Chapter No. 2
"Creating Only One Object with the Singleton Pattern"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.2 "Creating Only One Object with the Singleton Pattern"
A synopsis of the book’s content
Information on where to buy this book

About the Author

**Gennadiy Zlobin** works as Lead Software Engineer and Technical Leader in the Russian music service, Zvooq.ru. His current employer is Zvooq Ltd. He has been using Python as the primary language for more than 4 years, enjoying its elegance and power every day. His professional interests include high-load software architectures, good engineering practices, Android OS, and natural language processing.

Previously, he worked for the company that had the first search engine in Russia, called Rambler. He was engaged in airline tickets' meta search service and Rambler's index page.

I would like to thank my wife, Jane, for her patience and support. I really appreciate it.

I am also grateful to my parents, Galina and Vitality for believing in me. I love all of you.

For More Information:
Learning Python Design Patterns

Python is a great programming language, elegant and concise, and at the same time, very powerful. It has all the essential object-oriented features and can be used to implement design patterns. A design pattern is a general reusable solution to a commonly occurring problem within a given context. In everyday work, a programmer faces issues that have been solved so many times in the past by other developers that they have evolved common patterns to solve them.

The design pattern is not a concrete step to solve a problem, such as an algorithm; it is rather a practice or a description of how to solve a problem that can be used in different situations and implemented in different languages.

The design pattern accelerates the development process, providing a proven practice to solve some type of problem. It is often more preferable than using an unproven one because invisible problems often occur during the implementation, and the solving of unforeseen problems slows down the development dramatically.

Besides that, it's a tool of communication between programmers. It's much easier to say, "We use here the observer design pattern" rather than describing what the code actually does.

Studying design patterns is a good next step on the road to becoming a great developer, and this book is a good jumpstart.

What This Book Covers

Chapter 1, Model-View-Controller, describes what the model, view, and controller are, how to use them together, and ends with the implementation of a very simple URL shortening service.

Chapter 2, Creating Only One Object with the Singleton Pattern, describes ways to create a class whose instantiated object will only be one throughout the lifecycle of an application.

Chapter 3, Building Factories to Create Objects, describes the simple factory, Factory Method, Abstract Factory patterns, and how to use them to separate object creation.

Chapter 4, The Facade Design Pattern, is about simplifying the interface of a complex subsystem to facilitate the development.

For More Information:  
Chapter 5, *Facilitating Object Communication with Proxy and Observer Patterns*, is a pattern for implementing a publisher-subscriber model and a proxy, which provides an object that controls access to another object.

Chapter 6, *Encapsulating Calls with the Command Pattern*, describes a pattern that encapsulates an action and its parameters.

Chapter 7, *Redefining Algorithms with the Template Method*, is about a pattern that provides the ability to create variations of the algorithm with minimum modifications.

For More Information:
Creating Only One Object with the Singleton Pattern

There are situations where you need to create only one instance of data throughout the lifetime of a program. This can be a class instance, a list, or a dictionary, for example. The creation of a second instance is undesirable. This can result in logical errors or malfunctioning of the program. The design pattern that allows you to create only one instance of data is called singleton. In this chapter, you will learn about module-level, classic, and borg singletons; you’ll also learn about how they work, when to use them, and build a two-threaded web crawler that uses a singleton to access the shared resource.

Singleton is the best candidate when the requirements are as follows:

- If you need to control concurrent access to a shared resource
- If you need a global point of access for the resource from multiple or different parts of the system
- If you need to have only one object

Some typical use cases of a singleton are:

- The logging class and its subclasses (global point of access for the logging class to send messages to log)
- Printer spooler (your application should only have a single instance of the spooler in order to avoid having a conflicting request for the same resource)
- Managing a connection to a database
- File manager
- Retrieving and storing information on external configuration files
- Read-only singletons storing some global states (user language, time, time zone, application path, and so on)

For More Information:
There are several ways to implement singletons. We will look at a module-level singleton, classic singletons, and a borg singleton.

**A module-level singleton**

All modules are singletons by nature because of Python's module importing steps:

1. Check whether a module is already imported.
2. If yes, return it.
3. If not, find a module, initialize it, and return it.
4. Initializing a module means executing code, including all module-level assignments. When you import the module for the first time, all initializations are done; however, if you try to import the module for the second time, Python will return the initialized module. Thus, the initialization will not be done, and you get a previously imported module with all of its data.

So, if you want to quickly make a singleton, use the following code and keep the shared data as the module attribute:

```python
singleton.py:
only_one_var = "I'm only one var"

module1.py:
import singleton
classic
print singleton.only_one_var
singleton.only_one_var += " after modification"
import module2

module2.py:
import singleton
classic
print singleton.only_one_var
```

Here, if you try to import a global variable in a `singleton.py` module and change its value in the `module1.py` module, `module2.py` will receive a changed variable.

This function is quick and sometimes is all you need; however, we need to consider the following points:

- It's pretty error-prone. For example, if you happen to forget the global statements, variables local to the function will be created and the module's variables won't be changed, which is not what you want.
• It's ugly, especially if you have a lot of objects that should remain as singletons.
• It pollutes the module namespace with unnecessary variables.
• They don't permit lazy allocation and initialization; all global variables will be loaded during the module import process.
• It's not possible to reuse the code because you cannot use the inheritance.
• It has no special methods and no object-oriented programming benefits at all.

A classic singleton
In a classic singleton in Python, we check whether an instance is already created. If it is created, we return it; otherwise, we create a new instance, assign it to a class attribute, and return it.

Let's try to create a dedicated singleton class:

class Singleton(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

Here, before creating the instance, we check for the special __new__ method that is called right before __init__ if we had created an instance earlier. If not, we create a new instance; otherwise, we return the already created instance.

Let's check how it works:

>>> singleton = Singleton()
>>> another_singleton = Singleton()
>>> singleton is another_singleton
True
>>> singleton.only_one_var = "I'm only one var"
>>> another_singleton.only_one_var
'I'm only one var'

Try to subclass the Singleton class with another one:

class Child(Singleton):
    pass

If some class is a successor of Singleton, all successor's instances should also be the instances of Singleton, thus sharing its states. But this doesn't work, as illustrated in the following code:

>>> child = Child()
>>> child is singleton
False

For More Information:
Creating Only One Object with the Singleton Pattern

```python
>>> child.only_one_var
AttributeError: Child instance has no attribute 'only_one_var'
```

To avoid this situation, the borg singleton is used.

**The borg singleton**

Borg is also known as **monostate**. In the borg pattern, all of the instances are different, but they share the same state.

In the following code, the shared state is maintained in the `_shared_state` attribute. And all new instances of the `Borg` class will have this state as defined in the `__new__` class method:

```python
class Borg(object):
    _shared_state = {}

def __new__(cls, *args, **kwargs):
    obj = super(Borg, cls).__new__(cls, *args, **kwargs)
    obj.__dict__ = cls._shared_state
    return obj
```

Generally, Python stores the instance state in the `__dict__` dictionary and when instantiated normally, every instance will have its own `__dict__`. But, here we deliberately assign the class variable `_shared_state` to all of the created instances.

The following code shows how it works with subclassing:

```python
class Child(Borg):
    pass

>>> borg = Borg()
>>> another_borg = Borg()
>>> borg is another_borg
False
>>> child = Child()
>>> borg.only_one_var = "I'm the only one var"
>>> child.only_one_var
'I'm the only one var'
```

So, despite the fact that you can't compare objects by their identity, using the `is` statement, all child objects share the parents' state.

If you want to have a class that is a descendant of the `Borg` class but has a different state, you can reset `shared_state` as follows:

---

For More Information:
class AnotherChild(Borg):
    _shared_state = {}

>>> another_child = AnotherChild()
>>> another_child.only_one_var
AttributeError: AnotherChild instance has no attribute 'shared_state'

It is up to you to decide which type of singleton should be used. If you expect that your singleton will not be inherited, you can choose the classic singleton; otherwise, it's better to stick with borg.

Implementation in Python

As a practical example, we'll create a simple web crawler that scans a website you open on it, follows all the links that lead to the same website but to other pages, and downloads all of the images it'll find.

To do this, we'll need two functions: a function that scans a website for links that lead to other pages to build a set of pages to visit, and a function that scans a page for images and downloads them.

To make it quicker, we'll download images in two threads. These two threads should not interfere with each other, so don't scan pages if another thread has already scanned them, and don't download images that are already downloaded.

So, a set with downloaded images and scanned web pages will be a shared resource for our application, and we'll keep it in a singleton instance.

In this example, you will need a library for parsing and screen scraping websites named BeautifulSoup and an HTTP client library, httplib2. It should be sufficient to install both with either of the following commands:

• $ sudo pip install BeautifulSoup httplib2
• $ sudo easy_install BeautifulSoup httplib2

First of all, we'll create a Singleton class. Let's use the classic singleton in the following example:

```python
import httplib2
import os
import re
import threading
import urllib
from urlparse import urlparse, urljoin
```

For More Information:
Creating Only One Object with the Singleton Pattern

```python
from BeautifulSoup import BeautifulSoup

class Singleton(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

It will return the singleton objects to all parts of the code that request it.

Next, we'll create a class for creating a thread. In this thread, we'll download images from the website:

class ImageDownloaderThread(threading.Thread):
    """A thread for downloading images in parallel."""
    def __init__(self, thread_id, name, counter):
        threading.Thread.__init__(self)
        self.name = name

    def run(self):
        print 'Starting thread ', self.name
        download_images(self.name)
        print 'Finished thread ', self.name

The following function traverses the website using BFS algorithm, finds links, and adds them to a set for further downloading. We are able to specify the maximum links to follow if the website is too large:

def traverse_site(max_links=10):
    link_parser_singleton = Singleton()

    while link_parser_singleton.queue_to_parse:
        if len(link_parser_singleton.to_visit) == max_links:
            return
        url = link_parser_singleton.queue_to_parse.pop()

        http = httplib2.Http()
        try:
            status, response = http.request(url)
        except Exception:
            continue
```

For More Information:
# Skip if not a web page
if status.get('content-type') != 'text/html':
    continue

# Add the link to queue for downloading images
link_parser_singleton.to_visit.add(url)
print 'Added', url, 'to queue'

bs = BeautifulSoup(response)
for link in BeautifulSoup.findAll(bs, 'a'):
    link_url = link.get('href')

    # <img> tag may not contain href attribute
    if not link_url:
        continue

    parsed = urlparse(link_url)

    # If link follows to external webpage, skip it
    if parsed.netloc and parsed.netloc != parsed_root.netloc:
        continue

    # Construct a full url from a link which can be relative
    link_url = (parsed.scheme or parsed_root.scheme) + '://'
    + (parsed.netloc or parsed_root.netloc) + parsed.path or ''

    # If link was added previously, skip it
    if link_url in link_parser_singleton.to_visit:
        continue

    # Add a link for further parsing
    link_parser_singleton.queue_to_parse = [link_url] + 
    link_parser_singleton.queue_to_parse

The following function downloads images from the last web resource page in the
singleton.to_visit queue and saves it to the img directory. Here, we use
a singleton for synchronizing shared data, which is a set of pages to visit between
two threads:

def download_images(thread_name):
    singleton = Singleton()
    # While we have pages where we have not download images
    while singleton.to_visit:
url = singleton.to_visit.pop()

http = httplib2.Http()
print thread_name, 'Starting downloading images from', url

try:
    status, response = http.request(url)
except Exception:
    continue

bs = BeautifulSoup(response)

# Find all <img> tags
images = BeautifulSoup.findAll(bs, 'img')

for image in images:
    # Get image source url which can be absolute or relative
    src = image.get('src')
    # Construct a full url. If the image url is relative,
    # it will be prepended with webpage domain.
    # If the image url is absolute, it will remain as is
    src = urljoin(url, src)

    # Get a base name, for example 'image.png' to name file locally
    basename = os.path.basename(src)

    if src not in singleton.downloaded:
        singleton.downloaded.add(src)
        print 'Downloading', src
        # Download image to local filesystem
        urllib.urlretrieve(src, os.path.join('images', basename))

print thread_name, 'finished downloading images from', url
Our client code is as follows:

```python
if __name__ == '__main__':
    root = 'http://python.org'

    parsed_root = urlparse(root)

    singleton = Singleton()
    singleton.queue_to_parse = [root]
    # A set of urls to download images from
    singleton.to_visit = set()
    # Downloaded images
    singleton.downloaded = set()

    traverse_site()

    # Create images directory if not exists
    if not os.path.exists('images'):
        os.makedirs('images')

    # Create new threads
    thread1 = ImageDownloaderThread(1, "Thread-1", 1)
    thread2 = ImageDownloaderThread(2, "Thread-2", 2)

    # Start new Threads
    thread1.start()
    thread2.start()
```

Run a crawler using the following command:

```
$ python crawler.py
```

For More Information:

Creating Only One Object with the Singleton Pattern

You should get the following output (your output may vary because the order in which the threads access resources is not predictable):

If you go to the images directory, you will find the downloaded images there.

Summary

A singleton is a design pattern for creating only one instance of a class. Modules in Python are singletons by nature. A classic singleton checks whether the instance was created earlier; if not, it creates and returns it. The Borg singleton uses shared state for all objects. In the example shown in the chapter, we used the Singleton class for accessing a shared resource and a set of URLs to fetch images from, and both threads used it to properly parallelize their work.

In the next chapter, you will learn about other patterns for creating objects, including: factory, the factory method, the abstract factory, and how they help to build objects.

For More Information:
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.