Is it possible to "hack" Python's print function?

Asked 6 years, 9 months ago Modified 5 years, 11 months ago Viewed 17k times

Note: This question is for informational purposes only. I am interested to see how deep into Python's internals it is possible to go with this.

Not very long ago, a discussion began inside a certain question regarding whether the strings passed to print statements could be modified after/during the call to `print` has been made. For example, consider the function:

```
def print_something():
    print('This cat was scared.')
```

Now, when `print` is run, then the output to the terminal should display:

```
This dog was scared.
```

Notice the word "cat" has been replaced by the word "dog". Something somewhere somehow was able to modify those internal buffers to change what was printed. Assume this is done without the original code author's explicit permission (hence, hacking/hijacking).

This comment from the wise @abarnert, in particular, got me thinking:

> There are a couple of ways to do that, but they're all very ugly, and should never be done. The least ugly way is to probably replace the `code` object inside the function with one with a different `co_consts` list. Next is probably reaching into the C API to access the str's internal buffer. [...]

So, it looks like this is actually possible.

Here's my naive way of approaching this problem:

```
>>> import inspect
>>> exec(inspect.getsource(print_something).replace('cat', 'dog'))
>>> print_something()
This dog was scared.
```

Of course, `exec` is bad, but that doesn't really answer the question, because it does not actually modify anything *during when/after* `print` is called.

How would it be done as @abarnert has explained it?

python python-3.x printing python-internals

Share
Improve this question
Follow

edited May 6, 2018 at 9:26

Peter Mortensen
31.6k ● 22 ● 109 ● 133

asked Mar 14, 2018 at 7:18

cs95
402k ● 104 ● 735 ● 788

3 By the way, the internal storage for ints is a lot simpler than strings, and floats even more so. And, as a bonus, it's a lot more obvious why it's a bad idea to change the value of `42` to `23` than why it's a bad idea to change the value of `"My name is Y"` to `"My name is X"`. – abarnert Mar 14, 2018 at 7:56

4 Answers

Sorted by: Highest score (default)

250

First, there's actually a much less hacky way. All we want to do is change what `print` prints, right?

```
_print = print
def print(*args, **kw):
    args = (arg.replace('cat', 'dog') if isinstance(arg, str) else arg
            for arg in args)
    _print(*args, **kw)
```

Or, similarly, you can monkeypatch `sys.stdout` instead of `print`.

Also, nothing wrong with the `exec … getsource …` idea. Well, of course there's *plenty* wrong with it, but less than what follows here…

But if you do want to modify the function object's code constants, we can do that.

If you really want to play around with code objects for real, you should use a library like `bytecode` (when it's finished) or `byteplay` (until then, or for older Python versions) instead of doing it manually. Even for something this trivial, the `CodeType` initializer is a pain; if you actually need to do stuff like fixing up `lnotab`, only a lunatic would do that manually.

Also, it goes without saying that not all Python implementations use CPython-style code objects. This code will work in CPython 3.7, and probably all versions back to at least 2.2 with a few minor changes (and not the code-hacking stuff, but things like generator expressions), but it won't work with any version of IronPython.

```
import types

def print_function():
    print ("This cat was scared.")

def main():
    # A function object is a wrapper around a code object, with
    # a bit of extra stuff like default values and closure cells.
    # See inspect module docs for more details.
    co = print_function.__code__
    # A code object is a wrapper around a string of bytecode, with a
    # whole bunch of extra stuff, including a list of constants used
    # by that bytecode. Again see inspect module docs. Anyway, inside
    # the bytecode for string (which you can read by typing
    # dis.dis(string) in your REPL), there's going to be an
    # instruction like LOAD_CONST 1 to load the string literal onto
    # the stack to pass to the print function, and that works by just
    # reading co.co_consts[1]. So, that's what we want to change.
    consts = tuple(c.replace("cat", "dog") if isinstance(c, str) else c
                   for c in co.co_consts)
    # Unfortunately, code objects are immutable, so we have to create
    # a new one, copying over everything except for co_consts, which
    # we'll replace. And the initializer has a zillion parameters.
    # Try help(types.CodeType) at the REPL to see the whole list.
    co = types.CodeType(
        co.co_argcount, co.co_kwonlyargcount, co.co_nlocals,
        co.co_stacksize, co.co_flags, co.co_code,
        consts, co.co_names, co.co_varnames, co.co_filename,
        co.co_name, co.co_firstlineno, co.co_lnotab,
        co.co_freevars, co.co_cellvars)
    print_function.__code__ = co
    print_function()

main()
```

What could go wrong with hacking up code objects? Mostly just segfaults, `RuntimeError`s that eat up the whole stack, more normal `RuntimeError`s that can be handled, or garbage values that will probably just raise a `TypeError` or `AttributeError` when you try to use them. For examples, try creating a code object with just a `RETURN_VALUE` with nothing on the stack (bytecode `b'S\0'` for 3.6+, `b'S'` before), or with an empty tuple for `co_consts` when there's a `LOAD_CONST 0` in the bytecode, or with `varnames` decremented by 1 so the highest `LOAD_FAST` actually loads a freevar/cellvar cell. For some real fun, if you get the `lnotab` wrong enough, your code will only segfault when run in the debugger.

Using `bytecode` or `byteplay` won't protect you from all of those problems, but they do have some basic sanity checks, and nice helpers that let you do things like insert a chunk of code and let it worry about updating all offsets and labels so you can't get it wrong, and so on. (Plus, they keep you from having to type in that ridiculous 6-line constructor, and having to debug the silly typos that come from doing so.)

Now on to #2.

I mentioned that code objects are immutable. And of course the consts are a tuple, so we can't change that directly. And the thing in the const tuple is a string, which we also can't change directly. That's why I had to build a new string to build a new tuple to build a new code object.

But what if you could change a string directly?

Well, deep enough under the covers, everything is just a pointer to some C data, right? If you're using CPython, there's a C API to access the objects, and you can use `ctypes` to access that API from within Python itself, which is such a terrible idea that they put a `pythonapi` right there in the stdlib's `ctypes` module. :) The most important trick you need to know is that `id(x)` is the actual pointer to `x` in memory (as an `int`).

Unfortunately, the C API for strings won't let us safely get at the internal storage of an already-frozen string. So screw safely, let's just read the header files and find that storage ourselves.

If you're using CPython 3.4 - 3.7 (it's different for older versions, and who knows for the future), a string literal from a module that's made of pure ASCII is going to be stored using the compact ASCII format, which means the struct ends early and the buffer of ASCII bytes follows immediately in memory. This will break (as in probably segfault) if you put a non-ASCII character in the string, or certain kinds of non-literal strings, but you can read up on the other 4 ways to access the buffer for different kinds of strings.

To make things slightly easier, I'm using the `superhackyinternals` project off my GitHub. (It's intentionally not pip-installable because you really shouldn't be using this except to experiment with your local build of the interpreter and the like.)

```
import ctypes
import internals #
https://github.com/abarnert/superhackyinternals/blob/master/internals.py

def print_function():
    print ("This cat was scared.")

def main():
    for c in print_function.__code__.co_consts:
        if isinstance(c, str):
            idx = c.find('cat')
            if idx != -1:
                # Too much to explain here; just guess and learn to
                # love the segfaults...
                p = internals.PyUnicodeObject.from_address(id(c))
                assert p.compact and p.ascii
                addr = id(c) + internals.PyUnicodeObject.utf8_length.offset
                buf = (ctypes.c_int8 * 3).from_address(addr + idx)
                buf[:3] = b'dog'
```

```
    print_function()

main()
```

If you want to play with this stuff, `int` is a whole lot simpler under the covers than `str`. And it's a lot easier to guess what you can break by changing the value of `2` to `1`, right? Actually, forget imagining, let's just do it (using the types from `superhackyinternals` again):

```
>>> n = 2
>>> pn = PyLongObject.from_address(id(n))
>>> pn.ob_digit[0]
2
>>> pn.ob_digit[0] = 1
>>> 2
1
>>> n * 3
3
>>> i = 10
>>> while i < 40:
...     i *= 2
...     print(i)
10
10
10
```

… pretend that code box has an infinite-length scrollbar.

I tried the same thing in IPython, and the first time I tried to evaluate `2` at the prompt, it went into some kind of uninterruptable infinite loop. Presumably it's using the number `2` for something in its REPL loop, while the stock interpreter isn't?

Share

Improve this answer

Follow

edited Aug 8, 2018 at 21:37

Engineero

12.9k ● 5 ● 55 ● 77

answered Mar 14, 2018 at 7:41

abarnert

365k ● 54 ● 622 ● 690

11 @COLDSPEED The code-munging is *arguably* reasonable Python, although you generally only want to touch code objects for much better reasons (e.g., running the bytecode through a custom optimizer). Accessing the internal storage of a `PyUnicodeObject`, on the other hand, that's probably really only Python in the sense that a Python interpreter will run it… – abarnert Mar 14, 2018 at 7:48

4 Your first code snippet raises `NameError: name 'arg' is not defined`. Did you mean: `args = [arg.replace('cat', 'dog') if isinstance(arg, str) else arg for arg in args]`? An arguably better way to write this would be: `args = [str(arg).replace('cat', 'dog') for arg in args]`. Another, even shorter, option: `args = map(lambda a: str(a).replace('cat', 'dog'), args)`. This has the added benefit that `args` is lazy (which could also be accomplished by replacing the above list comprehension with a generator one— `*args` works either way). – Konstantin Mar 14, 2018 at 14:09

1 @coldspeed Yeah, IIRC I'm only using the `PyUnicodeObject` struct definition, but copying that into the answer would I think just get in the way, and I think the readme and/or source comments to `superhackyinternals` actually explain how to access the buffer (at least well enough to remind me next time I care; not sure if it'll be enough for anyone else…), which I didn't want to get into here. The relevant part is how to get from a live Python object to its `PyObject *` via `ctypes`. (And maybe simulating pointer arithmetic, avoiding automatic `char_p` conversions, etc.) – abarnert Mar 14, 2018 at 17:31

1 @jpmc26 I don't think you need to do it *before* importing modules, as long as you do it before they print. Modules will do the name lookup every time, unless they explicitly bind `print` to a name. You can also bind the name `print` for them: `import yourmodule; yourmodule.print = badprint`. – leewz Mar 15, 2018 at 15:02

1 @abarnert: I've noticed you've warned often about doing this (eg. "you never want to actually do this", "why it's a bad idea to change the value", etc.). It's not exactly clear what could possibly go wrong (sarcasm), would you be willing to elaborate a bit on that? It could possibly help for those tempted to blindly try it. – l'L'l Apr 18, 2018 at 0:39

38

Monkey-patch `print`

`print` is a builtin function so it will use the `print` function defined in the `builtins` module (or `__builtin__` in Python 2). So whenever you want to modify or change the behavior of a builtin function you can simply reassign the name in that module.

This process is called `monkey-patching`.

```
# Store the real print function in another variable otherwise
# it will be inaccessible after being modified.
_print = print

# Actual implementation of the new print
def custom_print(*args, **options):
    _print('custom print called')
    _print(*args, **options)

# Change the print function globally
import builtins
builtins.print = custom_print
```

After that every `print` call will go through `custom_print`, even if the `print` is in an external module.

However you don't really want to print additional text, you want to change the text that is printed. One way to go about that is to replace it in the string that would be printed:

```
_print = print

def custom_print(*args, **options):
    # Get the desired seperator or the default whitspace
    sep = options.pop('sep', ' ')
```

```
    # Create the final string
    printed_string = sep.join(args)
    # Modify the final string
    printed_string = printed_string.replace('cat', 'dog')
    # Call the default print function
    _print(printed_string, **options)

import builtins
builtins.print = custom_print
```

And indeed if you run:

```
>>> def print_something():
...     print('This cat was scared.')
>>> print_something()
This dog was scared.
```

Or if you write that to a file:

test_file.py

```
def print_something():
    print('This cat was scared.')

print_something()
```

and import it:

```
>>> import test_file
This dog was scared.
>>> test_file.print_something()
This dog was scared.
```

So it really works as intended.

However, in case you only temporarily want to monkey-patch print you could wrap this in a context-manager:

```
import builtins

class ChangePrint(object):
    def __init__(self):
        self.old_print = print

    def __enter__(self):
        def custom_print(*args, **options):
            # Get the desired seperator or the default whitspace
            sep = options.pop('sep', ' ')
            # Create the final string
            printed_string = sep.join(args)
            # Modify the final string
```

```
            printed_string = printed_string.replace('cat', 'dog')
            # Call the default print function
            self.old_print(printed_string, **options)

        builtins.print = custom_print

    def __exit__(self, *args, **kwargs):
        builtins.print = self.old_print
```

So when you run that it depends on the context what is printed:

```
>>> with ChangePrint() as x:
...     test_file.print_something()
...
This dog was scared.
>>> test_file.print_something()
This cat was scared.
```

So that's how you could "hack" `print` by monkey-patching.

Modify the target instead of the `print`

If you look at the signature of `print` you'll notice a `file` argument which is `sys.stdout` by default. Note that this is a dynamic default argument (it **really** looks up `sys.stdout` every time you call `print`) and not like normal default arguments in Python. So if you change `sys.stdout` `print` will actually print to the different target even more convenient that Python also provides a `redirect_stdout` function (from Python 3.4 on, but it's easy to create an equivalent function for earlier Python versions).

The downside is that it won't work for `print` statements that don't print to `sys.stdout` and that creating your own `stdout` isn't really straightforward.

```
import io
import sys

class CustomStdout(object):
    def __init__(self, *args, **kwargs):
        self.current_stdout = sys.stdout

    def write(self, string):
        self.current_stdout.write(string.replace('cat', 'dog'))
```

However this also works:

```
>>> import contextlib
>>> with contextlib.redirect_stdout(CustomStdout()):
...     test_file.print_something()
...
```

```
This dog was scared.
>>> test_file.print_something()
This cat was scared.
```

Summary

Some of these points have already be mentioned by @abarnet but I wanted to explore these options in more detail. Especially how to modify it across modules (using `builtins` / `__builtin__`) and how to make that change only temporary (using contextmanagers).

Share

Improve this answer

Follow

edited Mar 18, 2018 at 18:58

answered Mar 18, 2018 at 18:53

MSeifert

152k ● 41 ● 349 ● 366

5 Yeah, the closest thing to this question anyone should ever actually want to do is `redirect_stdout`, so it's nice to have a clear answer that leads to that. – abarnert Mar 23, 2018 at 15:02

5

A simple way to capture all output from a `print` function and then process it, is to change the output stream to something else, e.g. a file.

I'll use a `PHP` naming conventions (ob_start, ob_get_contents,...)

```
from functools import partial
output_buffer = None
print_orig = print
def ob_start(fname="print.txt"):
    global print
    global output_buffer
    print = partial(print_orig, file=output_buffer)
    output_buffer = open(fname, 'w')
def ob_end():
    global output_buffer
    close(output_buffer)
    print = print_orig
def ob_get_contents(fname="print.txt"):
    return open(fname, 'r').read()
```

Usage:

```
print ("Hi John")
ob_start()
print ("Hi John")
ob_end()
print (ob_get_contents().replace("Hi", "Bye"))
```

Would print

> Hi John Bye John

Share Improve this answer Follow

edited Mar 14, 2018 at 8:09

answered Mar 14, 2018 at 8:04

Uri Goren

13.7k ● 7 ● 62 ● 112

5

Let's combine this with frame introspection!

```
import sys

_print = print

def print(*args, **kw):
    frame = sys._getframe(1)
    _print(frame.f_code.co_name)
    _print(*args, **kw)

def greetly(name, greeting = "Hi")
    print(f"{greeting}, {name}!")

class Greeter:
    def __init__(self, greeting = "Hi"):
        self.greeting = greeting
    def greet(self, name):
        print(f"{self.greeting}, {name}!")
```

You'll find this trick prefaces every greeting with the calling function or method. This might be very useful for logging or debugging; especially as it lets you "hijack" print statements in third party code.

Share Improve this answer Follow

answered Mar 23, 2018 at 17:06

Rafaël Dera

409 ● 3 ● 10

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.