MONKEY-PATCHING: A MAGIC TRICK OR A POWERFUL TOOL?

Elizaveta Shashkova
@lisa_shashkova
elizaveta.shashkova@jetbrains.com

BIO

- Software Developer of PyCharm IDE at JetBrains
- Debugger used in PyDev & PyCharm

MONKEY-PATCHING

Monkey-patching is a dynamic modification of a class or a module at runtime

```
def safe_sqrt(num):
  # doesn't throw exception if num < 0
  if num < 0:
    return math.nan
  return math.original(num)

>>> import math
>>> math.original = math.sqrt
>>> math.sqrt = safe_sqrt
```

MONKEY-PATCHING

- Guerilla patch
  - Does something secretly and incompatibly with others
- Gorilla patch
- Monkey patch

OTHER LANGUAGES

Ruby

```ruby
class String
  def upcase
    self.reverse
  end
end

> puts "hello".upcase
"olleh"
```

PYTHON

Explicit is better than implicit

WHAT CAN WE PATCH?

Everything!

WHAT CAN WE PATCH?

Almost
Everything!

We can’t patch built-ins defined in C

LIGHT SIDE OF MONKEY-PATCHING

DARK SIDE OF MONKEY-PATCHING

LIGHT SIDE

- Workaround for bugs in third-party code

LIGHT SIDE

- Workaround for bugs in third-party code
  - Don’t forget to fix and create a pull request!

LIGHT SIDE

- Workaround for bugs in third-party code
  - Don’t forget to fix and create a pull request!
- Code instrumentation

LIGHT SIDE

- Workaround for bugs in third-party code
  - Don’t forget to fix and create a pull request!
- Code instrumentation
- Testing

LIGHT SIDE

- Workaround for bugs in third-party code
  - Don’t forget to fix and create a pull request!
- Code instrumentation
- Testing
- Changing of standard modules

EXAMPLES

LIGHT SIDE: CATCHING NEW PROCESS

LIGHT SIDE: NEW PROCESS

```
class ProcessManager:
    def __init__(self, system_fork):
        self.system_fork = system_fork

    def do_fork(self):
        pid = self.system_fork()
        if pid == 0:
            # child process
            start_tracing()
        return pid
```

LIGHT SIDE: NEW PROCESS

```
import os
manager = ProcessManager(os.fork)
os.fork = manager.do_fork


os.fork()
#tracing in a new process
```

LIGHT SIDE: IMPORT HOOK

LIGHT SIDE

- How to patch a module as soon as possible?
- Patch it right after importing

LIGHT SIDE: IMPORT HOOK

```
class ImportHookManager:
  def __init__(self, system_import):
    self.system_import = system_import

  def do_import(self, name):
    module = self.system_import(name)
    if name == "os":
      orig_fork = module.fork
      module.fork = ProcessManager(orig_fork).do_fork
    return module
```

LIGHT SIDE: IMPORT HOOK

```
import builtins
manager = ImportHookManager(builtins.__import__)
builtins.__import__ = manager.do_import



import os
# module is patched as soon as it is imported
```

LIGHT SIDE: GEVENT

LIGHT SIDE: GEVENT

- gevent provides asynchronous I/O
- Blocking system calls in the standard library

```
import socket
```

- Replace with gevent modules

```
from gevent import socket
```

LIGHT SIDE: GEVENT

- The simplest way

```
from gevent import monkey
monkey.patch_socket()
```

LIGHT SIDE

- Workaround for bugs in third-party code
- Code instrumentation
- Testing
- Changing of standard modules

LIGHT SIDE OF MONKEY-PATCHING

DARK SIDE OF MONKEY-PATCHING

DARK SIDE

Implicitness

DARK SIDE

- Implicit changes

DARK SIDE

- Implicit changes
- Undocumented changes

DARK SIDE

- Implicit changes
- Undocumented changes
- Unpredictable behaviour

DARK SIDE

- Implicit changes
- Undocumented changes
- Unpredictable behaviour
- Patching the same

EXAMPLES

DARK SIDE: GEVENT

DARK SIDE: GEVENT

- Patches standard modules
  socket, thread, time and others
- Should be done as soon as possible

RUNNING CODE UNDER DEBUGGER

- Event loop in debugger
  - Based on threads
- Event loop in user's code
  - Based on gevent loop

They should be separated

THE PROBLEM

Save the original versions of modules and use them with the patched versions

IMPORTING IN PYTHON

```
# very simple version
def import_module(name):
    if name in sys.modules:
        return sys.modules[name]
    module = internal_import(name)
    sys.modules[name] = module
    return module
```

SAVE MODULES

saved_modules.py:

```
import socket
# save the module object
save = sys.modules.pop('socket', None)

# it's not in sys.modules, reimport
import socket as socket_saved

# put the module available for patching
sys.modules['socket'] = save
```

SAVE MODULES

```
from saved_modules import saved_socket
import socket

# it will patch socket in sys.path
gevent.patch_socket()

# the patched function
socket.create_connection()
# the original function
saved_socket.create_connection()
```

THE PROBLEM

Save the original versions of modules and use them with the patched versions

Solved!

Light side of monkey-patching

- Workaround for bugs in third-party code
- Code instrumentation
- Testing
- Changing of standard modules

Dark side of monkey-patching

- Implicit changes
- Undocumented changes
- Broken readability
- Incompatibility with other patches

WHEN SHOULD I USE IT?

If and only if there are no other solutions for the problem

LIGHT SIDE OF MONKEY-PATCHING

DARK SIDE OF MONKEY-PATCHING

LIGHT SIDE OF MONKEY-PATCHING

Elizaveta Shashkova
@lisa_shashkova

DARK SIDE OF MONKEY-PATCHING