CH 9: Hidden Features of Python.

**9.1 python “With” Statement**

**9.1.1. A popular Patten in python program: Setting up and Tear Down:**

A commnd pattern in Python programming is like setting up the tools, doing some operations with the tools and destroy the tools. Consider this piece of code:

set things up

try:

do something

finally:

tear things down

Here, “set things up” could be opening a file, or acquiring some sort of external resource, and “tear things down” would then be closing the file, or releasing or removing the resource.

The **try-finally** construct guarantees that the “tear things down” part is always executed, even if the code that does the work doesn’t finish.

If you do this a lot, it would be quite convenient if you could put the “set things up” and “tear things down” code in a library function, to make it easy to reuse. You can of course do something like

def controlled\_execution(callback):

set things up

try:

callback(thing)

finally:

tear things down

def my\_function(thing):

do something

controlled\_execution(my\_function)

But that’s a bit verbose, especially if you need to modify local variables. Another approach is to use a one-shot generator, and use the **for-in** statement to “wrap” the code:

def controlled\_execution():

set things up

try:

yield thing

finally:

tear things down

for thing in controlled\_execution():

do something with thing

But **yield** isn’t even allowed inside a **try-finally** in 2.4 and earlier. And while that could be fixed (and it has been fixed in 2.5), it’s still a bit weird to use a loop construct when you know that you only want to execute something once.

**9.1.1. Example of Setting up and Tear Down in Python and how it can be done by with**

Python’s with statement provides a very convenient way of dealing with the situation where you have to do a setup and teardown to make something happen. A very good example for this is the situation where you want to gain a handler to a file, read data from the file and the close the file handler.

Without the with statement, one would write something along the lines of:

file = open("/tmp/foo.txt")

data = file.read()

file.close()

There are two annoying things here.

1. First, you end up forgetting to close the file handler.
2. The second is how to handle exceptions that may occur once the file handler has been obtained. One could write something like this to get around this:

file = open("/tmp/foo.txt") <- Here file is not initialize if open failed.

try:

data = file.read()

finally:

file.close()

While this works well, it is unnecessarily verbose. This is where with is useful. The good thing about with apart from the better syntax is that it is very good handling exceptions. The above code would look like this, when using with:

with open("/tmp/foo.txt") as file: # Will take care of start and distroy

data = file.read()

Let consider another example, as below:

Common application functionality is to access a database, add some records and finally commit those updates as a transaction. Using SQLAlchemy it would look something like this:

from sqlalchemy import \*

db = create\_engine('sqlite:///test.db') # Seeting up

dbcon = db.connect() # Setting up

transaction = dbcon.begin() # Body

dbcon.execute('insert into NameDB (FirstName, LastName,Age) values ("john","smith",34)')

dbcon.execute('insert into NameDB (FirstName, LastName,Age) values ("dave","smith",37)')

transaction.commit() #Tear Down and closing db

You can see that here the resource in question is the database transaction; we set it up before the inserts and close it after the commit. However because the SQLAlchemy transaction object implements a context manager the above code could be written more succinctly as:

from sqlalchemy import \*

db = create\_engine('sqlite:///test.db')

dbcon = db.connect()

with dbcon.begin():

dbcon.execute('insert into NameDB (FirstName, LastName,Age) values ("john","smith",34)')

dbcon.execute('insert into NameDB (FirstName, LastName,Age) values ("dave","smith",37)')

**9.1.1. Way context manager?**

There are a few Python library classes which provide context information that is used by the **with** statement. The most commonly-used class is the [**file**](http://www.itmaybeahack.com/book/python-2.6/html/p02/p02c09_files.html#file) class.

There are two forms of the **with** statement. In the first, the context object does not provide a context-specific object to work with. In the second, the context provides us an object to be used within the context.

with context :

Operation with this context

with context as variable :

operation use the context variable.

Let’s go back to the **file handler example,** When we open a file for processing, we are creating a context. When we leave that context, we want to be sure that the file is properly closed. Here’s the standard example of how this is used.

with file('someData.txt','r') as theFile: #open by the file handaler

for aLine in theFile:

print aLine

# the file was closed by the context manager

Let’s see how the context manager works in step by step.

1. We create the file, which can be used as a context manager. The with statement enters the context, which returns a file object that we can use for input and output purposes. The as clause specifies that the working object is assigned to theFile.
2. This is a pretty typical for statement that reads each line of a file.
3. The with statement also exits the context, irrespective of the presence or absence of exceptions. In the case of a [file](http://www.itmaybeahack.com/book/python-2.6/html/p02/p02c09_files.html#file) context manager, this will close the file.

**9.1.1. Defining a Context Manager Function in a class**

In programming, we also want to make out own context handler, for that we need to define the task which should be executed when a context is created or destroyed. Python provide a way to do this, explained as below:

class controlled\_execution:

def \_\_enter\_\_(self):

set things up

return thing

def \_\_exit\_\_(self, type, value, traceback): # It takes 4 argumnets

tear things down

with controlled\_execution() as thing:

some code

Now, when the “with” statement is executed, Python evaluates the expression, calls the \_\_enter\_\_ method on the resulting value (which is called a “context guard”), and assigns whatever \_\_enter\_\_ returns to the variable given by as. Python will then execute the code body, and no matter what happens in that code, call the guard object’s \_\_exit\_\_ method.

As an extra bonus, the \_\_exit\_\_ method can look at the exception, if any, and suppress it or act on it as necessary. To suppress the exception, just return a true value. For example, the following \_\_exit\_\_ method swallows any TypeError, but lets all other exceptions through:

def \_\_exit\_\_(self, type, value, traceback):

return isinstance(value, TypeError)

In Python 2.5, the file object has been equipped with **\_\_enter\_\_** and **\_\_exit\_\_** methods; the former simply returns the file object itself, and the latter closes the file:

>>> f = open("x.txt")

>>> f

<open file 'x.txt', mode 'r' at 0x00AE82F0>

>>> f.\_\_enter\_\_()

<open file 'x.txt', mode 'r' at 0x00AE82F0>

>>> f.read(1)

'X'

>>> f.\_\_exit\_\_(None, None, None)

>>> f.read(1)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: I/O operation on closed file

so to open a file, process its contents, and make sure to close it, you can simply do:

with open("x.txt") as f:

data = f.read()

do something with data

Another example, shows how \_\_enter\_\_() and exit() function works, illustrated as below: This can be demonstrated with the following example:

class Sample:

def \_\_enter\_\_(self):

print "In \_\_enter\_\_()"

return "Foo"

def \_\_exit\_\_(self, type, value, trace):

print "In \_\_exit\_\_()"

def get\_sample():

return Sample()

with get\_sample() as sample:

print "sample:", sample

**Output :**

In \_\_enter\_\_()

sample: Foo

In \_\_exit\_\_()

The Steps are as below:

1. The \_\_enter\_\_() function is executed
2. The value returned by it – in this case "Foo" is assigned to sample
3. The body of the block is executed, thereby printing the value ofsample ie. "Foo"
4. The \_\_exit\_\_() function is called.

Now let’s see how \_\_exit\_\_() works, more closely.:

#!/usr/bin/env python

# with\_example02.py

class Sample:

def \_\_enter\_\_(self):

return self

def \_\_exit\_\_(self, type, value, trace):

print "type:", type

print "value:", value

print "trace:", trace

def do\_something(self):

bar = 1/0

return bar + 10

with Sample() as sample:

sample.do\_something()

Output :

type: <type 'exceptions.ZeroDivisionError'>

value: integer division or modulo by zero

trace: <traceback object at 0x1004a8128>

Traceback (most recent call last):

File "./with\_example02.py", line 19, in <module>

sample.do\_something()

File "./with\_example02.py", line 15, in do\_something

bar = 1/0

ZeroDivisionError: integer division or modulo by zero

**9.1.1. Multiple context managers with “with”**

In the previous example, we see that we create a single context using single context manger, but it is possible to get multiple context with “with” as below:

with open("file1") as f1, open("file2") as f2:

#do stuff

With earlier Python versions it is thought that one is required to write:

with open("file1") as f1:

with open("file2") as f2:

#do stuff

Let’s create our own multi context class, as below:

class multicontext(object):

def \_\_init\_\_(self, \*args):

if (len(args) == 1 and

(hasattr(args[0], "\_\_len\_\_") or

hasattr(args[0], "\_\_iter\_\_"))):

self.objs = list(args[0])

else:

self.objs = args

def \_\_enter\_\_(self):

return tuple(obj.\_\_enter\_\_() for obj in self.objs)

def \_\_exit\_\_(self, type\_, value, traceback):

return all([obj.\_\_exit\_\_(type\_,value, traceback)

for obj in self.objs])

**Output:**

=================================================================

>>> with multicontext(open("%s.txt" %letter, "wt") for letter in "abc") as (a,b,c):

... a.write("a")

... b.write("b")

... c.write("c")

...

>>> a

<closed file 'a.txt', mode 'wt' at 0x7f4172f17ad0>

>>> b

<closed file 'b.txt', mode 'wt' at 0x7f4172f177a0>

>>> c

<closed file 'c.txt', mode 'wt' at 0x7f4172f179c0>

>>>

**9.2: Python Decorators**

**9.2.1 Make your Life Easy**

### 1. Intro.

Decorators are a useful addition to the Python programming language, but can be difficult to grasp at first. If you already read some articles on Python decorators and found them hard to understand, this tutorial, aimed at beginners, will attempt at explaining how you can use them in a simple and, hopefully, comprehensible way.

### 2. So what are decorators?

Put simply, “decorating” a function means embedding it inside another “decorator” function or class.  
If you had a previous exposure to Django, the popular Web application framework written in Python, you’ll probably have encountered the ‘@require\_login decorator’. Its purpose is quite simple: any ‘view’ method decorated with it will be accessible only to authenticated users.

@require\_login

def my\_index\_page():

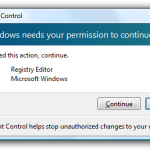
html = "<p>Welcome to my boring webpage, user!</p>"

return html

Basically, this code means: “the user will be required to login before he or she can access the page content.”  
In this tutorial, we will be writing a very similiar decorator function.

### 3. Our first decorator.

If you ever used Windows 7 or Vista you probably encountered a security mechanism called ‘UAC’. It attempts to protect the operating system from malware by requiring the user to explicitly ‘accept’ or ‘deny’ the execution of any program that requires  
super-user privileges.



“Windows needs your permission to continue. Continue or Cancel?”.  
We will replicate this functionality with a decorator function called ”authorize”, that will require the user to confirm the execution of any function decorated with it.

### 4. Let’s code it!

First of all, let’s define a very simple function:

def say\_hello():

print "Hello world!"

The say\_hello() command should be fairly self explanatory!  
We will now port the UAC mechanism to Python, call it ‘authorize()’, and decorate our function with it. Let’s fire up the Python interpreter and enter the following code:

>>> def authorize(command):

>>>     if raw\_input('>>> Do you wish to execute this command? ') == "y":

>>>         return command()

It’s a very simple function: a command will be passed to our decorator function as a value and a confirmation input will be requested from the user; if the user enters the letter ‘y’ as a confirmation, the command will be executed.  
Let’s decorate our say\_hello() function and see if it works…

>>> @authorize

>>> def say\_hello():

>>>     print "Hello world!"

The output will be:

[Do you wish to execute this command?] y

Hello world!

Congratulations! You’ve just written your first decorator.

ecorators. The shear mention of them brings fear to even the seasoned Python programmer.

Okay, maybe not. But decorators, at least in this author's opinion, have a weird syntax and inevitably complicated implementations that are especially foreign (I think) to many who have gotten used to the simplicity of Python.

## [1   The Basics](http://www.siafoo.net/article/68#id3)

Decorators modify functions. More specifically, a decorator is a function **that transforms another function**.

When you use a decorator, Python passes the decorated function -- we'll call this the target function -- to thedecorator function, and replaces it with the result. Without decorators, it would look something like this:

[# 's](http://www.siafoo.net/article/68)

1**def** decorator\_function(target):  
 2 *# Do something with the target function*  
 3 target.attribute = 1  
 4 **return** target  
 5  
 6**def** target(a,b):  
 7 **return** a + b  
 8  
 9*# This is what the decorator actually does*  
10target = decorator\_function(target)

This code has the exact same functionality, but uses decorators. Note that I can name my decorator function whatever I want. Here, I've chosen 'decorator\_function':

[# 's](http://www.siafoo.net/article/68)

1**def** decorator\_function(target):  
2 *# Do something with the target function*  
3 target.attribute = 1  
4 **return** target  
5  
6*# Here is the decorator, with the syntax '@function\_name'*  
7@decorator\_function  
8**def** target(a,b):  
9 **return** a + b

As you can see, you need to put the decorator function's name, prefaced with a @, on the line before the target function definition. Python internally will transform the target by applying the decorator to it and replacing it with the returned value.

Both of the above examples will have the same results:

[# 's](http://www.siafoo.net/article/68)

1**>>>** target(1,2)  
23  
3**>>>** target.attribute  
41

### [1.1   Does a decorator function have to return a function?](http://www.siafoo.net/article/68#id4)

No. The decorator function can return absolutely **anything**, and Python will replace the target function with that return value. For example, you could do something like this:

[# 's](http://www.siafoo.net/article/68)

1**def** decorator\_evil(target):  
 2 **return** False  
 3  
 4@decorator\_evil  
 5**def** target(a,b):  
 6 **return** a + b  
 7  
 8>>> target  
 9False  
10  
11>>> target(1,2)  
12**TypeError**: 'bool' object **is** **not** callable

This is really not something you want to be doing on a regular basis though -- I'm pretty sure that a basic design principal is to not have functions randomly turning into other sorts of things. It makes good sense to at least return some sort of callable.

## [2   Run-Time Transformations](http://www.siafoo.net/article/68#id5)

"But," I hear you saying, "I thought decorators did more than that. I want to do things at run-time, like conditionally calling the function and transforming the arguments and return value."

Can decorators do these things? Yes. Is that really something 'more' than we talked about above? Not really. It's important here not to get bogged down in the details -- you already know all there is to know about decorators. To do one of these more complex things, we're really just adding some plain old Python to the mix.

### [2.1   The Wrapper Function](http://www.siafoo.net/article/68#id6)

Remember, your decorator function can return an arbitrary function. We'll call it the wrapper function, for reasons which will become clear in a second. The trick here is to define the wrapper function inside the decorator function, giving it access to the decorator function's variable scope, including the target function.

[# 's](http://www.siafoo.net/article/68)

1**def** decorator(target):  
 2  
 3 **def** wrapper():  
 4 **print** 'Calling function "**%s**"' % target.\_\_name\_\_  
 5 **return** target()  
 6  
 7 *# Since the wrapper is replacing the target function, assigning an attribute to the target function won't do anything.*  
 8 *# We need to assign it to the \*wrapper function\*.*  
 9 wrapper.attribute = 1  
10 **return** wrapper  
11  
12@decorator  
13**def** target():  
14 **print** 'I am the target function'  
15  
16>>> target()  
17Calling function "target"  
18I am the target function  
19  
20>>> target.attribute  
211

As you can see, the **wrapper function** can do whatever it wants to the target function, including the simple case of returning the target's return value. But what happens to any arguments passed to the target function?

### [2.2   Getting the Arguments](http://www.siafoo.net/article/68#id7)

Since the returned wrapper function replaces the target function, the wrapper function will **receive the arguments** intended for the target function. Assuming you want your decorator to work for any target function, your wrapper function then should accept [arbitrary non-keyword arguments and arbitrary keyword arguments](http://www.siafoo.net/article/52#arbitrary-numbers-of-arguments), add, remove, or modify arguments if necessary, and [pass the arguments](http://www.siafoo.net/article/52#passing-a-list-or-dictionary-as-arguments) to the target function.

[# 's](http://www.siafoo.net/article/68)

1**def** decorator(target):  
 2  
 3 **def** wrapper(\*args, \*\*kwargs):  
 4 kwargs.update({'debug': True}) *# Edit the keyword arguments -- here, enable debug mode no matter what*  
 5 **print** 'Calling function "**%s**" with arguments **%s** and keyword arguments **%s**' % (target.\_\_name\_\_, args, kwargs)  
 6 **return** target(\*args, \*\*kwargs)  
 7  
 8 wrapper.attribute = 1  
 9 **return** wrapper  
10  
11@decorator  
12**def** target(a, b, debug=False):  
13 **if** debug: **print** '[Debug] I am the target function'  
14 **return** a+b  
15  
16>>> target(1,2)  
17Calling function "target" **with** arguments (1, 2) **and** keyword arguments {'debug': True}  
18[Debug] I am the target function  
193  
20  
21>>> target.attribute  
221

**Note**

You can also apply a decorator to a **class method**. If your decorator is always going to be used this way, and you need access to the current instance, your wrapper function can assume the first argument is always self:

[# 's](http://www.siafoo.net/article/68)

1**def** wrapper(self, \*args, \*\*kwargs):  
2 *# Do something with 'self'*  
3 **print** self  
4 **return** target(self, \*args, \*\*kwargs)

### [2.3   Summing It Up](http://www.siafoo.net/article/68#id8)

So, we have a wrapper function that accepts arbitrary arguments defined inside our decorator function. The wrapper function can call the target function if and when it wants, get the result, do something with it, and return whatever it wants.

Say I want certain function calls to require positive confirmation before they are executed, and then stringify the result of the function before returning it. Note that the built-in function raw\_input prints a message and then waits for a response from stdin.

[# 's](http://www.siafoo.net/article/68)

1**def** decorator(target): *# Python passes the target function to the decorator*  
 2  
 3 **def** wrapper(\*args, \*\*kwargs):  
 4  
 5 choice = raw\_input('Are you sure you want to call the function "**%s**"? ' % target.\_\_name\_\_)  
 6  
 7 **if** choice **and** choice[0].lower() == 'y':  
 8 *# If input starts with a 'y', call the function with the arguments*  
 9 result = target(\*args, \*\*kwargs)  
10 **return** str(result)  
11  
12 **else**:  
13 **print** 'Call to **%s** cancelled' % target.\_\_name\_\_  
14  
15 **return** wrapper  
16  
17@decorator  
18**def** target(a,b):  
19 **return** a+b  
20  
21>>> test.target(1,2)  
22Are you sure you want to call the function "target"? n  
23Call to target cancelled  
24  
25>>> test.target(1,2)  
26Are you sure you want to call the function "target"? y  
273

## [3   Dynamic Decorators](http://www.siafoo.net/article/68#id9)

Sometimes you might want to customize behavior by passing arbitrary options to your decorator function. A cursory look at decorator syntax suggests there's no way to do that. You could just abandon the decorator idea altogether, but you certainly don't have to.

The solution is define your decorator function inside another function -- call it the options function. Right before the target function definition, where you would normally list the decorator function (prepended with an @), call this options function(prepended with an @) instead. The options function then returns your decorator function, which Python will use as the passes the target function to as before.

### [3.1   Passing Options to the Decorator](http://www.siafoo.net/article/68#id10)

Your options function can accept any arguments you want it to. Since the decorator function is defined inside the options function, the decorator function has access to any of the arguments passed to the options function.

[# 's](http://www.siafoo.net/article/68)

1**def** options(value):  
 2  
 3 **def** decorator(target):  
 4 *# Do something with the target function*  
 5 target.attribute = value  
 6 **return** target  
 7 **return** decorator  
 8  
 9@options('value')  
10**def** target(a,b):  
11 **return** a + b  
12  
13>>> target(1,2)  
143  
15  
16>>> target.attribute  
17'value'

As you can see, nothing here about the decorator syntax itself has changed. Our decorator function is just in a dynamic scope instead of a static one.

### [3.2   Run-Time Tranformations](http://www.siafoo.net/article/68#id11)

You can do [Run-time Transformations](http://www.siafoo.net/article/68#run-time-transformations) by returning a wrapper function from your decorator function, just like before. For better or worse, though, there now must be three levels of functions:

[# 's](http://www.siafoo.net/article/68)

1**def** options(debug\_level):  
 2  
 3 **def** decorator(target):  
 4  
 5 **def** wrapper(\*args, \*\*kwargs):  
 6 kwargs.update({'debug\_level': debug\_level}) *# Edit the keyword arguments*  
 7 *# here, set debug level to whatever specified in the options*  
 8  
 9 **print** 'Calling function "**%s**" with arguments **%s** and keyword arguments **%s**' % (target.\_\_name\_\_, args, kwargs)  
10 **return** target(\*args, \*\*kwargs)  
11  
12 **return** wrapper  
13  
14 **return** decorator  
15  
16@options(5)  
17**def** target(a, b, debug\_level=0):  
18 **if** debug\_level: **print** '[Debug Level **%s**] I am the target function' % debug\_level  
19 **return** a+b  
20  
21>>> target(1,2)  
22Calling function "target" **with** arguments (1, 2) **and** keyword arguments {'debug\_level': 5}  
23[Debug Level 5] I am the target function  
243

## [4   Caveat: Function Signatures](http://www.siafoo.net/article/68#id12)

Phew. Understand everything you can do with decorators now? Good :). However, there is one drawback that must be mentioned.

The function returned from the decorator function -- usually a wrapper function -- replaces the target function completely. Any later introspection into what appears to be the target function will actually be into the wrapper function.

**Most of the time, this is okay**. Generally you just call a function with some options. Your program doesn't check to see what the function's \_\_name\_\_ or what arguments it accepts. So usually this problem won't be a problem.

However sometimes you care if the function you are calling supports a certain option, supports arbitrary options, or, perhaps, what its \_\_name\_\_ is. Or maybe you are interested in one if the function's attributes. If you look at a function that has been decorated, you will actually be looking at the wrapper function.

In the example below, note that the getargspec function of the [inspect](http://docs.python.org/lib/inspect-classes-functions.html) module gets the names and default values of a function's arguments.

[# 's](http://www.siafoo.net/article/68)

1*# This function is the same as the function 'target', except for the name*  
 2**def** standalone\_function(a,b):  
 3 **return** a+b  
 4  
 5**def** decorator(target):  
 6  
 7 **def** wrapper(\*args, \*\*kwargs):  
 8 **return** target()  
 9  
10 **return** wrapper  
11  
12@decorator  
13**def** target(a,b):  
14 **return** a+b  
15  
16>>> from **inspect** import getargspec  
17  
18>>> standalone\_function.\_\_name\_\_  
19'standalone\_function'  
20  
21>>> getargspec(standalone\_function)  
22(['a', 'b'], None, None, None)  
23  
24>>> target.\_\_name\_\_  
25'wrapper'  
26  
27>>> getargspec(target)  
28([], 'args', 'kwargs', None)

As you can see, the wrapper function reports that it accepts different arguments than the original target function Its call signature has changed.

### [4.1   A Solution](http://www.siafoo.net/article/68#id13)

This is not an easy problem to solve. The update\_wrapper method of the [functools module](http://docs.python.org/lib/module-functools.html) provides a partial solution, copying the \_\_name\_\_ and other attributes from one function to another. But it does not solve what might be the largest problem of all: the changed call signature.

The decorator function of the [decorator module](http://www.phyast.pitt.edu/~micheles/python/documentation.html) provides the best solution: it can wrap your wrapper function in a dynamically-evaluated function with the correct arguments, restoring the original call signature. Similar to the update\_wrapperfunction, it can also update your wrapper function with the \_\_name\_\_ and other attributes from the target function.

**Note**

For the remainder of this section, when I speak of the decorator function, I mean the one from this module, not one of the decorator functions that we've been using to transform our target functions.

**Another way to create decorators**

Unfortunately, though, the decorator function wasn't written with this use in mind. Instead it was written to turn standalone wrapper functions into full-fledged decorators, without having to worry about the function nesting described in [Run-Time Transformations](http://www.siafoo.net/article/68#run-time-transformations), above.

While this technique is often useful, it is much less customizable. **Everything must be done at run-time**, each time the function is executed. You cannot do **any** work when the target function is defined, including assigning the targetor wrapper functions attributes or [passing options to the decorator](http://www.siafoo.net/article/68#passing-options-to-the-decorator).

Also, in this author's opinion it is a bit of a black box; I'd rather know what my decorators are doing even if it is a little messier.

But we can make it work for us to solve this problem.

Ideally you would just call decorator(wrapper) and be done with it. However, things are never as simple as we'd like. As described above, the decorator function wraps the function passed to it -- our wrapper function -- in a dynamic function to fix the signature. But we still have a few problems:

Problem #1:

The dynamic function calls our wrapper function with (func, \*args, \*\*kwargs)

Solution #1:

Make our wrapper function accept (func, \*args, \*\*kwargs) instead of just (\*args, \*\*kwargs).

Problem #2:

The dynamic function is then wrapped in another function that expects to be used as an actual decorator -- it expects to be called with the target function, and will return the wrapper function.

Solution #2:

Call decorator's return value with the target function to get back to the dynamic function, which has the right signature.

This technique is a bit of a hack, and is a bit hard to explain, but it is easy to implement and works well.

This is the same example as before, but now with the decorator function (and a name change so things don't get too confusing):

[# 's](http://www.siafoo.net/article/68)

1from **decorator** import decorator  
 2  
 3**def** my\_decorator(target):  
 4  
 5 **def** wrapper(target, \*args, \*\*kwargs): *# the target function has been prepended to the list of arguments*  
 6 **return** target(\*args, \*\*kwargs)  
 7  
 8 *# We are calling the returned value with the target function to get a 'proper' wrapper function back*  
 9 **return** decorator(wrapper)(target)  
10  
11  
12@my\_decorator  
13**def** target(a,b):  
14 **return** a+b  
15  
16>>> from **inspect** import getargspec  
17  
18>>> target.\_\_name\_\_  
19'target'  
20  
21>>> getargspec(target)  
22(['a', 'b'], None, None, None)

## [5   Putting it All Together](http://www.siafoo.net/article/68#id14)

Sometimes, you really need a [customizable decorator](http://www.siafoo.net/article/68#dynamic-decorators) that does work both at [parse-time](http://www.siafoo.net/article/68#the-basics) and [run-time](http://www.siafoo.net/article/68#run-time-transformations), and has the [signature](http://www.siafoo.net/article/68#caveat-function-signatures) of the original target function.

Here's an example that ties everything together. Expanding on the example from earlier, say you want certain function calls to require positive confirmation before they are executed, and you want to be able to customize the confirmation string for each target function. Furthermore, for some reason [[1]](http://www.siafoo.net/article/68#id2), you need the decorated function's signature to match the target function.

Here we go:

[# 's](http://www.siafoo.net/article/68)

1from **decorator** import decorator  
 2  
 3*# The 'options' function. Recieves options and returns a decorator.*  
 4**def** confirm(text):  
 5 *'''*  
 6 *Pass a string to be sent as a confirmation message. Returns a decorator.*  
 7 *'''*  
 8  
 9 *# The actual decorator. Recieves the target function.*  
10 **def** my\_decorator(target):  
11 *# Anything not in the wrapper function is done when the target function is initially parsed*  
12  
13 *# This is okay because the decorator function will copy the attribute to the wrapper function*  
14 target.attribute = 1  
15  
16 *# The wrapper function. Replaces the target function and receives its arguments*  
17 **def** wrapper(target, \*args, \*\*kwargs):  
18 *# You could do something with the args or kwargs here*  
19  
20 choice = raw\_input(text)  
21  
22 **if** choice **and** choice[0].lower() == 'y':  
23 *# If input starts with a 'y', call the function with the arguments*  
24 result = target(\*args, \*\*kwargs)  
25 *# You could do something with the result here*  
26 **return** result  
27  
28 **else**:  
29 **print** 'Call to **%s** cancelled' % target.\_\_name\_\_  
30  
31 *# Fix the wrapper's call signature*  
32 **return** decorator(wrapper)(target)  
33  
34 **return** my\_decorator  
35  
36@confirm('Are you sure you want to add these numbers? ')  
37**def** target(a,b):  
38 **return** a+b  
39  
40>>> Are you sure you want to add these numbers? yes  
413  
42  
43>>> target.attribute  
441

Hey, what do you know, it actually works.

## [6   Conclusion](http://www.siafoo.net/article/68#id15)

As always, if you have a better way to do anything mentioned here, or if I've left anything out, leave a comment or feel free edit this article to fix the problem.

**9.2.2 Functions**

**Functions**

When the Python interpreter encounters this code:

def hello():

print ("Hello, world!")

it:

* compiles the code to create a function object
* binds the name “hello” to that function object.

Then, to run the function object, you can code

hello()

which causes this to be printed:

Hello, world!

If you code:

print (hello)

you will get something like:

<function hello at 0x02D021E0>

which is the string representation of the *hello* function object.

**9.2.3 Macro and It’s Importnt**

Indeed, you can use Python decorators to implement the \*Decorator\* pattern, but

that's an extremely limited use of it. Python decorators, I think, are best

equated to macros.

History of Macros

==========================

The macro has a long history, but most people will probably have had experience

with C preprocessor macros. The problems with C macros were (1) they were in a

different language (not C) and (2) the behavior was sometimes bizarre, and often

inconsistent with the behavior of the rest of C.

Both Java and C# have added \*annotations\*, which allow you to do some things to

elements of the language. Both of these have the problems that (1) to do what

you want, you sometimes have to jump through some enormous and untenable hoops,

which follows from (2) these annotation features have their hands tied by the

bondage-and-discipline (or as `Martin Fowler gently puts it: "Directing"

<http://martinfowler.com/bliki/SoftwareDevelopmentAttitude.html>`\_) nature of

those languages.

In a slightly different vein, many C++ programmers (myself included) have noted

the generative abilities of C++ templates and have used that feature in a macro-

like fashion.

Many other languages have incorporated macros, but without knowing much about it

I will go out on a limb and say that Python decorators are similar to Lisp

macros in power and possibility.

The Goal of Macros

============================

I think it's safe to say that the goal of macros in a language is to provide a

way to modify elements of the language. That's what decorators do in Python --

they modify functions, and in the case of \*class decorators\*, entire classes.

This is why they usually provide a simpler alternative to metaclasses.

The major failings of most language's self-modification approaches are that they

are too restrictive and that they require a different language (I'm going to say

that Java annotations with all the hoops you must jump through to produce an

interesting annotation comprises a "different language").

Python falls into Fowler's category of "enabling" languages, so if you want to

do modifications, why create a different or restricted language? Why not just

use Python itself? And that's what Python decorators do.

**What we can do using decorator?**

What Can You Do With Decorators?

===================================

Decorators allow you to inject or modify code in functions or classes. Sounds a

bit like \*Aspect-Oriented Programming\* (AOP) in Java, doesn't it? Except that

it's both much simpler and (as a result) much more powerful. For example,

suppose you'd like to do something at the entry and exit points of a function

(such as perform some kind of security, tracing, locking, etc. -- all the

standard arguments for AOP). With decorators, it looks like this::

@entryExit

def func1():

print("inside func1()")

@entryExit

def func2():

print("inside func2()")

The ``@`` indicates the application of the decorator.

**Function Decorator:**

Function Decorators

==============================

A function decorator is applied to a function definition by placing it on the

line before that function definition begins. For example::

@myDecorator

def aFunction():

print("inside aFunction")

When the compiler passes over this code, ``aFunction()`` is compiled and the

resulting function object is passed to the ``myDecorator`` code, which does

something to produce a function-like object that is then substituted for the

original ``aFunction()``.

What does the ``myDecorator`` code look like? Well, most introductory examples

show this as a function, but I've found that it's easier to start understanding

decorators by using classes as decoration mechanisms instead of functions. In

addition, it's more powerful.

**Decorator as a Class**

The only constraint upon the object returned by the decorator is that it can be

used as a function -- which basically means it must be callable. Thus, any

classes we use as decorators must implement ``\_\_call\_\_``.

What should the decorator do? Well, it can do anything but usually you expect

the original function code to be used at some point. This is not required,

however::

# PythonDecorators/my\_decorator.py

class my\_decorator(object):

def \_\_init\_\_(self, f):

print("inside my\_decorator.\_\_init\_\_()")

f() # Prove that function definition has completed

def \_\_call\_\_(self):

print("inside my\_decorator.\_\_call\_\_()")

@my\_decorator

def aFunction():

print("inside aFunction()")

print("Finished decorating aFunction()")

aFunction()

When you run this code, you see::

inside my\_decorator.\_\_init\_\_()

inside aFunction()

Finished decorating aFunction()

inside my\_decorator.\_\_call\_\_()

Notice that the constructor for ``my\_decorator`` is executed at the point of

decoration of the function. Since we can call ``f()`` inside ``\_\_init\_\_()``, it

shows that the creation of ``f()`` is complete before the decorator is called.

Note also that the decorator constructor receives the function object being

decorated. Typically, you'll capture the function object in the constructor and

later use it in the ``\_\_call\_\_()`` method (the fact that decoration and calling

are two clear phases when using classes is why I argue that it's easier and more

powerful this way).

When ``aFunction()`` is called after it has been decorated, we get completely

different behavior; the ``my\_decorator.\_\_call\_\_()`` method is called instead of

the original code. That's because the act of decoration \*replaces\* the original

function object with the result of the decoration -- in our case, the

``my\_decorator`` object replaces ``aFunction``. Indeed, before decorators were

added you had to do something much less elegant to achieve the same thing::

def foo(): pass

foo = staticmethod(foo)

With the addition of the ``@`` decoration operator, you now get the same result

by saying::

@staticmethod

def foo(): pass

This is the reason why people argued against decorators, because the ``@`` is

just a little syntax sugar meaning "pass a function object through another

function and assign the result to the original function."

The reason I think decorators will have such a big impact is because this little

bit of syntax sugar changes the way you think about programming. Indeed, it

brings the idea of "applying code to other code" (i.e.: macros) into mainstream

thinking by formalizing it as a language construct.

Slightly More Useful

========================

Now let's go back and implement the first example. Here, we'll do the more

typical thing and actually use the code in the decorated functions::

# PythonDecorators/entry\_exit\_class.py

class entry\_exit(object):

def \_\_init\_\_(self, f):

self.f = f

def \_\_call\_\_(self):

print("Entering", self.f.\_\_name\_\_)

self.f()

print("Exited", self.f.\_\_name\_\_)

@entry\_exit

def func1():

print("inside func1()")

@entry\_exit

def func2():

print("inside func2()")

func1()

func2()

The output is::

Entering func1

inside func1()

Exited func1

Entering func2

inside func2()

Exited func2

You can see that the decorated functions now have the "Entering" and "Exited"

trace statements around the call.

The constructor stores the argument, which is the function object. In the call,

we use the ``\_\_name\_\_`` attribute of the function to display that function's

name, then call the function itself.

**Inner function as Decoratror**

Using Functions as Decorators

=====================================

The only constraint on the result of a decorator is that it be callable, so it

can properly replace the decorated function. In the above examples, I've

replaced the original function with an object of a class that has a

``\_\_call\_\_()`` method. But a function object is also callable, so we can rewrite

the previous example using a function instead of a class, like this::

# PythonDecorators/entry\_exit\_function.py

def entry\_exit(f):

def new\_f():

print("Entering", f.\_\_name\_\_)

f()

print("Exited", f.\_\_name\_\_)

return new\_f

@entry\_exit

def func1():

print("inside func1()")

@entry\_exit

def func2():

print("inside func2()")

func1()

func2()

print(func1.\_\_name\_\_)

``new\_f()`` is defined within the body of ``entry\_exit()``, so it is created and

returned when ``entry\_exit()`` is called. Note that ``new\_f()`` is a \*closure\*,

because it captures the actual value of ``f``.

Once ``new\_f()`` has been defined, it is returned from ``entry\_exit()`` so that

the decorator mechanism can assign the result as the decorated function.

The output of the line ``print(func1.\_\_name\_\_)`` is ``new\_f``, because the

``new\_f`` function has been substituted for the original function during

decoration. If this is a problem you can change the name of the decorator

function before you return it::

def entry\_exit(f):

def new\_f():

print("Entering", f.\_\_name\_\_)

f()

print("Exited", f.\_\_name\_\_)

new\_f.\_\_name\_\_ = f.\_\_name\_\_

return new\_f

The information you can dynamically get about functions, and the modifications

you can make to those functions, are quite powerful in Python.

**Decorator without Arguments**

Review: Decorators without Arguments

=========================================

If we create a decorator without arguments, the function to be decorated is

passed to the constructor, and the ``\_\_call\_\_()`` method is called whenever the

decorated function is invoked::

# PythonDecorators/decorator\_without\_arguments.py

class decorator\_without\_arguments(object):

def \_\_init\_\_(self, f):

"""

If there are no decorator arguments, the function

to be decorated is passed to the constructor.

"""

print("Inside \_\_init\_\_()")

self.f = f

def \_\_call\_\_(self, \*args):

"""

The \_\_call\_\_ method is not called until the

decorated function is called.

"""

print("Inside \_\_call\_\_()")

self.f(\*args)

print("After self.f(\*args)")

@decorator\_without\_arguments

def sayHello(a1, a2, a3, a4):

print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")

sayHello("say", "hello", "argument", "list")

print("After first sayHello() call")

sayHello("a", "different", "set of", "arguments")

print("After second sayHello() call")

Any arguments for the decorated function are just passed to ``\_\_call\_\_()``. The

output is::

Inside \_\_init\_\_()

After decoration

Preparing to call sayHello()

Inside \_\_call\_\_()

sayHello arguments: say hello argument list

After self.f(\*args)

After first sayHello() call

Inside \_\_call\_\_()

sayHello arguments: a different set of arguments

After self.f(\*args)

After second sayHello() call

Notice that ``\_\_init\_\_()`` is the only method called to perform decoration, and

``\_\_call\_\_()`` is called every time you call the decorated ``sayHello()``.

**Decorator with Argumnets.**

Decorators with Arguments

====================================

The decorator mechanism behaves quite differently when you pass arguments to the

decorator.

Let's modify the above example to see what happens when we add arguments to the

decorator::

# PythonDecorators/decorator\_with\_arguments.py

class decorator\_with\_arguments(object):

def \_\_init\_\_(self, arg1, arg2, arg3):

"""

If there are decorator arguments, the function

to be decorated is not passed to the constructor!

"""

print("Inside \_\_init\_\_()")

self.arg1 = arg1

self.arg2 = arg2

self.arg3 = arg3

def \_\_call\_\_(self, f):

"""

If there are decorator arguments, \_\_call\_\_() is only called

once, as part of the decoration process! You can only give

it a single argument, which is the function object.

"""

print("Inside \_\_call\_\_()")

def wrapped\_f(\*args):

print("Inside wrapped\_f()")

print("Decorator arguments:", self.arg1, self.arg2, self.arg3)

f(\*args)

print("After f(\*args)")

return wrapped\_f

@decorator\_with\_arguments("hello", "world", 42)

def sayHello(a1, a2, a3, a4):

print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")

sayHello("say", "hello", "argument", "list")

print("after first sayHello() call")

sayHello("a", "different", "set of", "arguments")

print("after second sayHello() call")

From the output, we can see that the behavior changes quite significantly::

Inside \_\_init\_\_()

Inside \_\_call\_\_()

After decoration

Preparing to call sayHello()

Inside wrapped\_f()

Decorator arguments: hello world 42

sayHello arguments: say hello argument list

After f(\*args)

after first sayHello() call

Inside wrapped\_f()

Decorator arguments: hello world 42

sayHello arguments: a different set of arguments

After f(\*args)

after second sayHello() call

Now the process of decoration calls the constructor and then immediately invokes

``\_\_call\_\_()``, which can only take a single argument (the function object) and

must return the decorated function object that replaces the original. Notice

that ``\_\_call\_\_()`` is now only invoked once, during decoration, and after that

the decorated function that you return from ``\_\_call\_\_()`` is used for the

actual calls.

Although this behavior makes sense -- the constructor is now used to capture the

decorator arguments, but the object ``\_\_call\_\_()`` can no longer be used as the

decorated function call, so you must instead use ``\_\_call\_\_()`` to perform the

decoration -- it is nonetheless surprising the first time you see it because

it's acting so much differently than the no-argument case, and you must code the

decorator very differently from the no-argument case.

**Decorator with decorated argunets**

Decorator Functions with Decorator Arguments

==================================================

Finally, let's look at the more complex decorator function implementation, where

you have to do everything all at once::

# PythonDecorators/decorator\_function\_with\_arguments.py

def decorator\_function\_with\_arguments(arg1, arg2, arg3):

def wrap(f):

print("Inside wrap()")

def wrapped\_f(\*args):

print("Inside wrapped\_f()")

print("Decorator arguments:", arg1, arg2, arg3)

f(\*args)

print("After f(\*args)")

return wrapped\_f

return wrap

@decorator\_function\_with\_arguments("hello", "world", 42)

def sayHello(a1, a2, a3, a4):

print('sayHello arguments:', a1, a2, a3, a4)

print("After decoration")

print("Preparing to call sayHello()")

sayHello("say", "hello", "argument", "list")

print("after first sayHello() call")

sayHello("a", "different", "set of", "arguments")

print("after second sayHello() call")

Here's the output::

Inside wrap()

After decoration

Preparing to call sayHello()

Inside wrapped\_f()

Decorator arguments: hello world 42

sayHello arguments: say hello argument list

After f(\*args)

after first sayHello() call

Inside wrapped\_f()

Decorator arguments: hello world 42

sayHello arguments: a different set of arguments

After f(\*args)

after second sayHello() call

The return value of the decorator function must be a function used to wrap the

function to be decorated. That is, Python will take the returned function and

call it at decoration time, passing the function to be decorated. That's why we

have three levels of functions; the inner one is the actual replacement

function.

Because of closures, ``wrapped\_f()`` has access to the decorator arguments

``arg1``, ``arg2`` and ``arg3``, \*without\* having to explicitly store them as in

the class version. However, this is a case where I find "explicit is better than

implicit," so even though the function version is more succinct I find the class

version easier to understand and thus to modify and maintain.

**Step by step as decorator:**

# ython Decorators

Posted on [**2012/04/29**](http://pythonconquerstheuniverse.wordpress.com/2012/04/29/python-decorators/)

In August 2009, I wrote a post titled [Introduction to Python Decorators](http://pythonconquerstheuniverse.wordpress.com/2009/08/06/introduction-to-python-decorators-part-1). It was an attempt to explain Python decorators in a way that I (and I hoped, others) could grok.

Recently I had occasion to re-read that post. It wasn’t a pleasant experience — it was pretty clear to me that the attempt had failed.

That failure — and two other things — have prompted me to try again.

* Matt Harrison has published an excellent e-book [Guide to: Learning Python Decorators](http://www.amazon.com/Guide-Learning-Python-Decorators-ebook/dp/B006ZHJSIM/).
* I now have a theory about why most explanations of decorators (mine included) fail, and some ideas about how better to structure an introduction to decorators.

There is an old saying to the effect that “Every stick has two ends, one by which it may be picked up, and one by which it may not.” I believe that most explanations of decorators fail because they pick up the stick by the wrong end.

In this post I will show you what the wrong end of the stick looks like, and point out why I think it is wrong. And I will show you what I think the right end of the stick looks like.

# The wrong way to explain decorators

Most explanations of Python decorators start with an example of a function to be decorated, like this:

|  |  |
| --- | --- |
| 1  2 | def aFunction():      print("inside aFunction") |

and then add a decoration line, which starts with an @ sign:

|  |  |
| --- | --- |
| 1  2  3 | @myDecorator  def aFunction():      print("inside aFunction") |

At this point, the author of the introduction often defines a decorator as the line of code that begins with the “@”. (In my older post, I called such lines “annotation” lines. I now prefer the term “decoration” line.)

For instance, in 2008 Bruce Eckel [wrote on his Artima blog](http://www.artima.com/weblogs/viewpost.jsp?thread=240808)

A function decorator is applied to a function definition by placing it on the line before that function definition begins.

and in 2004, Phillip Eby wrote in [an article in Dr. Dobb’s Journal](http://www.drdobbs.com/web-development/184406073)

Decorators may appear before any function definition…. You can even stack multiple decorators on the same function definition, one per line.

Now there are two things wrong with this approach to explaining decorators. The first is that the explanation begins in the wrong place. It starts with an example of a function to be decorated and an decoration line, when it should begin with the decorator itself. The explanation should end, not start, with the decorated function and the decoration line. The decoration line is, after all, merely syntactic sugar — is not at all an essential element in the concept of a decorator.

The second is that the term “decorator” is used incorrectly (or ambiguously) to refer both to the decorator and to the decoration line. For example, in his Dr. Dobb’s Journalarticle, after using the term “decorator” to refer to the decoration line, Phillip Eby goes on to define a “decorator” as a callable object.

But before you can do that, you first need to have some decorators to stack. A decorator is a callable object (like a function) that accepts one argument—the function being decorated.

So… it would seem that a decorator is both a callable object (like a function) **and** a single line of code that can appear before the line of code that begins a function definition. This is sort of like saying that an “address” is both a building (or apartment) at a specific location**and** a set of lines (written in pencil or ink) on the front of a mailing envelope. The ambiguity may be almost invisible to someone familiar with decorators, but it is very confusing for a reader who is trying to learn about decorators from the ground up.

# The right way to explain decorators

So how **should** we explain decorators?

Well, we start with the decorator, not the function to be decorated.

**One**  
We start with the [basic notion of a function](http://www.informit.com/articles/article.aspx?p=1849243) — a function is something that generates a value based on the values of its arguments.

**Two**  
We note that in Python, functions are first-class objects, so they can be passed around like other values (strings, integers, objects, etc.).

**Three**  
We note that because functions are first-class objects in Python, we can write functions that both (a) accept function objects as argument values, and (b) return function objects as return values. For example, here is a function foobarthat accepts a function objectoriginal\_functionas an argument and returns a function object new\_functionas a result.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def foobar(original\_function):        # make a new function      def new\_function():          # some code        return new\_function |

**Four**  
We define “decorator”.

*A****decorator****is a function (such as*foobar*in the above example) that takes a function object as an argument, and returns a function object as a return value.*

So there we have it — the definition of a decorator. Anything else that we say about decorators is a refinement of, or an expansion of, or an addition to, this definition of a decorator.

**Five**  
We show what the internals of a decorator look like. Specifically, we show different ways that a decorator can use the original\_function in the creation of the new\_function. Here is a simple example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def verbose(original\_function):        # make a new function that prints a message when original\_function starts and finishes      def new\_function(\*args, \*\*kwargs):          print("Entering", original\_function.\_\_name\_\_)          original\_function(\*args, \*\*kwargs)          print("Exiting ", original\_function.\_\_name\_\_)        return new\_function |

**Six**  
We show how to invoke a decorator — how we can pass into a decorator one function object (its input) and get back from it a different function object (its output). In the following example, we pass the widget\_func function object to the verbose decorator, and we get back a new function object to which we assign the name talkative\_widget\_func.

|  |  |
| --- | --- |
| 1  2  3  4 | def widget\_func():      # some code    talkative\_widget\_func = verbose(widget\_func) |

**Seven**  
We point out that decorators are often used to add features to the original\_function. Or more precisely, decorators are often used to create a new\_function that does roughly whatoriginal\_function does, but also does things in addition to what original\_function does.

And we note that the output of a decorator is typically used to replace the original functionthat we passed in to the decorator as an argument. A typical use of decorators looks like this. (Note the change to line 4 from the previous example.)

|  |  |
| --- | --- |
| 1  2  3  4 | def widget\_func():      # some code    widget\_func = verbose(widget\_func) |

So for all practical purposes, in a typical use of a decorator we pass a function (widget\_func) through a decorator (verbose) and get back an enhanced (or souped-up, or “decorated”) version of the function.

**Eight**  
We introduce Python’s “decoration syntax” that uses the “@” to create decoration lines. This feature is basically syntactic sugar that makes it possible to re-write our last example this way:

|  |  |
| --- | --- |
| 1  2  3 | @verbose  def widget\_func():      # some code |

The result of this example is exactly the same as the previous example — after it executes, we have a widget\_func that has all of the functionality of the original widget\_func, plus the functionality that was added by the verbose decorator.

Note that in **this** way of explaining decorators, the “@” and decoration syntax is one of the **last** things that we introduce, not one of the first.

And we absolutely do **not** refer to line 1 as a “decorator”. We might refer to line 1 as, say, a “decorator invocation line” or a “decoration line” or simply a “decoration”… whatever. But line 1 is **not** a “decorator”.

Line 1 is a line of code. A decorator is a function — a different animal altogether.

**Nine**  
Once we’ve nailed down these basics, there are a few advanced features to be covered.

* We explain that a decorator need not be a function (it can be any sort of callable, e.g. a class).
* We explain how decorators can be nested within other decorators.
* We explain how  decoration lines can be “stacked”. A better way to put it would be: we explain how decorators can be “chained”.
* We explain how additional arguments can be passed to decorators, and how decorators can use them.

**Ten — A decorators cookbook**

The material that we’ve covered up to this point is what any basic introduction to Python decorators would cover. But a Python programmer needs something more in order to be productive with decorators. He (or she) needs a catalog of recipes, patterns, examples, and commentary that describes / shows / explains when and how decorators can be used to accomplish specific tasks. (Ideally, such a catalog would also include examples and warnings about decorator gotchas and anti-patterns.) Such a catalog might be called “Python Decorator Cookbook” or perhaps “Python Decorator Patterns”.

* As far as I know, no such decorator cookbook currently exists.
* The [Python Decorator Library](http://wiki.python.org/moin/PythonDecoratorLibrary) on the Python wiki is a collection of decorator examples. It has its uses, but it does not have the systematic organization and explanatory material of a true cookbook.
* Something similar to a descriptor cookbook, although still not systematically organized, can be generated by a [search of the ActiveState Python Cookbook, filtering on “descriptor”.](http://code.activestate.com/search/recipes/#q=decorator)

So that’s it. I’ve described what I think is wrong (well, let’s say suboptimal) about most introductions to decorators. And I’ve sketched out what I think is a better way to structure an introduction to decorators.

Now I can explain why I like Matt Harrison’s e-book [Guide to: Learning Python Decorators](http://www.amazon.com/Guide-Learning-Python-Decorators-ebook/dp/B006ZHJSIM/). Matt’s introduction is structured in the way that I think an introduction to decorators shouldbe structured. It picks up the stick by the proper end.

The first two-thirds of the Guide hardly talk about decorators at all. Instead, Matt begins with a thorough discussion of how Python functions work. By the time the discussion gets to decorators, we have been given a strong understanding of the internal mechanics of functions. And since most decorators are functions (remember our definition of decorator), at that point it is relatively easy for Matt to explain the internal mechanics of decorators.

Which is just as it should be.

**9.2.10 : a chain of function decorators in Python**

def makebold(fn):

def wrapped():

return "<b>" + fn() + "</b>"

return wrapped

def makeitalic(fn):

def wrapped():

return "<i>" + fn() + "</i>"

return wrapped

@makebold

@makeitalic

def hello():

return "hello world"

print hello() ## returns <b><i>hello world</i></b>

def \_\_exit\_\_(self, type, value, traceback):

return isinstance(value, TypeError)

## In Python 2.5, the file object has been equipped with \_\_enter\_\_ and \_\_exit\_\_ methods; the former simply returns the file object itself, and the latter closes the

## 9.3 Python generator, Yield, Co-routines and Sub-routines

## How normal function works in Python? When we call a normal Python function, execution starts at function's first line and continues until a return statement, exception, or the end of the function (which is seen as an implicit return None) is encountered. Once a function returns control to its caller, that's it. Any work done by the function and stored in local variables is lost. A new call to the function creates everything from scratch.

## We want something different then how normal function works. This is all very standard when discussing functions (more generally referred to as [subroutines](http://en.wikipedia.org/wiki/Subroutine)) in computer programming. There are times, though, when it's beneficial to have the ability to create a "function" which, instead of simply returning a single value, is able to yield a series of values. To do so, such a function would need to be able to "save its work," so to speak. I said, "yield a series of values" because our hypothetical function doesn't "return" in the normal sense. Return implies that the function is *returning control of execution* to the point where the function was called. "Yield," however, implies that *the transfer of control is temporary and voluntary*, and our function expects to regain it in the future. In Python, "functions" with these capabilities are called generators, and they're incredibly useful.

To better understand the problem **generators** solve, let's take a look at an example. Throughout the example, keep in mind the core problem being solved: **generating a series of values.**

Suppose our boss asks us to write a function that takes a list of ints and returns some Iterable containing the elements which are prime[1](http://www.jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/#fn:prime) numbers. *Remember, an* [Iterable](http://docs.python.org/3/glossary.html#term-iterable) *is just an object capable of returning its members one at a time.*

"Simple," we say, and we write the following:

def get\_primes(input\_list):

result\_list = list()

for element in input\_list:

if is\_prime(element):

result\_list.append()

return result\_list

# This is a better way.

def get\_primes(input\_list):

return (element for element in input\_list if is\_prime(element))

#Helper Functions

def is\_prime(number):

if number > 1:

if number == 2:

return True

if number % 2 == 0:

return False

for current in range(3, int(math.sqrt(number) + 1), 2):

if number % current == 0:

return False

return True

return False

Either get\_primes implementation above fulfills the requirements, so we tell our boss we're done. She reports our function works and is exactly what she wanted.

A few days later, our boss comes back and tells us she's run into a small problem: she wants to use our get\_primes function on a very large list of numbers. In fact, the list is so large that merely creating it would consume all of the system's memory. To work around this, she wants to be able to call get\_primes with a start value and get all the primes larger than start.

Once we think about this new requirement, it becomes clear that it requires more than a simple change to get\_primes. Clearly, we can't return a list of all the prime numbers from start to infinity *(operating on infinite sequences, though, has a wide range of useful applications)*. The chances of solving this problem using a normal function seem bleak. **Now is the question is how to return a infinite sequences?**

Before we give up, let's determine the core obstacle preventing us from writing a function that satisfies our boss's new requirements. Thinking about it, we arrive at the following: **functions only get one chance to return results, and thus must return finite results at once.** It seems pointless to make such an obvious statement; "functions just work that way," we think. The real value lies in asking, "but what if they didn't?"

Imagine what we could do if get\_primes could simply return the *next* value instead of all the values at once. It wouldn't need to create a list at all. No list, no memory issues. Since our boss told us she's just iterating over the results, she wouldn't know the difference. Unfortunately, this doesn't seem possible. Even if we had a magical function that allowed us to iterate from n to infinity, we'd get stuck after returning the first value:

def get\_primes(start):

for element in magical\_infinite\_range(start):

if is\_prime(element):

return element # return only Once and Game is Over

**Enter the Generator?** This sort of problem is so common that a new construct was added to Python to solve it: the generator. A generator "generates" values. Creating generators was made as straightforward as possible through the concept of generator functions, introduced simultaneously.

A **generator function** is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function (even if it also contains a return statement). There's nothing else we need to do to create one.

Generator functions create generator iterators. That's the last time you'll see the term generator iterator, though, since they're almost always referred to as "generators". Just remember that a generator is a special type of iterator. To be considered an iterator, generators must define a few methods, one of which is\_\_next\_\_(). To get the next value from a generator, we use the same built-in function as for iterators: next().

This point bear repeating: **to get the next value from a** generator**, we use the same built-in function as for** iterators**:** next().

So whenever next() is called on a generator, the generator is responsible for passing back a value to whomever called next(). It does so by calling yield along with the value to be passed back (e.g. yield 7). The easiest way to remember what yield does is to think of it as return (plus a little magic) for generator functions.

Again, this bears repeating: yield **is just** return **(plus a little magic) for** generator functions**.**

Here's a simple generator function:

>>> def simple\_generator\_function():

>>> yield 1

>>> yield 2

>>> yield 3

**And here are two simple ways to use it:**

>>> for value in simple\_generator\_function():

>>> print(value)

1

2

3

>>> our\_generator = simple\_generator\_function()

>>> next(our\_generator)

1

>>> next(our\_generator)

2

>>> next(our\_generator)

3

What's the magic part? Glad you asked! When a generator function calls yield, the "state" of the generator function is frozen; the values of all variables are saved and the next line of code to be executed is recorded until next() is called again. Once it is, the generator function simply resumes where it left off. If next() is never called again, the state recorded during the yield call is (eventually) discarded.

Let's rewrite get\_primes as a generator function. Notice that we no longer need the magical\_infinite\_range function. Using a simple while loop, we can create our own infinite sequence:

def get\_primes(number):

while True:

if is\_prime(number):

yield number

number += 1

If a generator function calls return or reaches the end its definition, a StopIteration exception is raised.

## 5.1. Iterators

We use for statement for looping over a list.

>>> **for** i **in** **[**1**,** 2**,** 3**,** 4**]:**

... **print** i**,**

...

1

2

3

4

If we use it with a string, it loops over its characters.

>>> **for** c **in** "python"**:**

... **print** c

...

p

y

t

h

o

n

If we use it with a dictionary, it loops over its keys.

>>> **for** k **in** **{**"x"**:** 1**,** "y"**:** 2**}:**

... **print** k

...

y

x

If we use it with a file, it loops over lines of the file.

>>> **for** line **in** open**(**"a.txt"**):**

... **print** line**,**

...

first line

second line

So there are many types of objects which can be used with a for loop. These are called iterable objects.

There are many functions which consume these iterables.

>>> ",".join**([**"a"**,** "b"**,** "c"**])**

'a,b,c'

>>> ",".join**({**"x"**:** 1**,** "y"**:** 2**})**

'y,x'

>>> list**(**"python"**)**

['p', 'y', 't', 'h', 'o', 'n']

>>> list**({**"x"**:** 1**,** "y"**:** 2**})**

['y', 'x']

### 5.1.1. The Iteraton Protocol

The built-in function iter takes an iterable object and returns an iterator.

>>> x = iter**([**1**,** 2**,** 3**])**

>>> x

<listiterator object at 0x1004ca850>

>>> x.next**()**

1

>>> x.next**()**

2

>>> x.next**()**

3

>>> x.next**()**

**Traceback (most recent call last):**

File "<stdin>", line 1, in <module>

StopIteration

Each time we call the next method on the iterator gives us the next element. If there are no more elements, it raises a StopIteration.

Iterators are implemented as classes. Here is an iterator that works like built-in xrangefunction.

**class** yrange**:**

**def** \_\_init\_\_**(**self**,** n**):**

self.i = 0

self.n = n

**def** \_\_iter\_\_**(**self**):**

**return** self

**def** next**(**self**):**

**if** self.i < self.n**:**

i = self.i

self.i += 1

**return** i

**else:**

**raise** **StopIteration()**

The \_\_iter\_\_ method is what makes an object iterable. Behind the scenes, the iterfunction calls \_\_iter\_\_ method on the given object.

The return value of \_\_iter\_\_ is an iterator. It should have a next method and raiseStopIteration when there are no more elements.

Lets try it out:

>>> y = yrange**(**3**)**

>>> y.next**()**

0

>>> y.next**()**

1

>>> y.next**()**

2

>>> y.next**()**

**Traceback (most recent call last):**

File "<stdin>", line 1, in <module>

File "<stdin>", line 14, in next

StopIteration

Many built-in functions accept iterators as arguments.

>>> list**(**yrange**(**5**))**

[0, 1, 2, 3, 4]

>>> sum**(**yrange**(**5**))**

10

In the above case, both the iterable and iterator are the same object. Notice that the\_\_iter\_\_ method returned self. It need not be the case always.

**class** zrange**:**

**def** \_\_init\_\_**(**self**,** n**):**

self.n = n

**def** \_\_iter\_\_**(**self**):**

**return** zrange\_iter**(**self.n**)**

**class** zrange\_iter**:**

**def** \_\_init\_\_**(**self**,** n**):**

self.i = 0

self.n = n

**def** \_\_iter\_\_**(**self**):**

*# Iterators are iterables too.*

*# Adding this functions to make them so.*

**return** self

**def** next**(**self**):**

**if** self.i < self.n**:**

i = self.i

self.i += 1

**return** i

**else:**

**raise** **StopIteration()**

If both iteratable and iterator are the same object, it is consumed in a single iteration.

>>> y = yrange**(**5**)**

>>> list**(**y**)**

[0, 1, 2, 3, 4]

>>> list**(**y**)**

[]

>>> z = zrange**(**5**)**

>>> list**(**z**)**

[0, 1, 2, 3, 4]

>>> list**(**z**)**

[0, 1, 2, 3, 4]

**Problem 1:**Write an iterator class reverse\_iter, that takes a list and iterates it from the reverse direction. ::

>>> it = reverse\_iter**([**1**,** 2**,** 3**,** 4**])**

>>> it.next**()**

4

>>> it.next**()**

3

>>> it.next**()**

2

>>> it.next**()**

1

>>> it.next**()**

**Traceback (most recent call last):**

File "<stdin>", line 1, in <module>

StopIteration

## 5.2. Generators

Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.

**def** yrange**(**n**):**

i = 0

**while** i < n**:**

**yield** i

i += 1

Each time the yield statement is executed the function generates a new value.

>>> y = yrange**(**3**)**

>>> y

<generator object yrange at 0x401f30>

>>> y.next**()**

0

>>> y.next**()**

1

>>> y.next**()**

2

>>> y.next**()**

**Traceback (most recent call last):**

File "<stdin>", line 1, in <module>

StopIteration

So a generator is also an iterator. You don’t have to worry about the iterator protocol.

The word “generator” is confusingly used to mean both the function that generates and what it generates. In this chapter, I’ll use the word “generator” to mean the genearted object and “generator function” to mean the function that generates it.

Can you think about how it is working internally?

When a generator function is called, it returns an generator object without even beginning execution of the function. When next` method is called for the first time, the function starts executing until it reaches yield statement. The yielded value is returned by the next call.

The following example demonstrates the interplay between yield and call to nextmethod on generator object.

>>> **def** foo**():**

... **print** "begin"

... **for** i **in** range**(**3**):**

... **print** "before yield"**,** i

... **yield** i

... **print** "after yield"**,** i

... **print** "end"

...

>>> f = foo**()**

>>> f.next**()**

begin

before yield 0

0

>>> f.next**()**

after yield 0

before yield 1

1

>>> f.next**()**

after yield 1

before yield 2

2

>>> f.next**()**

after yield 2

end

**Traceback (most recent call last):**

File "<stdin>", line 1, in <module>

StopIteration

>>>

Lets see an example:

**def** integers**():**

*"""Infinite sequence of integers."""*

i = 1

**while** True**:**

**yield** i

i = i + 1

**def** squares**():**

**for** i **in** integers**():**

**yield** i \* i

**def** take**(**n**,** seq**):**

*"""Returns first n values from the given sequence."""*

seq = iter**(**seq**)**

result = **[]**

**try:**

**for** i **in** range**(**n**):**

result.append**(**seq.next**())**

**except** **StopIteration:**

**pass**

**return** result

**print** take**(**5**,** squares**())** *# prints [1, 4, 9, 16, 25]*

## 5.3. Generator Expressions

Generator Expressions are generator version of list comprehensions. They look like list comprehensions, but returns a generator back instead of a list.

>>> a = **(**x\*x **for** x **in** range**(**10**))**

>>> a

<generator object <genexpr> at 0x401f08>

>>> sum**(**a**)**

285

We can use the generator expressions as arguments to various functions that consume iterators.

>>> sum**(((**x\*x **for** x **in** range**(**10**)))**

285

When there is only one argument to the calling function, the parenthesis around generator expression can be omitted.

>>> sum**(**x\*x **for** x **in** range**(**10**))**

285

Another fun example:

Lets say we want to find first 10 (or any n) pythogorian triplets. A triplet (x, y, z) is called pythogorian triplet if x\*x + y\*y == z\*z.

It is easy to solve this problem if we know till what value of z to test for. But we want to find first n pythogorian triplets.

>>> pyt = **((**x**,** y**,** z**)** **for** z **in** integers**()** **for** y **in** xrange**(**1**,** z**)** **for** x **in** range**(**1**,** y**)** **if** x\*x + y\*y == z\*z**)**

>>> take**(**10**,** pyt**)**

[(3, 4, 5), (6, 8, 10), (5, 12, 13), (9, 12, 15), (8, 15, 17), (12, 16, 20), (15, 20, 25), (7, 24, 25), (10, 24, 26), (20, 21, 29)]

### 5.3.1. Example: Reading multiple files

Lets say we want to write a program that takes a list of filenames as arguments and prints contents of all those files, like cat command in unix.

The traditional way to implement it is:

**def** cat**(**filenames**):**

**for** f **in** filenames**:**

**for** line **in** open**(**f**):**

**print** line**,**

Now, lets say we want to print only the line which has a particular substring, like grepcommand in unix.

**def** grep**(**pattern**,** filenames**):**

**for** f **in** filenames**:**

**for** line **in** open**(**f**):**

**if** pattern **in** line**:**

**print** line**,**

Both these programs have lot of code in common. It is hard to move the common part to a function. But with generators makes it possible to do it.

**def** readfiles**(**filenames**):**

**for** f **in** filenames**:**

**for** line **in** open**(**f**):**

**yield** line

**def** grep**(**pattern**,** lines**):**

**return** **(**line **for** line **in** lines **if** pattern **in** lines**)**

**def** printlines**(**lines**):**

**for** line **in** lines**:**

**print** line**,**

**def** main**(**pattern**,** filenames**):**

lines = readfiles**(**filenames**)**

lines = grep**(**pattern**,** lines**)**

printlines**(**lines**)**

The code is much simpler now with each function doing one small thing. We can move all these functions into a separate module and reuse it in other programs.

**Problem 2:**Write a program that takes one or more filenames as arguments and prints all the lines which are longer than 40 characters.

**Problem 3:**Write a function findfiles that recursively descends the directory tree for the specified directory and generates paths of all the files in the tree.

**Problem 4:**Write a function to compute the number of python files (.py extension) in a specified directory recursively.

**Problem 5:**Write a function to compute the total number of lines of code in all python files in the specified directory recursively.

**Problem 6:**Write a function to compute the total number of lines of code, ignoring empty and comment lines, in all python files in the specified directory recursively.

**Problem 7:**Write a program split.py, that takes an integer n and a filename as command line arguments and splits the file into multiple small files with each having nlines.

## 5.4. Itertools

The itertools module in the standard library provides lot of intersting tools to work with iterators.

Lets look at some of the interesting functions.

**chain** – chains multiple iterators together.

>>> it1 = iter**([**1**,** 2**,** 3**])**

>>> it2 = iter**([**4**,** 5**,** 6**])**

>>> itertools.chain**(**it1**,** it2**)**

[1, 2, 3, 4, 5, 6]

**izip** – iterable version of zip

>>> **for** x**,** y **in** itertools.izip**([**"a"**,** "b"**,** "c"**],** **[**1**,** 2**,** 3**]):**

... **print** x**,** y

...

a 1

b 2

c 3

**Problem 8:**Write a function peep, that takes an iterator as argument and returns the first element and an equivalant iterator.

>>> it = iter**(**range**(**5**))**

>>> x**,** it1 = peep**(**it**)**

>>> **print** x**,** list**(**it1**)**

0 [0, 1, 2, 3, 4]

**Problem 9:**The built-in function enumerate takes an iteratable and returns an iterator over pairs (index, value) for each value in the source.

>>> list**(**enumerate**([**"a"**,** "b"**,** "c"**])**

[(0, "a"), (1, "b"), (2, "c")]

>>> **for** i**,** c **in** enumerate**([**"a"**,** "b"**,** "c"**]):**

... **print** i**,** c

...

0 a

1 b

2 c

Write a function my\_enumerate that works like enumerate.

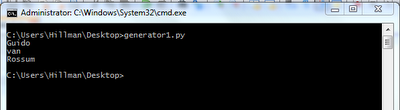
**Problem 10:**Implement a function izip that works like itertools.izip.

### Another Example: [Python Generators - examples and applications](http://everydayscripting.blogspot.in/2009/07/python-generators-examples-and.html)

Python generators are very handy - albeit a little hard to understand if you have never worked with them before. It took me a while to find out what it is they actually do, and even longer to figure out a use for one.  
  
If you don't know what a generator is, or what one can do, this is the definition I came up with while learning about them: A generator is like an extended function that remembers that state it was in the last time it was called, and will continue from there using that same state. Generators look a lot like regular functions, but they have that characteristic "yield" keyword that sets them apart from conventional functions.  
  
Here is a simple example:

# Generator example  
def printName(name):  
 for section in name.split(' '):  
 yield section  
  
for section in printName("Guido van Rossum"):  
 print section

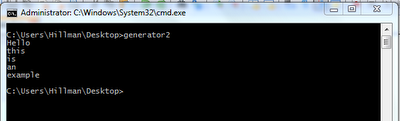
All this generator does is it takes a name and splits it into the different sections that are separated by spaces. It's output will look like this:

[](http://2.bp.blogspot.com/_7au8LV-2iSE/Sm6B2cyo6zI/AAAAAAAAAEk/m1ibN-8xphM/s1600-h/generator1.PNG)

In this example, we are treating the generator like an iterator - which is a very common usage of generators, but definitely not the only usage.  
  
Another way we can use a generator is by assigning it to another variable - then, every time we want the next value "yielded" by the generator, we call "<variablename>.next()".

def getNextWordGenerator():  
 yield "Hello"  
 yield "this"  
 yield "is"  
 yield "an"  
 yield "example"  
  
generator = getNextWordGenerator()  
  
print generator.next()  
print generator.next()  
print generator.next()  
print generator.next()  
print generator.next()

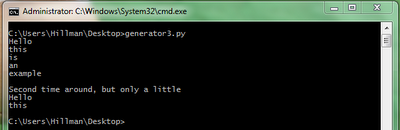
Running this will give us:

[](http://3.bp.blogspot.com/_7au8LV-2iSE/Sm6JH6hjI0I/AAAAAAAAAEs/E8FkPVGa_B8/s1600-h/generator2.png)

But, if we try to call "generator.next()" in this example, we will get an error. This is because there is nothing left in the generator to yield. This may be what you want to happen, but maybe not. Sometimes you would rather have it start all over again. In that case, you can just put all of your yield statements in a "while True:" loop, like so:

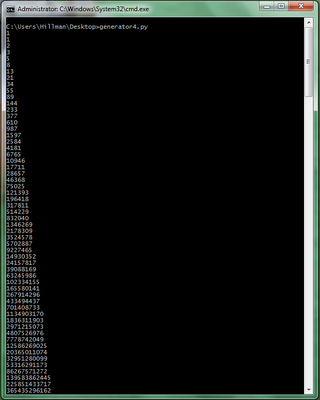
def getNextWordGenerator():  
 while True:  
 yield "Hello"  
 yield "this"  
 yield "is"  
 yield "an"  
 yield "example"  
  
generator = getNextWordGenerator()  
  
print generator.next()  
print generator.next()  
print generator.next()  
print generator.next()  
print generator.next()  
print  
print 'Second time around, but only a little'  
print generator.next()  
print generator.next()

The output this time around will look like this:

[](http://1.bp.blogspot.com/_7au8LV-2iSE/Sm6KtOh8_lI/AAAAAAAAAE0/O7svXf6pP2U/s1600-h/generator3.PNG)

This is just the beginning, however - you can do more complicated and useful things (of course, that depends on your definition of useful).  
  
Lets says, for example, you are a member of [Project Euler](http://projecteuler.net/) and you need a function that will spit out Fibonacci numbers. There are a few ways of calculating Fibonacci numbers, but I think that Python generators is the easiest, and fastest way. Here is an example:

def fib():  
 x,y = 1,1  
 while True:  
 yield x  
 x,y = y, x+y  
  
for num in fib():  
 print num

Output:  
  
[](http://3.bp.blogspot.com/_7au8LV-2iSE/Sm6QfPHaTAI/AAAAAAAAAE8/Ygh1Jon8EDQ/s1600-h/generator4.PNG)  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Running this will print out Fibonacci numbers at an alarming rate. You will have to hit CTRL-C to kill the script, or it will keep on going forever - and fast. I felt...odd...putting a "while True:" loop in my code for this first time when doing this. That seemed like something I should avoid. Don't worry though - it is not going to peg your processor or anything - the loop will only be run when called. It will not be run in he background without you knowing about it.  
  
This example also acts as a reminder that a generator can remember it's state. In this case, the generator keeps track of the fact that it is in a loop, and it remembers the values in "x" and "y". So, the next time fib().next() is called by the iterator (this all happens automatically in the for loop) it **doesn't** start over at the top of the generator with "x,y = 1,1", but in the loop where it last left off. Very handy.  
  
So, how can this help in every day scripting situations? Well, I am not too sure. I have found them very useful in solutions to Project Euler problems, and I used a generator in a school project to return all valid adjacent points to a point that I passed into the generator - but other than that, I haven't found a great reason to use them frequently in every day situations. But, there are some libraries and other built in functions of Python that do use them heavily. For example - xrange uses a "generator" rather than creating a list first in memory like "range" does. "count" in the "itertools" uses a generator to give you the next number in sequence for as long as you want.  
  
You can use generators to simulate [continuation programming](http://en.wikipedia.org/wiki/Continuation) - which also takes some time getting used to, but is pretty neat when you see it. Here is an example of that:

def fib():  
 x,y = 1,1  
 while True:  
 yield x  
 x,y = y, x+y  
  
def odd(seq):  
 for number in seq:  
 if number % 2:  
 yield number  
   
def underFourMillion(seq):  
 for number in seq:  
 if number > 4000000:  
 break  
 yield number   
   
print sum(odd(underFourMillion(fib())))

This program sums all of the odd Fibonacci numbers that are under 4 million, but at no point does it store anything in a data structure. The "sum" built-in method will add numbers coming from the "odd" generator, which yields any odd numbers coming from the "underFourMillion" generator, which yields any number that is under 4 million coming from the "fib" generator. Neat. It is easy to change any part of this program, or even add another "filter" generator to the mix.  
  
Do you have any other uses for generators? Share them in the comments.