) or for a debug purpose (you do not want to modify it because it's temporary). And probably you can use them to extend several functions with the same code without rewriting it every time.

4.5 Performance measurements

These benchmarks (Figure 1-3) should measure the performance for three language extensions which are explained above, in comparison to an equivalent implementation without them. The values on the x-axis define the size of the test data, e.g. the length of a list the function using a list comprehension has to work on. Only the Closures seems to be a bit slower and according to this measurements there is no big performance variation between using an implementation that makes use of lambda functions or list comprehensions in comparison to one that does not.

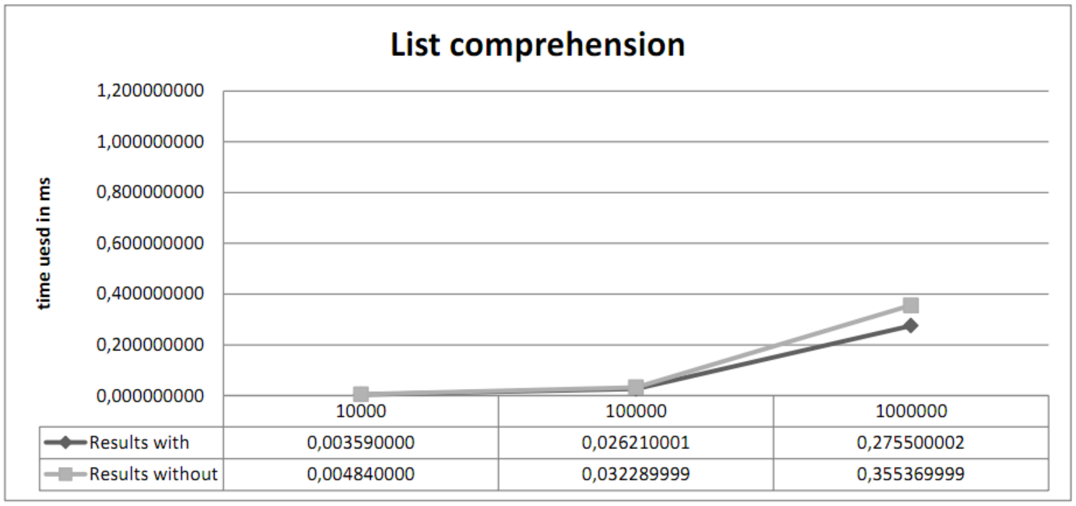
But nevertheless Python is barely usable in high performance computing and with the usage of these language extensions it could gets worse. But as mentioned above those implementations are never indispensable - all three used test samples demonstrate that. They can be found at "Additional material" at the last section.

[18] See http://stackoverflow.com/questions/739654/understanding-python-decorators#answer-739665 if want to know more about decorators.



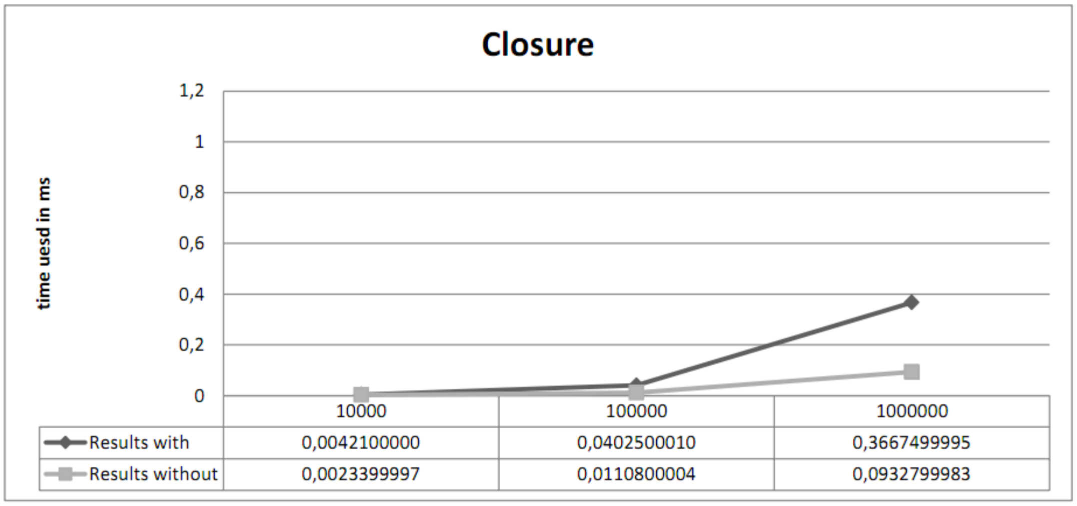
	10000	100000	1000000
◆ Results with	0,0039000010	0,0368199992	0,3753299999
■ Results without	0,0042199993	0,0374399996	0,3865600014

Figure 1 Comparing the time used for executing a function with/without lambda.



	10000	100000	1000000
◆ Results with	0,003590000	0,026210001	0,275500002
■ Results without	0,004840000	0,032289999	0,355369999

Figure 2 Comparing the time used for executing a function with/without a list comprehension.



	10000	100000	1000000
◆ Results with	0,0042100000	0,0402500010	0,3667499995
■ Results without	0,0023399997	0,0110800004	0,0932799983

Figure 3 Comparing the time used for executing a function with/without a closure.

5. Design patterns in Python

In software engineering the term pattern describes a proven solution to a common problem in a specific context. Patterns can be divided into three different categories depending on their level of abstraction and implementation language independency: architectural patterns, design patterns and idioms[5]. I want to concentrate on two categories: design patterns as they are described in what is known as the Gang-of-Four book (GOF from now on)[7] and Python language idioms. The patterns are not only micro architectural models but also useful as a common design vocabulary among software engineers. The overall architecture of the system and related design decisions can be explained by giving a set of patterns used. While new patterns emerge the GOF still remains as the definite reference on design patterns. For this reason it is important to introduce these patterns, the notions and the theory behind them and their applicability to Python community.

GOF is divided to the three parts and each part describes the patterns related to the theme of the part. The themes describe the purpose of the patterns. *Creational* patterns address object instantiation issues. *Structural* patterns concentrate on object composition and their relations in the runtime object structures. Whereas the structural patterns describe the layout of the object system, the *behavioural* patterns focus on the internal dynamics and object interaction in the system.

While the design patterns strive to be language independent they still require - at least implicitly - some support from the implementation language and especially from its object model. In GOF the languages of choice are C++ and Smalltalk. Therefore the availability of access specifiers and static member functions (class methods) are assumed. The aim here is to look at some GOF patterns and try to implement them in Python whose object model is very different from that of C++. At some point I will make a comparison to a common Java implementation too.

Please note: this section is not designed to give deep and widespread explanations for what design patterns are and why you should use them. So it is assumed that you already know the basics about these patterns. If not you should read the GOF book. I do not want to let you alone so I tried to give a small lead-in at the first lines of every section.

5.1 Creational Patterns

As mentioned above these patterns abstract the process of instantiating objects. They help making a system independent of how its objects are created, composed, and represented. The so called *class creational pattern* uses inheritance to vary to the class that is instantiated while an *object creational pattern* will delegate instantiation to another object.[7]

These patterns are very closely related, so I have chosen the following two exemplary to highlight their properties and how they are usable with Python.

5.1.1 Singleton and Borg

The Singleton pattern provides a mechanism to limit the number of the instances of the class to one. Thus same object is always shared by different parts of the code. Singleton can be seen as a more elegant solution to a global variable because actual data is hidden behind Singleton class interface. First we look at how Singleton can be implemented in C++ and Java and then provide a Python solution with the same features. This would be the solution for C++:

```
class Singleton {

public:
  static Singleton& Handle();
private:
  static Singleton* psingle;
  Singleton();
  Singleton( const Singleton& );
  Singleton& operator=( const Singleton& );
};

Singleton& Singleton::Handle() {

  if( !psingle ) {
    psingle = new Singleton;
  }
  return *psingle;
}
```

Our minimal Singleton class definition has only one public member: class method Handle. The defined interface is not very useful as such, because the client can do nothing with the instance, but now we concentrate only on the techniques that ensure only one Singleton instance at a time. The private constructor forbids creating an object outside the class. Thus the clients of the class have to rely on the static member function Handle: it either creates a new instance or returns a reference to the Singleton object.

In Java we would use the state of the art *enum singleton pattern* which makes it possible to do it without static variables.[13]

```
public enum Singleton {
    INSTANCE;
    ...
}
```

Because Python has no private constructors we have to find some other way to prevent instantiations:

```
class Singleton:
    __single = None
    def __init__( self ):
        if Singleton.__single:
            raise Singleton.__single
        Singleton.__single = self
```

The approach now is to raise an exception if the Singleton object is already instantiated (private class attribute *__single* is other than *None*). The exception object is the Singleton instance! The downside of this solution is that it requires the clients to be prepared for the exception. It could also be argued that using exceptions for other purposes than error handling is not very good style. These concerns can be avoided to some extent by providing a similar Handle function than in the example above. This function hides the unorthodox instantiation details and returns a reference to the Singleton object.[11]

Due to Python's type system the code works for the whole Singleton hierarchy. Because there exists an is-a relationship between base class and subclass there can be only one instance of the whole hierarchy. To solve this problem in Python we use the Borg pattern. This pattern allows multiple class instances, but shares state between instances so the end user cannot tell them apart. Here is the Borg example from the Python Cookbook.[12]

```
class Borg:
    __shared_state = {}
    def __init__(self):
        self.__dict__ = self.__shared_state
```

Problems with the Borg as a pattern start when you begin to write new style classes. The *__dict__* attribute is not always assignable, but worse any attributes defined within *__slots__* will simply not be shared. The Borg is cool, but it is not your friend.

There is another implementation of Singleton which is even simpler than the one given above. Just use *import Singleton*. You can access the singleton object using the import statement. You can set and access attributes on the object. Obviously in real life you might want a few methods, or some initial values, so just put them in the module. **Python modules are Singleton instances**: another case of Python taking a design pattern and making it a fundamental part of the language.

5.1.2 Factory

The most fundamental of patterns introduced by the GOF are probably the Factory and Abstract Factory. The Factory pattern in a language such as C++ wraps the usual object creation syntax *new someclass()* in a function or method which can control the creation. The advantage of this is that the code using the class no longer needs to know all of the details of creation. It may not even know the exact type of object it has created. In other words it reduces the dependencies between modules. A more advanced form of factory (Abstract Factory) provides the extra indirection to let the type of object created vary.

The factory pattern is fundamental in Python: where other languages use special syntax to indicate creation of an object, Python uses function call syntax as the (almost) only way to create any object: some of the builtin types such as int, str, list, and dict, have their own special syntax, but they all support factory construction as well. (these types were mentioned already in the basics above) Moreover, Python uses Abstract Factory for everything. The dynamic nature of the system means that any factory may be overridden.[14] See the following code for example:

```
import random

def listOfRandom(n):
        return [random.random() for i in
            range(n)]
```

At first sight it looks as though this function will return a list of pseudo-random numbers. However you can reassign random at the module level, and make it return anything you wish. Although at first this may sound like a crazy thing to do, in fact it is one of the reasons why Python is such a great language for writing unit tests. It is hard to write an automated test for a function with a pseudo-random result, but if you can temporarily replace the random number generator with a known, repeatable, sequence, you can have repeatable tests. Python makes this easy.

In Java it would be something similar to this but please keep in mind: this is actually no real factory pattern in Java just because you are not able to do the reassignment at module level here.[19]

```
public List<Integer> listOfRandom(int n) {
        List<Integer> result = new
            LinkedList<Integer>();
        Random randomGenerator = new Random
            ();
        for (int i=0; i<n; i++)
                result.add(randomGenerator.
                    nextInt(100));
        return result;
}
```

It is hard to say whether this really counts as a pattern in Python (or Java) at all. At one level it is basic to the language, and does not involve actual code. On the other hand, the pattern is so well known that it is important to acknowledge that it corresponds to the Factory pattern.

Python 2.2 introduced a new way to control object creation. New-style objects (where object is the base class, you might remember from an earlier section) allow a method *__new__* to control the actual creation of the object. This may be seen as another form of the factory pattern, and one where there is actual code necessary to implement it. You have seen this in the singleton section above.

5.2 Behavioural Patterns

Behavioural design patterns are those that identify common communication patterns between objects and realize them. By doing so, these patterns increase flexibility in carrying out this communication.

5.2.1 Iterator

This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. The iterator pattern is one which Python has fully embraced, albeit in a slightly simpler form than the one proposed by the GOF. Latter have the following methods:

- First()
- Next()
- IsDone()
- CurrentItem()

Python's iterator interface requires the following methods to be defined:

- __iter__() - Returns self
- next()- Returns the next value or throws StopIteration

See this example in Python to get a basic understanding how to implement this pattern by yourself:

```
class MyFib(object):
    def __init__(self):
        self.i = 2
    def __iter__(self):
        return self
    def next(self):
        if self.i > 1000:
            raise StopIteration
        self.i = self.i * self.i
        return self.i

>>> print [x for x in MyFib()]
[4, 16, 256, 65536]
```

But even the following examples use the built-in iterator pattern in Python:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

[19] For a more explicit example with a Abstract Factory see http://en.wikipedia.org/wiki/Abstract_factory_pattern

5.2.2 Observer

This pattern defines one-to-many dependencies between objects so that when one object changes state, all its dependences are notified and updated automatically. You have two classes, the subject and an observer which registers itself with the subject and receives notification callbacks when data changes in the subject. The GOF form of this pattern is somehow limiting, because they describe a system where a subject class has a general 'notify' method used for everything (although they do suggest a mechanism for generating more selective events using aspects). This example is more general and self explaining. It makes heavy use of first class functions and closures:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def scale(self, n):
        self.x = n * self.x
        self.y = n * self.y

def notify(f):
    def g(self, n):
        print "Action performed:", f, "
            Logged:", n
        return f(self, n)
    return g

>>> Point.scale = notify(Point.scale)
>>> p = Point(2.0, 3.0)
>>> p.scale(2.5)
Action performed: <unbound method Point.
    scale> Logged: 2.5
```

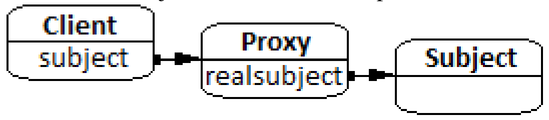
In Java you can use *Observable* and *Observer* which will handle the nasty implementation in the background for you.[17]

5.3 Structural Patterns

Structural patterns describe how classes and objects can be combined to form larger structures. The difference between class patterns and object patterns is that the former describe how inheritance can be used to provide more useful program interfaces. Object patterns, on the other hand, describe how objects can be composed into larger structures using object composition, or the inclusion of objects within other ones.

5.3.1 Proxy

From the point of view of the object-oriented design the ability to trap attribute access provides some new solutions to old problems. A good example is the Proxy pattern, which is used when an object has to be shielded from its clients. There may be a number of reasons for this: reference counting, different levels of access rights, lazy evaluation of the state of the object and so on. The runtime object structure for the pattern looks like this:[11]



The client does not need to know that it is not accessing the real object (*Subject*) directly. In other words the proxy substitutes for the real thing. In C++ this would mean that both the proxy and the subject class have a common base class. In Python the same effect can be achieved by masquerading the proxy instance as the subject by providing the same method interface.

The Proxy class is based on the near-universal Wrapper class. As such the Proxy class is only a trivial wrapper for any subject. However, its main function is to serve as a base class for more specific proxy implementations. Because the *__gettattr__* method is called only as the last resort the Proxy can handle a subject method differently by "overloading" it. Let us consider the following situation: we access an instance of class *RGB* first directly, then using a generic Proxy instance as a wrapper and finally via *NoBlueProxy* instance.

This is the way our Proxy structure should work:

```
>>> rgb = RGB( 100, 192, 240 )
>>> rgb.Red()
100
>>> proxy = Proxy( rgb )
>>> proxy.Green()
192
>>> noblue = NoBlueProxy( rgb )
>>> noblue.Green()
192
>>> noblue.Blue()
0
```

The near universal wrapper class which was mentioned already:[15]

```
class Proxy:
    def __init__( self, subject ):
        self.__subject = subject
    def __getattr__( self, name ):
        return getattr( self.__subject, name
            )
```

The Subject class:

```
class RGB:
        def __init__( self, red, green, blue
            ):
        self.__red = red
        self.__green = green
        self.__blue = blue
    def Red( self ):
        return self.__red
    def Green( self ):
        return self.__green
    def Blue( self ):
        return self.__blue
```

And the (very simple) more specific Proxy implementation:

```
class NoBlueProxy( Proxy ):
    def Blue( self ):
        return 0
```

While Python provides some mechanisms that are usually implemented with the Proxy pattern (for example simple reference counting) the pattern is still highly applicable to the Python community.[20]

5.3.2 Adapter

The Adapter pattern (also known as the Wrapper pattern or simply a Wrapper) translates one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original one. The adapter translates calls to its interface into calls to the original interface, and the

[20] Example is Daniel Larsson's RemoteCall package, which provides a simple mechanism for distributed applications.

amount of code necessary to do this is typically small. The adapter is also responsible for transforming data into appropriate forms. There are two types of Adapter pattern:

- Object Adapter pattern: In this type, the adapter contains an instance of the class it wraps. In this situation, the adapter makes calls to the instance of the wrapped object.
- Class Adapter pattern: This uses multiple inheritance to achieve its goal. The adapter is created inheriting from both the interface that is expected and the interface that is pre-existing. It is typical for the expected interface to be created as a pure interface class, especially in languages such as Java that do not support multiple inheritance.

Consider now this (very simple) example for an Object Adapter:

```
class Adaptee:
    def specific_request(self):
        return 'Adaptee'

class Adapter:
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return self.adaptee.specific_request
            ()

client = Adapter(Adaptee())
>>> print client.request()
Adaptee
```

In Java it would look like this:

```
public class Adaptee {
        public String specificRequest() {
                return "Adaptee";
        }
}

public class Adapter {
        private Adaptee adaptee;

        public Adapter(Adaptee adaptee) {
                this.adaptee = adaptee;
        }

        public String request() {
                return this.adaptee.
                    specificRequest();
        }
}
...
Adapter client = new Adapter(new Adaptee());
System.out.println(client.request());
```

Basically Python and Java are very similar at this point. But now lets take a look at the following example of a Class Adapter:

```
class Adaptee1:
    def __init__(self):
        pass

    def specific_request(self):
        return 'Adaptee1'

class Adaptee2:
    def __init__(self):
        pass

    def specific_request(self):
        return 'Adaptee2'

class Adapter(Adaptee1, Adaptee2):
    def __init__(self):
        Adaptee1.__init__(self)
        Adaptee2.__init__(self)

    def request(self):
        return Adaptee1.specific_request(
            self), Adaptee2.specific_request
            (self)

>>> client = Adapter()
>>> print client.request()
('Adaptee1', 'Adaptee2')
```

With Java you have a problem right now. A Class Adapter is only usable if you have a language that supports all needed features such as multiple inheritance and private inheritance (e.g. C++). Java does not have them at all. It is possible to try to fix it with a combination of class inheritance and interface implementation. But doing so will make all methods of the adapter class accessible for all clients. This is basically usable as an adapter but it is no Adapter pattern concerning the meaning of the GOF book which wants to transfer one interface into another.

Conclusion

At the end a word of warning: [14] [16] *"Small Boy with a Patterns Book. After spending a bunch of time thinking about these ideas, over a few days now, I finally recognized in myself what I call "Small Boy with a Patterns Book". You can always tell when someone on your team is reading the Gang of Four book (Gamma, et al., Design Patterns). Every day or so, this person comes in with a great idea for a place in the system that is just crying out for the use of Composite, or whatever chapter he read last night. There's an old saying: To a small boy with a hammer, everything looks like a nail. As programmers, we call into the same trap all too often. We learn about some new technology or solution, and we immediately begin seeing places to apply it."*

Design Patterns are very useful tools which can be addictive. They give you a language for thinking about the design and allow you to recognise familiar problems in new contexts but **do not overuse them**! Recognising a pattern lets you immediately relate that pattern back to previous experiences both good and bad. Some patterns are almost universal across programming languages but other patterns are specific to the features or even the syntax of a particular language. To become fluent in a programming language means not just understanding the syntax but also adopting a common pattern of thought with the other developers. Even within a language possible implementations of patterns change as the language evolves. In conclusion Python is not Java without compilation. It is a rich language with lots of features that obviate the need for many patterns. You need to ask yourself, does Python let me do this better with e.g. First Class Functions, First Class Classes or Closures? These features reduce/remove patterns, and thus shorten code. There are still patterns, and where those patterns exist, there is a ripe place for a new language feature in the future.

Acknowledgments

I would like to thank Felix Schmitt, Martin Morgenstern, Franz Gregor and Florian Heidenreich for reviewing. Also special thanks for their interest and many useful comments which helped to improve the paper.

Additional material

Here you can find all code that was used for the performance measurements. *Test environment:* Intel Core 2 Solo ULV @ 1.4GHz, 2GiB RAM, Windows 7 x64 and Ubuntu 10.10 x64 with stock kernel 2.6.35, Python 2.7. All tests ran each 100 times at Windows and Ubuntu, results are average values calculated from both operating systems. To minimize side effects the tests were divided into one that uses a language extension and one that does not.

This is the time measuring function which was used in all tests:

```
import time

def measure(callable):
    t1 = time.time()
    callable()
    t2 = time.time()
    return t2 - t1
```

Code for testing a lambda function:

```
''' 10000 up to 1000000 '''
RANGE = 10000

f = lambda x : x + x

def do_the_trick_lamda():
    for i in range(1,RANGE):
        f(i)

if __name__ == '__main__':
    sum = 0
    count = 100

    for i in range(1,count):
        sum += measure(do_the_trick_lamda)

    print "Time used at average with lambda:
        ", sum / count, "ms"
```

The same without using a lambda function:

```
''' 10000 up to 1000000 '''
RANGE = 10000

def double_manually(x):
    return x + x

def do_the_trick_manually():
    for i in range(1,RANGE):
        double_manually(i)

if __name__ == '__main__':
    sum = 0
    count = 100
    for i in range(1,count):
        sum += measure(do_the_trick_manually
            )

    print "Time used at average without
        lambda: ", sum / count, "ms"
```

Code for testing list comprehensions:

```
''' 10000 up to 1000000 '''
lst = range(1,10000)

def with_list_comp():
    return [x*2 for x in lst if x%2 == 0]

if __name__ == '__main__':
    sum = 0
    count = 100

    for i in range(1,count):
        sum += measure(with_list_comp)

    print "Time used at average with list
        comp: ", sum / count, "ms"
```

The same without using a list comprehensions:

```
''' 10000 up to 1000000 '''
lst = range(1,10000)

def without_list_comp():
    retlist = list()
    for i in lst:
        if i%2 == 0:
            retlist.append(i*2)
    return retlist

if __name__ == '__main__':
    sum = 0
    count = 100
    for i in range(1,count):
        sum += measure(without_list_comp)

    print "Time used at average without list
         comp: ", sum / count, "ms"
```

Code for testing a closure:

```
''' 10000 up to 1000000 '''
lst = range(1, 10000)

def with_closures():
    def outer(outer_argument):
        def inner(inner_argument):
            return outer_argument +
                inner_argument
        return inner

    g = outer(1)
    for i in lst:
        g(i)

if __name__ == '__main__':
    sum = 0
    count = 100

    for i in range(1,count):
        sum += measure(with_closures)

    print "Time used at average with closure
        : ", sum / count, "ms"
```

The same without using a closure:

```
''' 10000 up to 1000000 '''
lst = range(1, 1000000)

def without_closures():
    x = 1
    sum = 0;
    for i in lst:
        sum = x + i

if __name__ == '__main__':
    sum = 0
    count = 100
    for i in range(1,count):
        sum += measure(without_closures)

    print "Time used at average without
        closure: ", sum / count, "ms"
```

References

[1] Voidspace. Introduction to OOP with Python. URL *http://www.voidspace.org.uk/python/articles/OOP.shtml*, 10/26/2010 - 11:40.

[2] http://docs.python.org Python v2.7 documentation for classes URL *http://docs.python.org/tutorial/classes.html*, 10/26/2010 - 13:30.

[3] http://docs.python.org Python v2.7 documentation for functions URL *http://docs.python.org/library/functions.html*, 10/26/2010 - 13:30.

[4] http://diveintopython.org Dive Into Python - Lambda Functions URL *http://diveintopython.org/power_of_introspection/lambda_functions.html*, 10/30/2010 - 22:30.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, 1996. 457 pages.

[6] Paul Dubois. *Introduction to Python*, Tutorial at TOOLS USA 96, August 1996

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995. 395 pages.

[8] Vespe Savikko. *Design Patterns in Python*, Tampere University of Technology URL *http://www.python.org/workshops/1997-10/proceedings/savikko.html*

[9] Alex Martelli (aleax@google.com). *Design Patterns in Python*, Google URL *http://www.aleax.it/gdd_pydp.pdf*

[10] Joe Gregorio (Google). *(The Lack of) Design Patterns in Python*, OSCON 5:20pm Thursday, 07/24/2008 Portland 256 URL *http://assets.en.oreilly.com/1/event/12/_The%20Lack%20of_%20Design %20Patterns%20in%20Python%20Presentation.pdf*

[11] http://docs.python.org Design Patterns in Python URL *http://www.python.org/workshops/1997-10/proceedings/savikko.html*, 11/12/2010 - 20:00.

[12] Alex Martelli, David Ascher. *Python Cookbook*, O'Reilly Media, 2002. 608 pages.

[13] Joshua Bloch.*Effective Java*, Addison-Wesley Longman, Amsterdam; Auflage: 2nd Revised edition (REV). 384 Seiten.

[14] www.suttoncourtenay.org.uk Patterns in Python URL *http://www.suttoncourtenay.org.uk/duncan/accu/pythonpatterns.html*, 11/14/2010 - 11:40.

[15] Guido van Rossum. *Python Tutorial*, URL http://www.python.org October 1996.

[16] Ron Jeffries. *Adventures in C#: Some Things We Ought to Do*, Jan 2003 URL http://www.xprogramming.com/xpmag/acsMusings.htm

[17] Oracle. *Java API Doc for 1.6*, URL http://download.oracle.com/javase/6/docs/api/index.html