

# From scripting to object-oriented Python programming

## How to become an object-oriented Python programmer

Skill Level: Intermediate

[Noah Gift \(noah.gift@giftcs.com\)](mailto:noah.gift@giftcs.com)

Software Engineer  
Giftcs, LLC

14 Jul 2008

Often it is difficult to make the transition from procedural scripting to object-oriented programming. This article explores how to reuse knowledge from PHP, Bash, or Python scripting to transition to object-oriented programming in Python. The article also briefly touches on the appropriate use of functional programming.

## Introduction

Python has begun soaring in popularity in recent years, and part of the reason is that the language is very flexible, yet incredibly powerful. Python can be used for systems administration, Web development, GUI programming, scientific computing, and more. The main aim of this article is to introduce people who are used to scripting procedural code in Bash, PHP, or some other language, and to assist them in moving into object-oriented Python developing. This rising popularity of Python means that developers currently using other programming languages might be called upon to do some of their projects in Python, in addition to their favorite language.

Procedural programming certainly has its place, and it can be a highly effective way to solve a problem. On a very basic level, procedural programming can be defined as a list of instructions, and Bash and PHP often are written in such a manner. Because of Python's popularity, though, PHP and Bash scripters who are Web developers or systems administrators are getting thrown into situations in which they

have to learn both object-oriented programming and Python at the same time.

Object-oriented can be a lot to digest at once, so this article takes on procedural Bash and PHP scripts, and first converts them to procedural Python. As a final step, they get converted into the end goal of object-oriented Python. The article concludes by touching on a few advantages of object-oriented Python, and then finally with some disadvantages in which either procedural or functional programming might be a better fit. By the end of this article, Bash or PHP programmers should be able to jump into object-oriented Python projects without fear.

If you haven't heard of functional programming before, I would highly recommend reading some of the articles on functional programming in the [resources](#) section. Briefly, though, functional programming can be described as "passing functions around." Often functional programming can be a more succinct and clear way to express an idea than object-oriented programming.

## Writing a disk-monitoring function in PHP and Bash

While PHP is mostly meant to be run inside a browser, it can also perform system calls by way of the exec function. The first example, written in PHP, will capture the output of the shell command "df -h," place the output into an array, and then examine each line of output against a regular expression. If the line matches the regular expression, then you print out that line. If you want to run this example from home, you will only need to call this script **index.php** and place it in the served-out directory of an Apache/mod\_php server.

### PHP disk-monitoring example

```
<html>
<body>
<?php

//Analyzes disk usage
//Takes regex pattern and message
function disk_space( $pattern="/2[0-9]%/ ", $message="CAPACITY WARNING:" )

{
    exec(escapeshellcmd("df -h"),$output_lines,$return_value);
    foreach ($output_lines as $output) {
        if (preg_match( $pattern, $output ))
            echo "<b>$message</b> $output <br />";
    }
}

disk_space()

?>
</body>
</html>
```

If you run this Web page in a browser, you get the following results:

```
CAPACITY WARNING: /dev/sda1 3.8G 694M 2.9G 20% /
```

Looking at the code, you can see that the regular expression pattern was set to match a line that contained 20-29%. This could be modified easily to fit some other flag, such as 90-99%, as 20% is a very low disk capacity.

Next let's look at doing this in a Bash function. In Bash, the problem is much easier to solve because you are really processing system calls. In this example, you don't even need to use an array or a regular expression library, because a pipe to grep is much easier. Setting default parameters for functions in Bash are always a bit verbose, though.

### Bash disk-monitoring example

```
#!/usr/bin/env bash

#function flags disk usage takes pattern and message optionally
function disk_space ()
{
    #checks for pattern parameter
    if [ "$1" != "" ]; then
        pattern=$1
    else
        pattern="2[0-9]%"
    fi

    #checks for message parameter
    if [ "$2" != "" ]; then
        message=$2
    else
        message="CAPACITY WARNING:"
    fi

    #looks at output for pattern to flag
    output_lines=`df -h | grep $pattern`
    if [ "$output_lines" != "" ]; then
        echo $message $output_lines
    fi
}

#example of optional parameters usage
#disk_space 9[0-9]% ALERT:

disk_space
```

When you run this script, you get the same output, so you can skip showing it. What you can correlate from the PHP version of the script and Bash version is that this procedural code does in fact run like a set of instructions. It is almost as if the computer is a small child, and you are telling the child how to do something like tie his shoes for the first time. Before you get into thinking in the "Object-Oriented Paradigm" in Python, let's look at making a procedural version of this same function in Python.

## Python disk-monitoring example

```
from subprocess import Popen, PIPE
import re

def disk_space(pattern="2[0-9]%", message="CAPACITY WARNING:"):

    #takes shell command output
    ps = Popen("df -h", shell=True, stdout=PIPE, stderr=PIPE)
    output_lines = ps.stdout.readlines()
    for line in output_lines():
        line = line.strip()
        if re.search(pattern, line):
            print "%s %s" % (message, line)

disk_space()
```

Looking over the procedural Python version of our code, it quite similar to both the Bash and PHP versions. With Python, the subprocess module handles making the system call to the shell command, and puts it into a list, called in array in Bash or PHP. Much like the PHP version, I then iterate over the items in the list that are lines of the standard out of the command. I look for a regular expression that makes the pattern I am looking for, and then print the line of the disk report with a special message that gets injected. This is a classic example of how a top-down scripting problem can be solved, but in the next section you change your approach completely and think in terms of objects.

## From procedural to object-oriented Python

Procedural programming is often the most natural style of programming for a beginning developer, and it is also highly effective for many problems. On the other hand, object-oriented programming can be a very useful way to create abstraction, and thus reusable code. Often procedural code can show cracks in its foundation when a project approaches a certain level of complexity, though. Let's jump right into an object-oriented version of the last example and see how that changes things.

## Object-oriented Python disk-monitoring script

```
#!/usr/bin/env python

from subprocess import Popen, PIPE
import re

class DiskMonitor():
    """Disk Monitoring Class"""
    def __init__(self,
                 pattern="2[0-9]%",
                 message="CAPACITY WARNING",
                 cmd = "df -h"):
        self.pattern = pattern
        self.message = message
        self.cmd = cmd

    def disk_space(self):
        """Disk space capacity flag method"""
        ps = Popen(self.cmd, shell=True, stdout=PIPE, stderr=PIPE)
```

```

        output_lines = ps.stdout.readlines()
        for line in output_lines:
            line = line.strip()
            if re.search(self.pattern,line):
                print "%s %s" % (self.message,line)

if __name__ == "__main__":
    d = DiskMonitor()
    d.disk_space()

```

In looking at the object-oriented version of the code, you can see that the code becomes more abstract. Sometimes too much abstraction can lead to design problems, but in this case it allows you to separate the problem into more reusable pieces. The `DiskMonitor` class has an `__init__` method where you define your new parameters, and the `disk_space` function is now a method inside of the class.

With this new style, you can easily reuse and customize things without changing the original code, as is often necessary with procedural code. One of the more powerful, and also overused, aspects of object-oriented design is inheritance. Inheritance allows you to reuse and customize existing code inside a new class. Let's see what that might look like in this next example.

### Object-oriented Python disk-monitoring script using inheritance

```

#!/usr/bin/env python

from subprocess import Popen, PIPE
import re

class DiskMonitor():
    """Disk Monitoring Class"""
    def __init__(self,
                 pattern="2[0-9]%",
                 message="CAPACITY WARNING",
                 cmd = "df -h"):
        self.pattern = pattern
        self.message = message
        self.cmd = cmd

    def disk_space(self):
        """Disk space capacity flag method"""
        ps = Popen(self.cmd, shell=True,stdout=PIPE,stderr=PIPE)
        output_lines = ps.stdout.readlines()
        for line in output_lines:
            line = line.strip()
            if re.search(self.pattern,line):
                print "%s %s" % (self.message,line)

class MyDiskMonitor(DiskMonitor):
    """Customized Disk Monitoring Class"""

    def disk_space(self):
        ps = Popen(self.cmd, shell=True,stdout=PIPE,stderr=PIPE)
        print "RAW DISK REPORT:"
        print ps.stdout.read()

if __name__ == "__main__":
    d = MyDiskMonitor()
    d.disk_space()

```

If you run this version of the script that uses inheritance, you get the following output:

```
RAW DISK REPORT:
Filesystem      Size  Used Avail Use% Mounted on
/dev/sdal       3.8G  694M  2.9G  20% /
varrun          252M   48K  252M   1% /var/run
varlock         252M    0  252M   0% /var/lock
udev            252M   52K  252M   1% /dev
devshm          252M    0  252M   0% /dev/shm
```

This output is very different from the previous -- flagged -- version, because it is just the unfiltered results of `df -h` with a print statement injected at the top. You were able to change the intent of the `disk_space` method completely by overriding the method in the `MyDiskMonitor` class.

The Python magic that allowed you to reuse the attributes from the other class were this "`MyDiskMonitor(DiskMonitor)`." You only needed to place the name of the previous class inside of paranthesis when you defined the name of the new class. As soon as that happens, you instantly gain access to the other classes attributes to do with as you wish. The fun doesn't stop there, either. You could further customize the new class by adding another method, perhaps called `disk_alert(self)`, which would e-mail flagged messages. This is the beauty of object-oriented design; it allows experienced developers to constantly reuse code they have written, and save themselves quite a bit of time.

There is a dark side to object-oriented programming, as well, unfortunately. All of this abstraction comes at a cost of complexity, and if abstraction is taken far enough, it can get downright ugly. Because Python supports multiple inheritance, abstraction can be taken to a level of complexity that is quite unhealthy. Can you imagine having to look at several files to just write one method? Believe it or not, this does happen, and it represents the unfortunate reality of one side of object-oriented programming.

One alternate to object-oriented programming is *functional programming*, and Python offers resources to program in a functional style, as well as object oriented and procedural. In the final example, let's take a look out how to write our now tiresome disk-monitoring code in a functional style.

```
There's a simple answer to the problem of overuse of inheritance:
Think about whether the problem can be refactored as "A contains
B" (a.b) rather than "A is a subclass of B" (A(B)). If so, the 'contains'
approach is almost always better.
```

## Functional-style Python disk-monitoring script

```
from subprocess import Popen, PIPE
import re
```

```
def disk_space(pattern="2[0-9]%", message="CAPACITY WARNING: "):
    #Generator Pipeline To Search For Critical Items
    ps = Popen("df -h", shell=True, stdout=PIPE, stderr=PIPE)
    outline = (line.split() for line in ps.stdout)
    flag = (" ".join(row) for row in outline if re.search(pattern, row[-2]))
    for line in flag:
        print "%s %s" % (message, line)

disk_space()
```

If you look at this last example, it is much different from anything else you have seen in this article. If you walk through the code line by line, you can first start with something you have seen before in "ps" variable. The next two lines of code-use generator expressions to handle the file object ps.stdout and to parse it and search it for the lines you are looking for. If you cut and pasted these lines of code into an interactive Python shell, you would see that outline and flag are both generator objects if you printed them. Generator objects come with a next method attached to them, and as such, allow you "pipeline" actions together.

The outline line strips the newline character from one row, and then passes that one row down to the next generator expression, which searches for a regular expression match, in each row, one at a time, and then passes the output to flag. This type of compact workflow can be an alternative to the object-oriented programming style, and it is quite interesting. There are drawbacks to this style as well, though, as the conciseness of the code can lead to errors that are difficult to debug unless each line of code is executed independantly. Functional programming also stretches the brain, as it makes you think of solving problems by chaining solutions together. This is quite different from either procedural or object-oriented styles.

## Summary

This article was somewhat experimental, as it went from Bash and PHP, to procedural, object-oriented, and finally functional Python using the same basic code. I hope it illustrated that Python is a very flexible and powerful language that developers in other programming languages can also learn to appreciate. As Python continues to grow in popularity, it will become more important for other developers to learn about in addition to their language of choice.

Two of the largest recent growth areas of Python are Web development and systems administration. In terms of Web development, developers in PHP may soon have to make weekly choices about which project makes more sense in Python, and which project makes more sense in PHP. For systems administrators, Bash and Perl scripters, are often being asked to look at doing some of their projects in Python. Partly this is out of choice, and partly because many vendors are offering Python API's for their products. Having a bit of Python in your toolkit can never hurt anyone.

## Downloads

Description	Name	Size	Download method
Sample Object Oriented Python scripts	scripting_oo_code	54K	<a href="#">HTTP</a>

[Information about download methods](#)

## Resources

- The [PHP Manual](#) consists of reference information plus PHP features.
- [PHP Shell](#).
- [Understanding MVC in PHP](#) covers the basics of MVC Web frameworks.
- [Charming Python: Functional programming in Python, Part 1](#) (David Mertz, developerWorks 01 March 2001) discusses general concepts of functional programming, and illustrates ways of implementing functional techniques in Python.
- Take a tour of Python's features suitable for implementing programs with [Functional Programming in Python AMK](#).
- [Charming Python: Functional programming in Python, Part 2](#) (David Mertz, developerWorks 01 April 2001) introduces different paradigms of program problem-solving.
- [Generator's David Beazely](#) discusses various techniques for using generator functions and generator expressions in the context of systems programming.
- [New Style Classes Python](#)

## About the author

Noah Gift

Noah Gift is the co-author of "Python For Unix and Linux" by O'Reilly. He is an author, speaker, consultant, and community leader, writing for publications such as IBM Developerworks, Red Hat Magazine, O'Reilly, and MacTech. His consulting company's website is [www.giftcs.com](http://www.giftcs.com), and his personal website is [www.noahgift.com](http://www.noahgift.com). Noah is also the current organizer for [www.pyatl.org](http://www.pyatl.org), which is the Python User Group for Atlanta, GA.

He has a Master's degree in CIS from Cal State Los Angeles, B.S. in Nutritional Science from Cal Poly San Luis Obispo, is an Apple and LPI certified SysAdmin, and has worked at companies such as, Caltech, Disney Feature Animation, Sony Imageworks, and Turner Studios. In his free time he enjoys spending time with his wife Leah, and their son Liam, playing the piano, and exercising religiously.