Import-ant Decisions

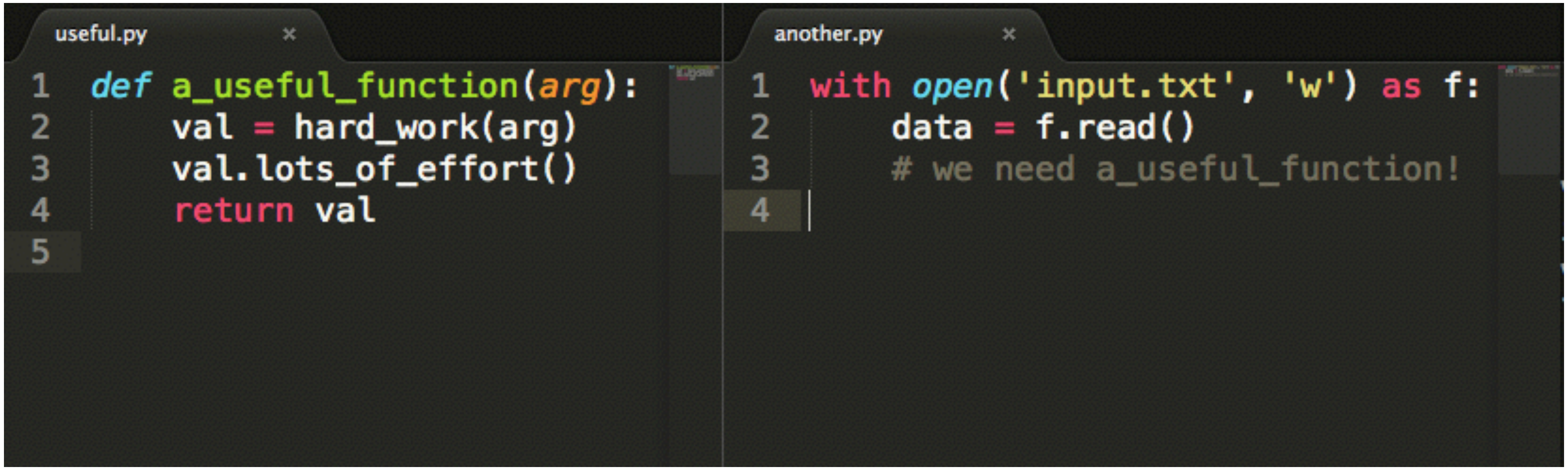
Allison Kaptur
PyCon 2014
allison.kaptur@gmail.com
github.com/akaptur
@akaptur

Hacker School

THIS IS A COUP

import

The problem



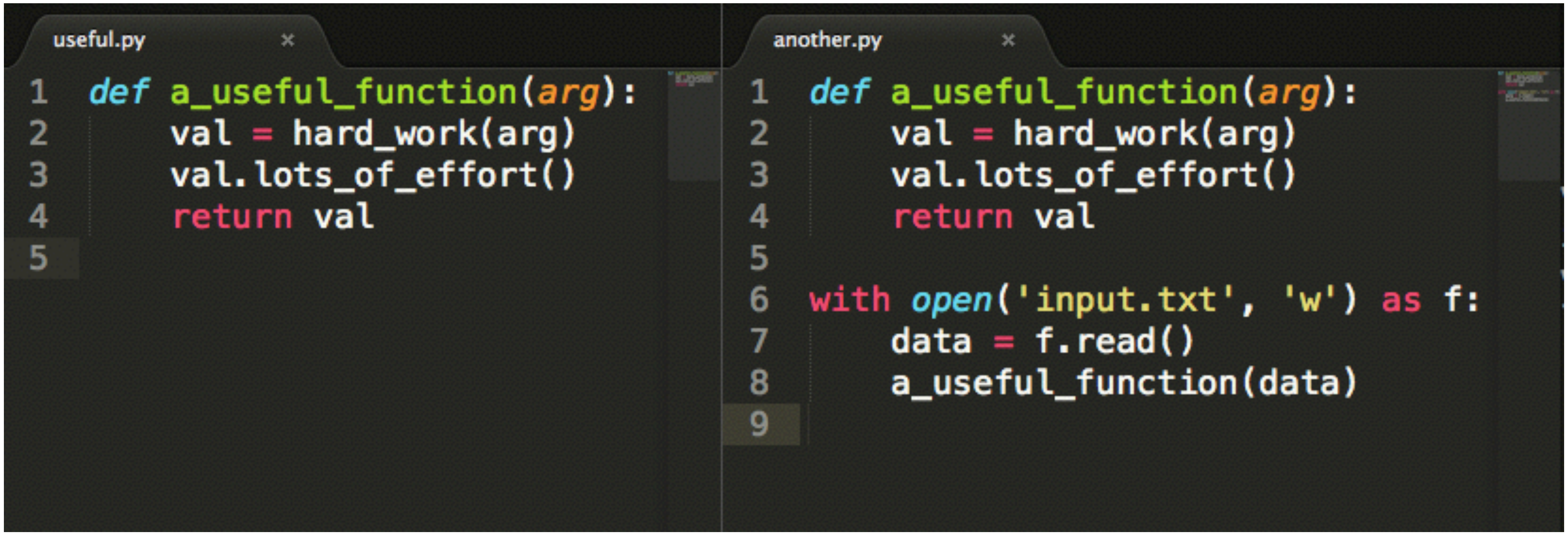
useful.py

```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

another.py

```python
with open('input.txt', 'w') as f:
    data = f.read()
    # we need a_useful_function!

```

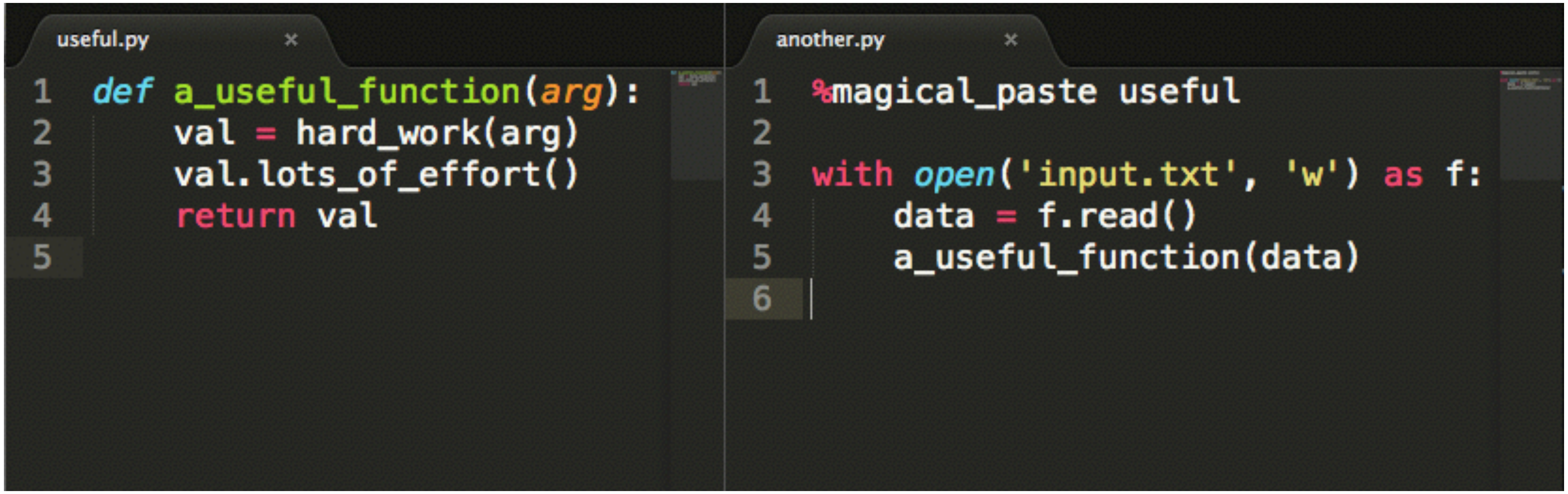
Copy & paste



```
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val
```

```
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

with open('input.txt', 'w') as f:
    data = f.read()
    a_useful_function(data)
```

A magical copy & paste


useful.py

```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

another.py

```python
%magical_paste useful

with open('input.txt', 'w') as f:
    data = f.read()
    a_useful_function(data)

```

A magical copy & paste

Implemented outside of Python
(e.g. a bash script):

1. Take your .py file
2. Look for “%magical_paste useful”
3. Find the file useful.py
4. Replace the magical_paste line with the contents of useful.py

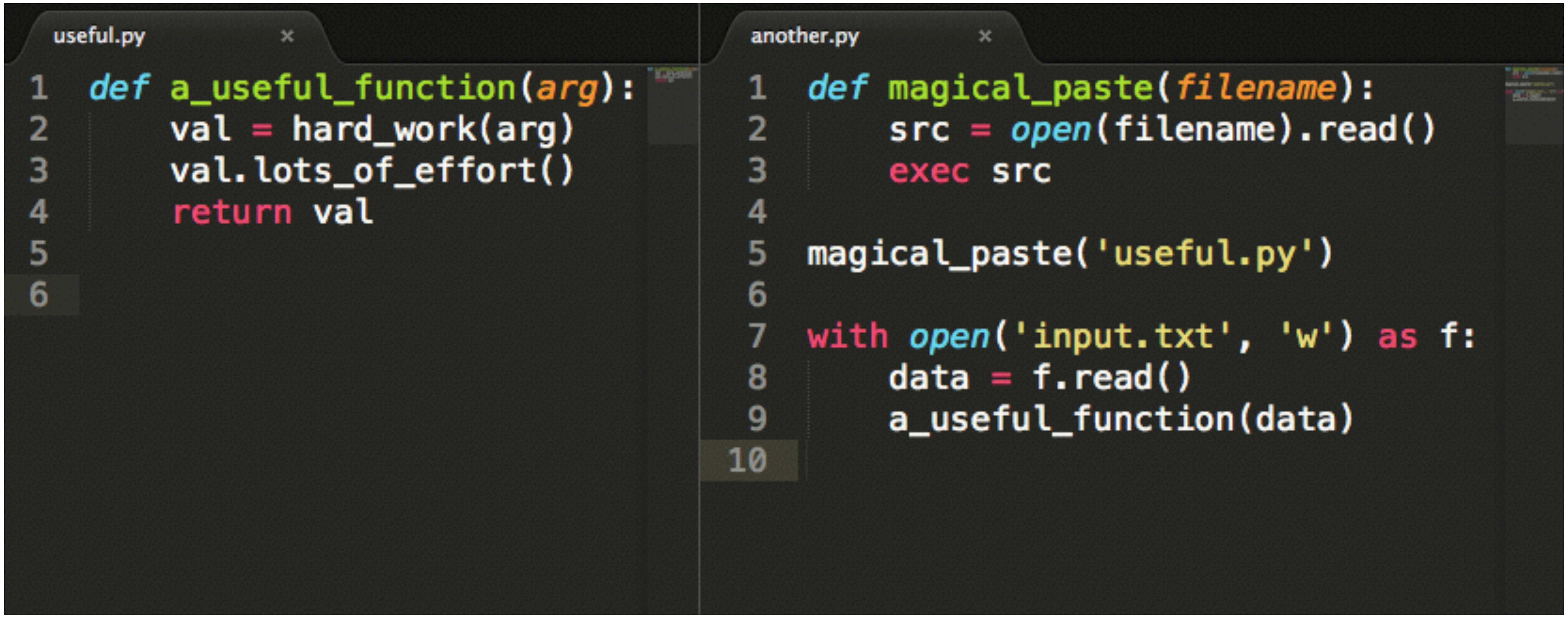
We have problems

1. Static
2. Name collisions
3. Executes multiple times
4. Rigid

We have problems

1. Static
2. Name collisions
3. Executes multiple times
4. Rigid

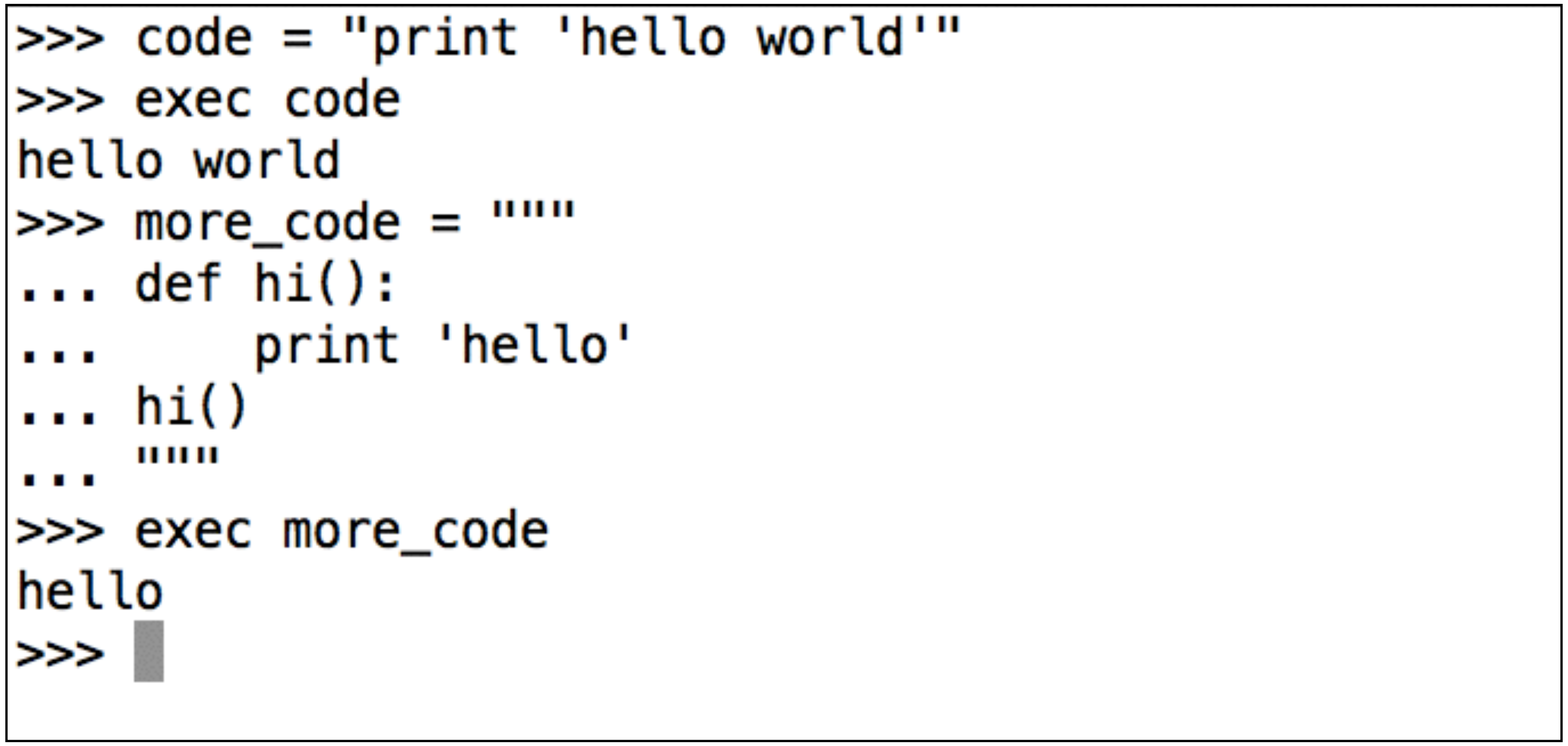
Call a function



```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

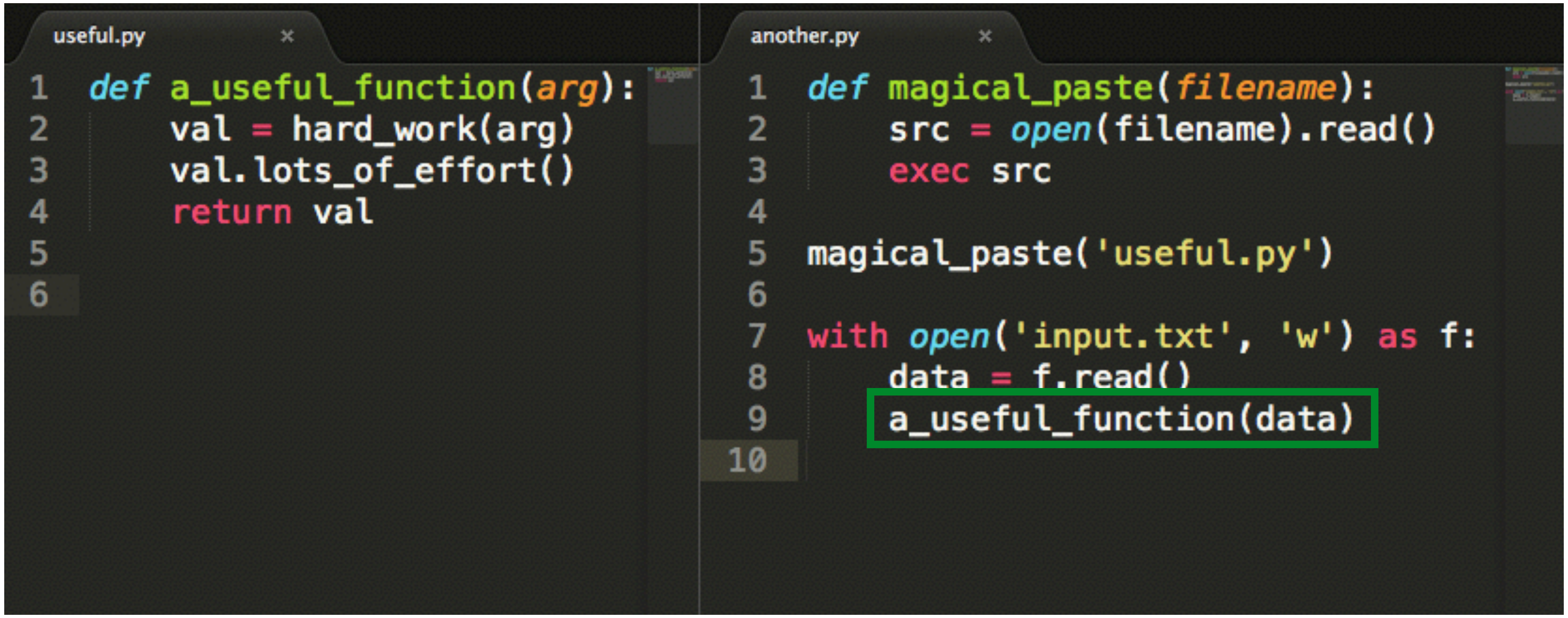
```python
def magical_paste(filename):
    src = open(filename).read()
    exec src

magical_paste('useful.py')

with open('input.txt', 'w') as f:
    data = f.read()
    a_useful_function(data)

```

exec



```
>>> code = "print 'hello world'"
>>> exec code
hello world
>>> more_code = """
... def hi():
...     print 'hello'
... hi()
... """
>>> exec more_code
hello
>>> 
```

Call a function



useful.py

```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

another.py

```python
def magical_paste(filename):
    src = open(filename).read()
    exec src

magical_paste('useful.py')

with open('input.txt', 'w') as f:
    data = f.read()
    a_useful_function(data)

```

We have problems

1. Static: use Python at run-time
2. Name collisions
3. Executes multiple times
4. Rigid

We have problems

1. Static: use Python at run-time
2. Name collisions
3. Executes multiple times
4. Rigid

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Solution: namespaces

Montréal

Montréal

Images:
http://en.wikipedia.org/wiki/Montr%C3%A9al,_Ard%C3%A8che
http://en.wikipedia.org/wiki/Montreal

Solution: namespaces

Montréal, France

Montréal, Canada

Solution: namespaces

France.Montréal

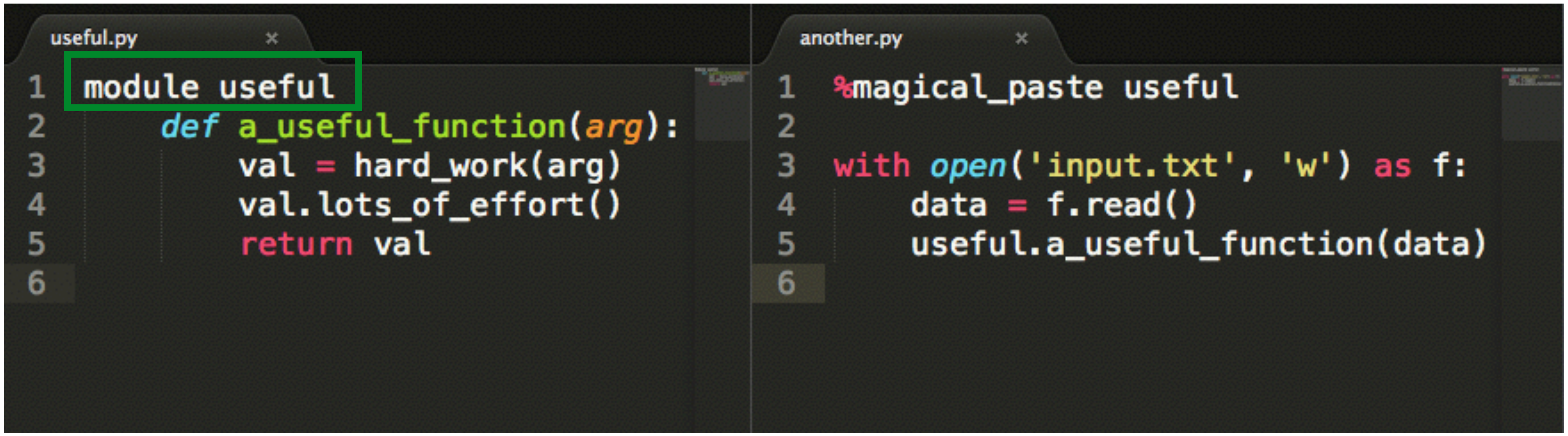
Canada.Montréal

Solution: namespaces

France

Canada

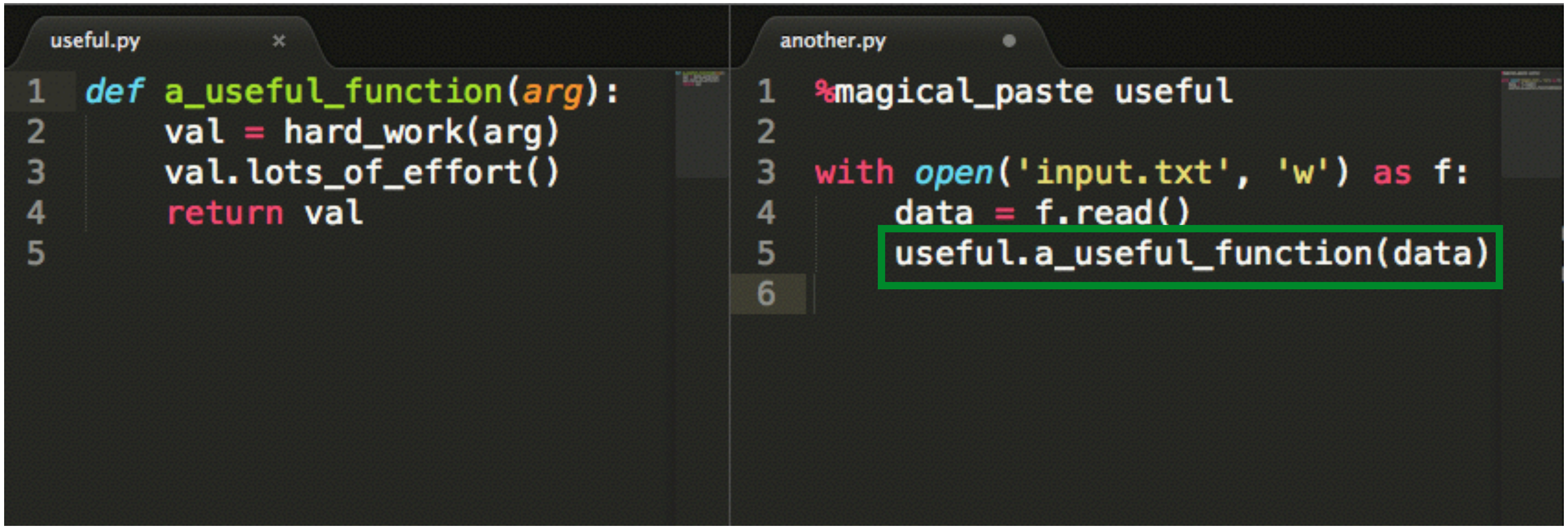
Module



```
module useful
    def a_useful_function(arg):
        val = hard_work(arg)
        val.lots_of_effort()
        return val
```

```
%magical_paste useful

with open('input.txt', 'w') as f:
    data = f.read()
    useful.a_useful_function(data)
```

Module



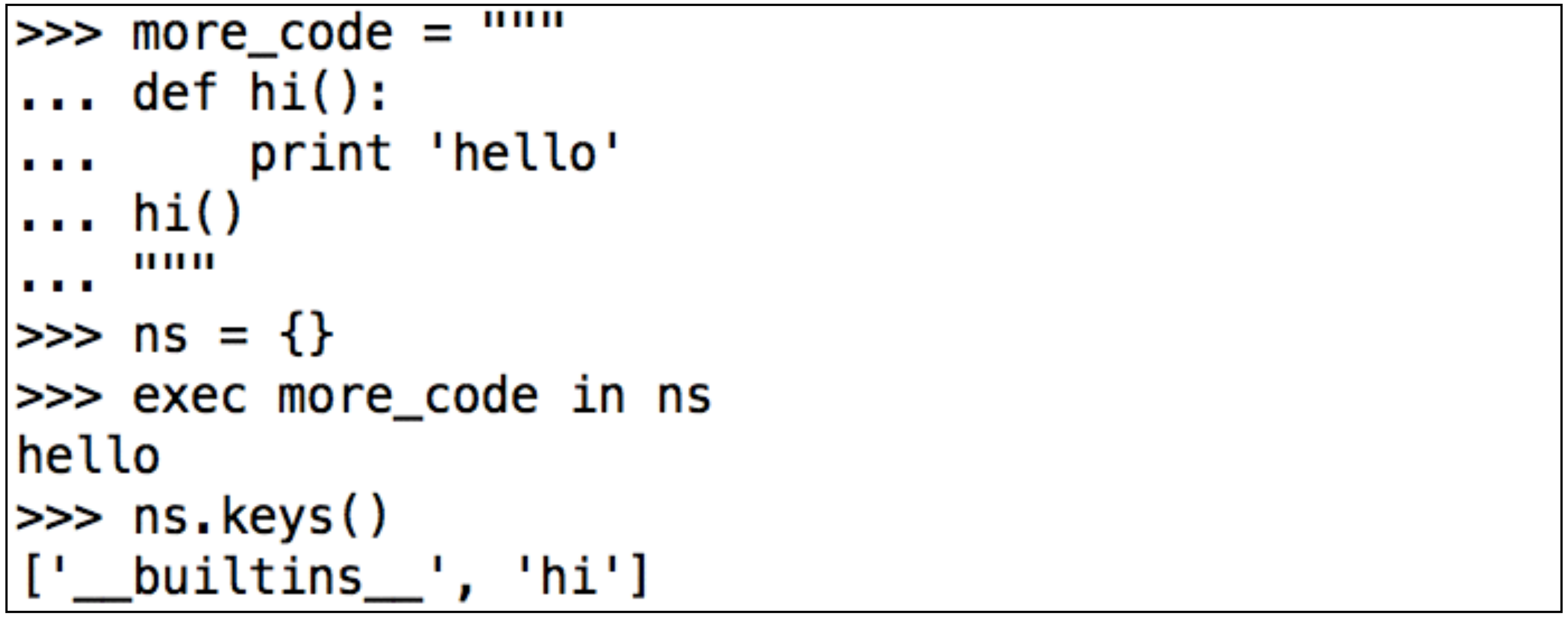
useful.py

```
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val
```

another.py

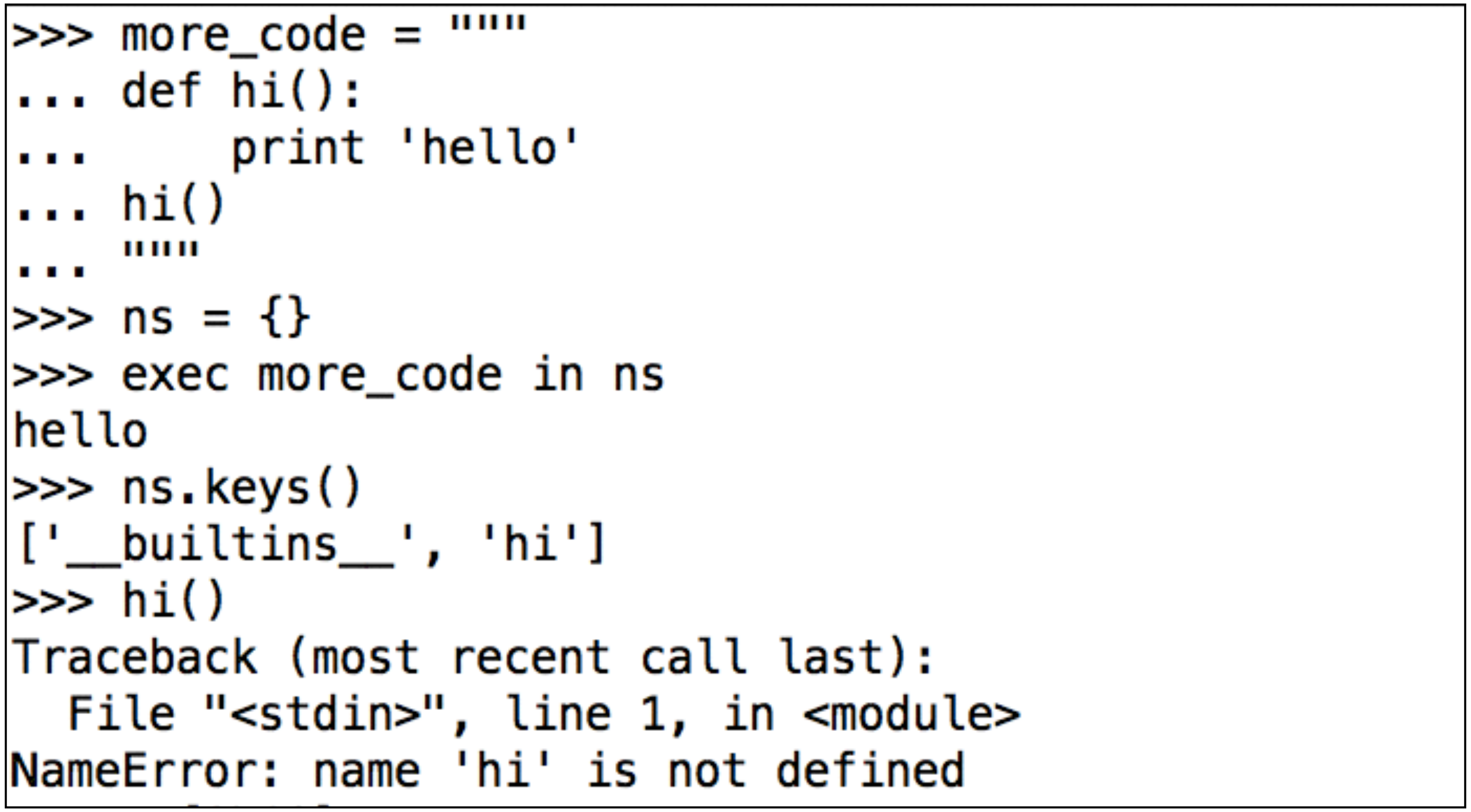
```
%magical_paste useful

with open('input.txt', 'w') as f:
    data = f.read()
    useful.a_useful_function(data)
```

exec with a namespace



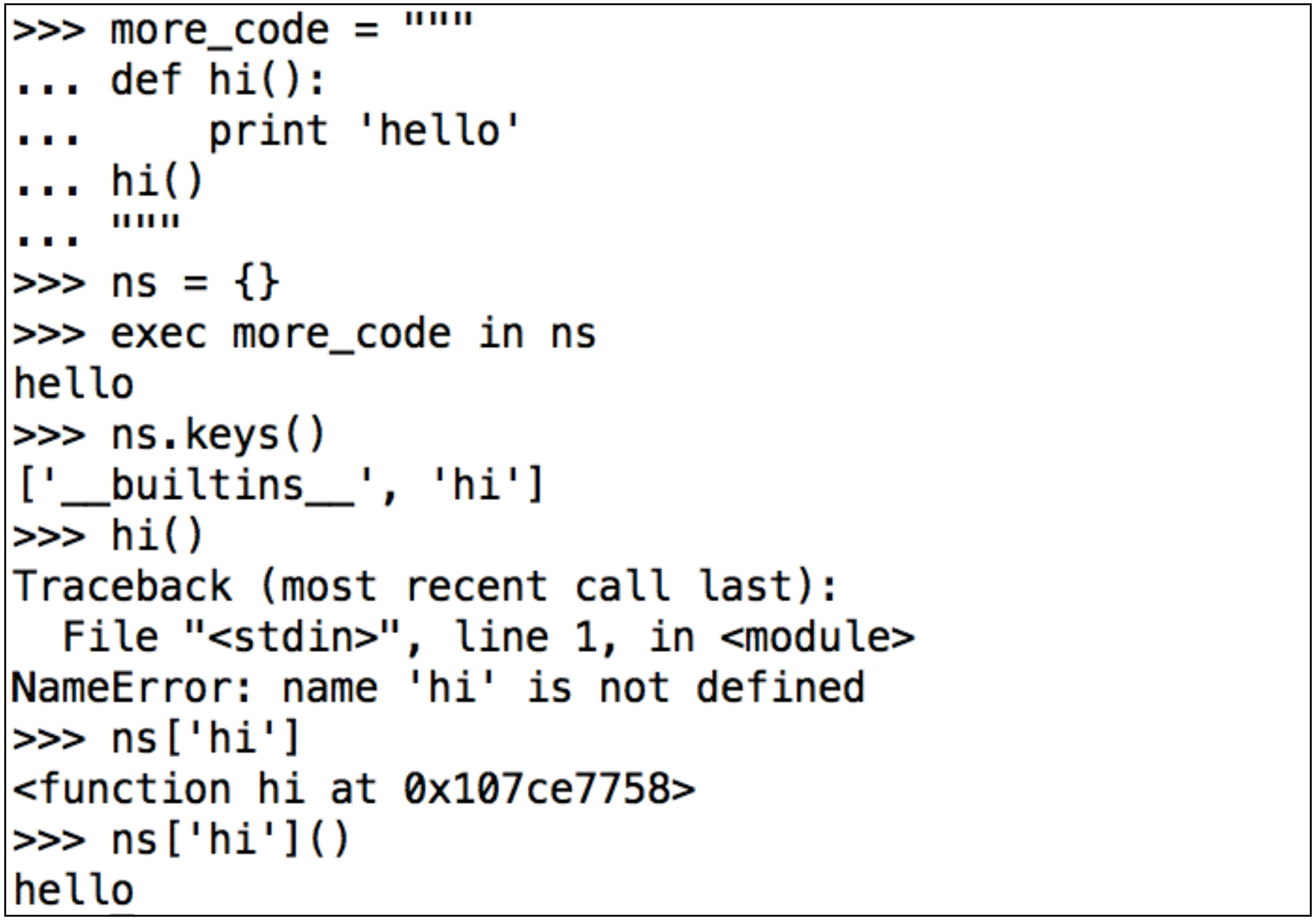
```
>>> more_code = """
... def hi():
...     print 'hello'
... hi()
... """
>>> ns = {}
>>> exec more_code in ns
hello
>>> ns.keys()
['__builtins__', 'hi']
```

exec with a namespace



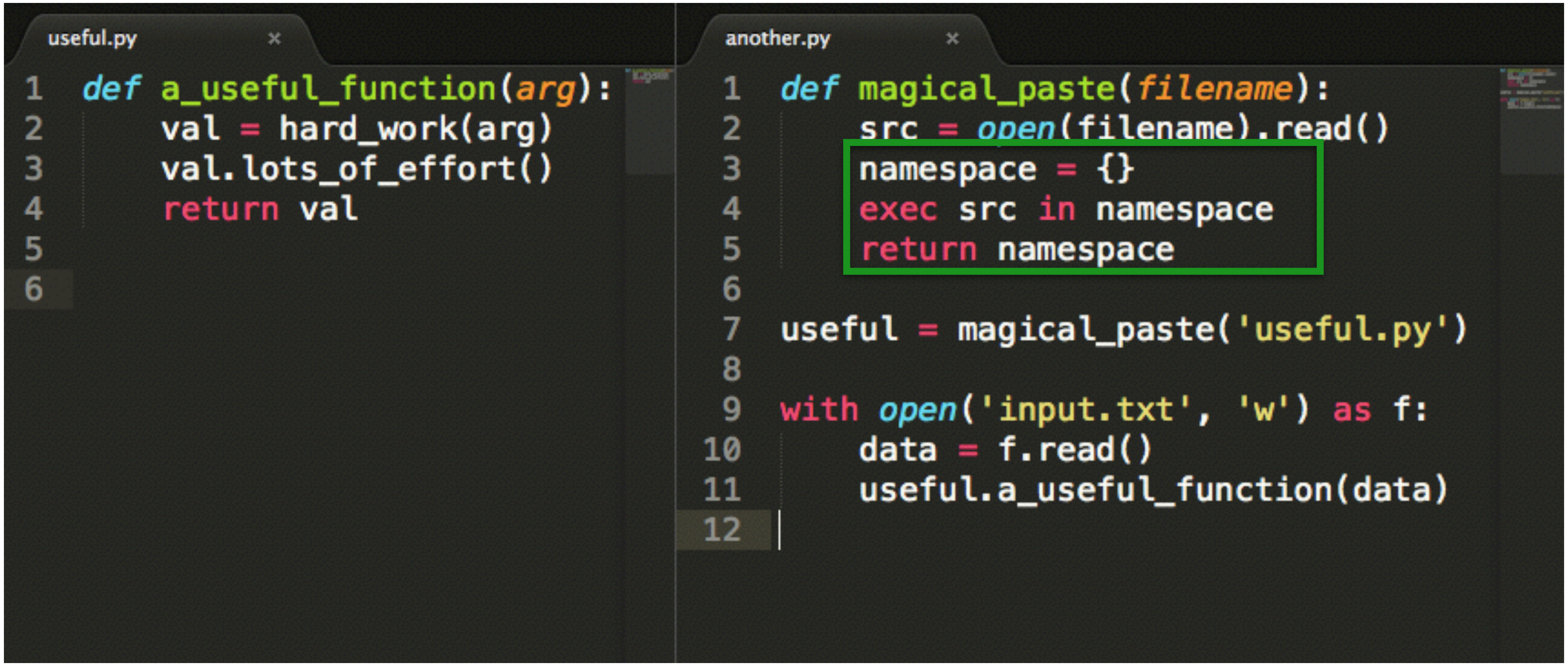
```
>>> more_code = """
... def hi():
...     print 'hello'
... hi()
... """
>>> ns = {}
>>> exec more_code in ns
hello
>>> ns.keys()
['__builtins__', 'hi']
>>> hi()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hi' is not defined
```

exec with a namespace



```
>>> more_code = """
... def hi():
...     print 'hello'
... hi()
... """
>>> ns = {}
>>> exec more_code in ns
hello
>>> ns.keys()
['__builtins__', 'hi']
>>> hi()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hi' is not defined
>>> ns['hi']
<function hi at 0x107ce7758>
>>> ns['hi']()
hello
```

Adding a namespace

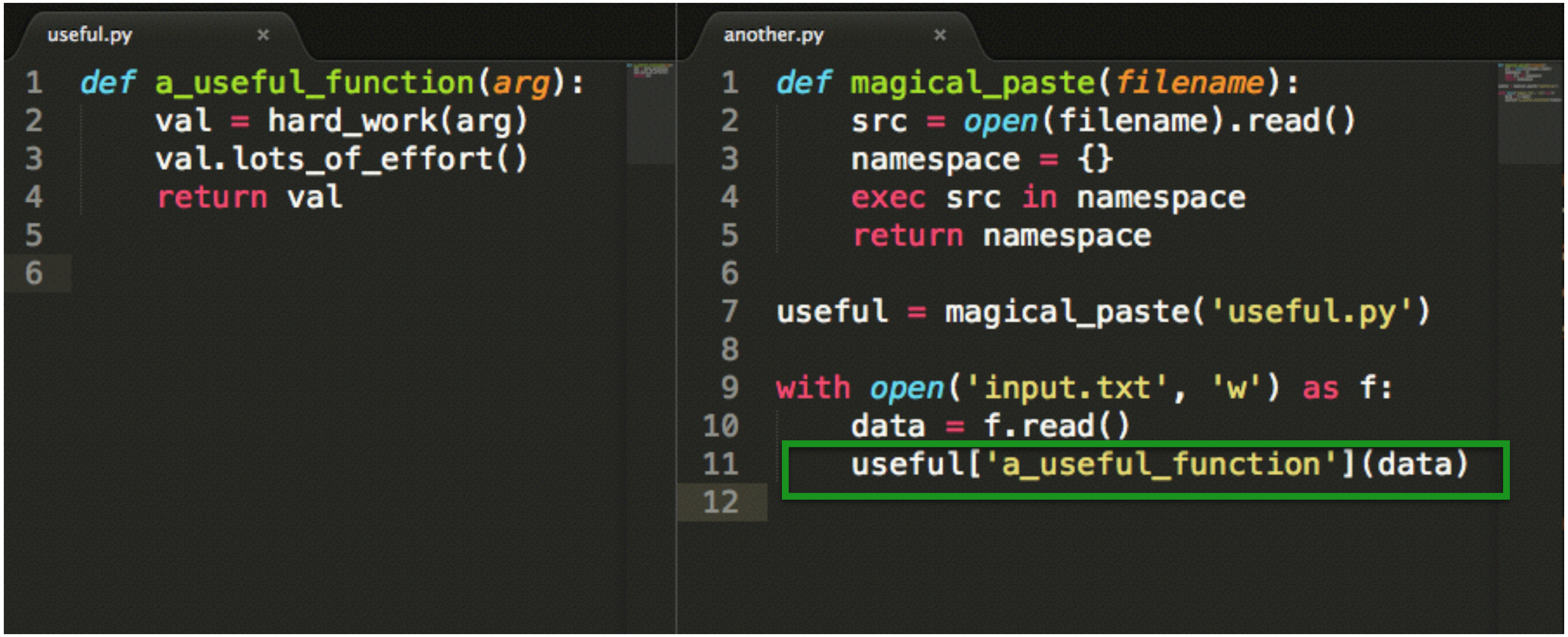

useful.py

```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

another.py

```python
def magical_paste(filename):
    src = open(filename).read()
    namespace = {}
    exec src in namespace
    return namespace

useful = magical_paste('useful.py')

with open('input.txt', 'w') as f:
    data = f.read()
    useful.a_useful_function(data)

```

Adding a namespace



useful.py

```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

another.py

```python
def magical_paste(filename):
    src = open(filename).read()
    namespace = {}
    exec src in namespace
    return namespace

useful = magical_paste('useful.py')

with open('input.txt', 'w') as f:
    data = f.read()
    useful['a_useful_function'](data)

```

(This is valid code)

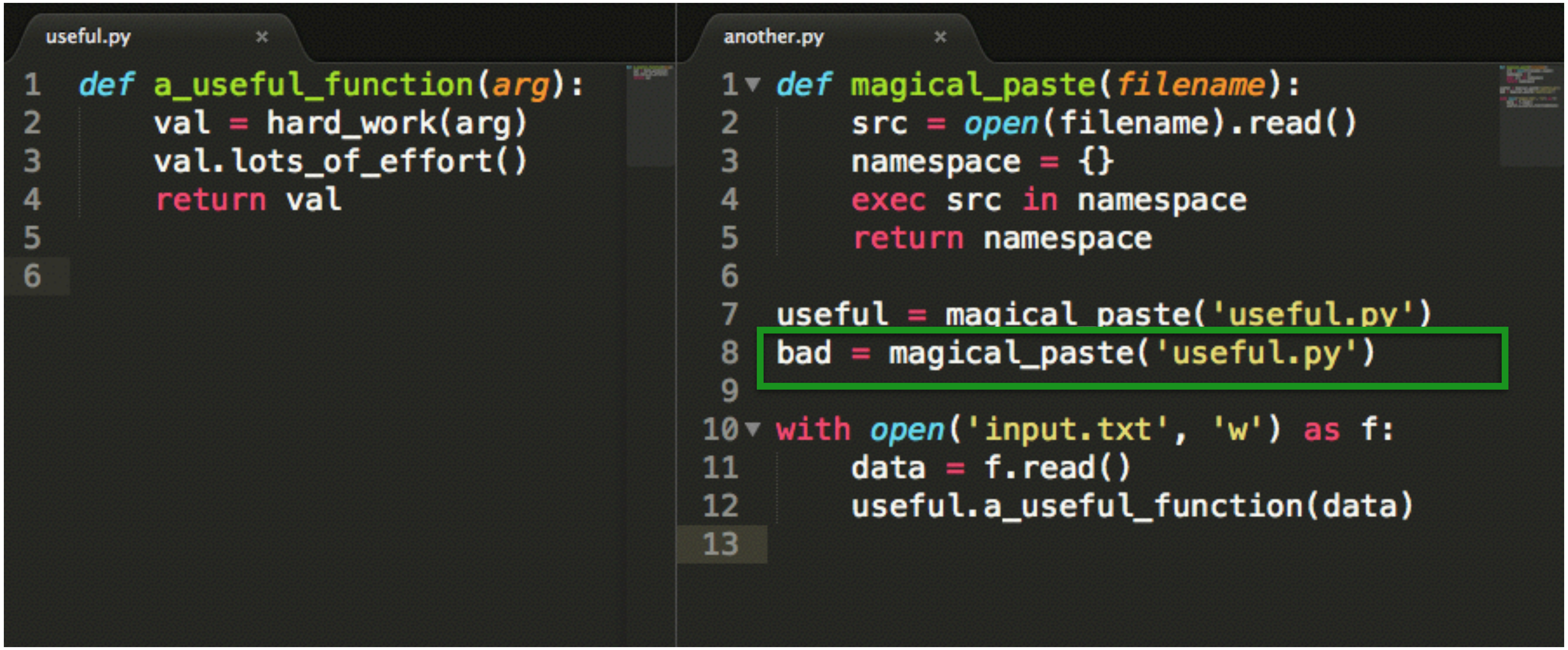
We have problems

1. Static: use Python at run-time
2. Name collisions: use namespaces and modules
3. Executes multiple times
4. Rigid

We have problems

1. Static: use Python at run-time
2. Name collisions: use namespaces and modules
3. Executes multiple times
4. Rigid

Executes multiple times



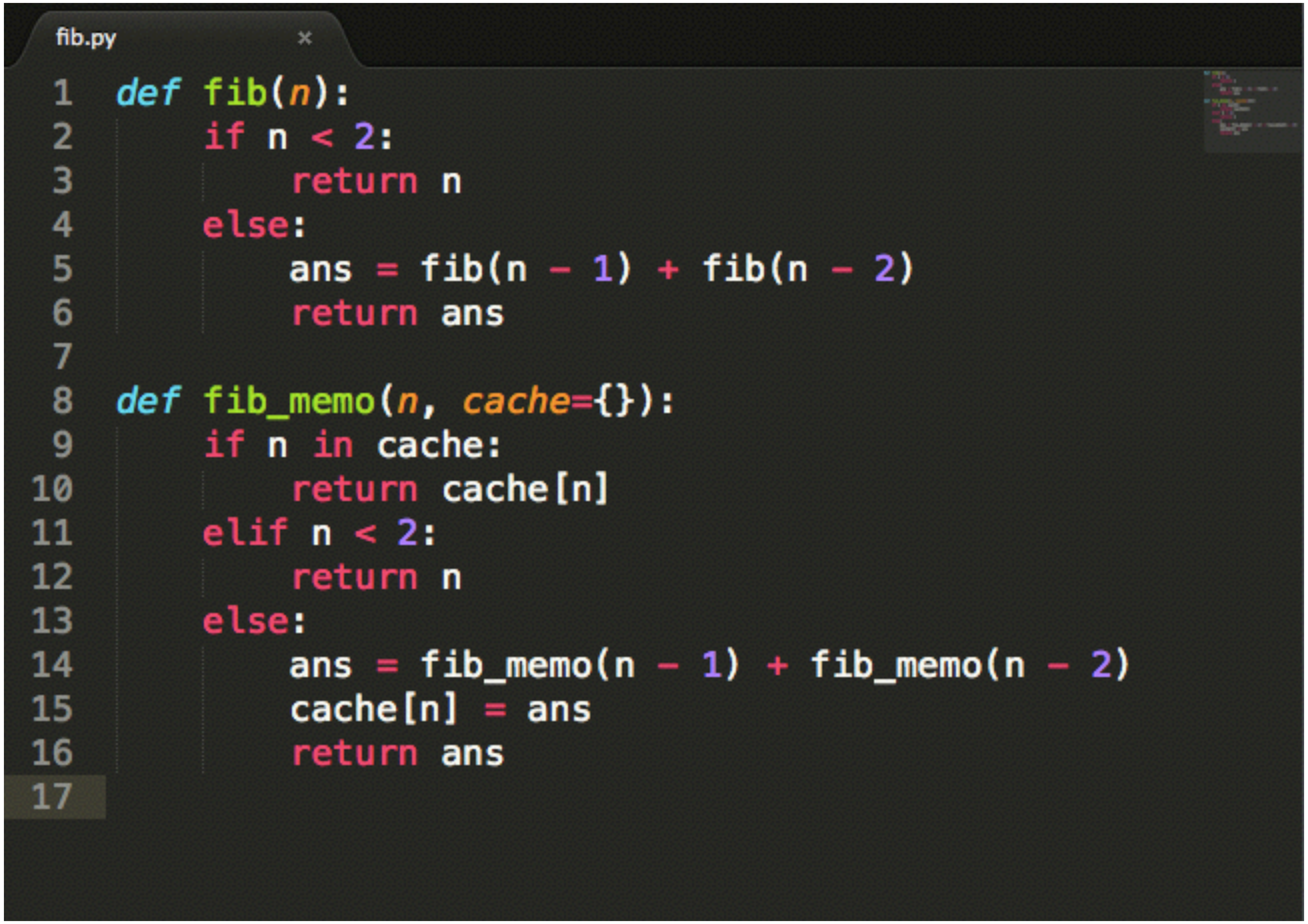
useful.py

```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

another.py

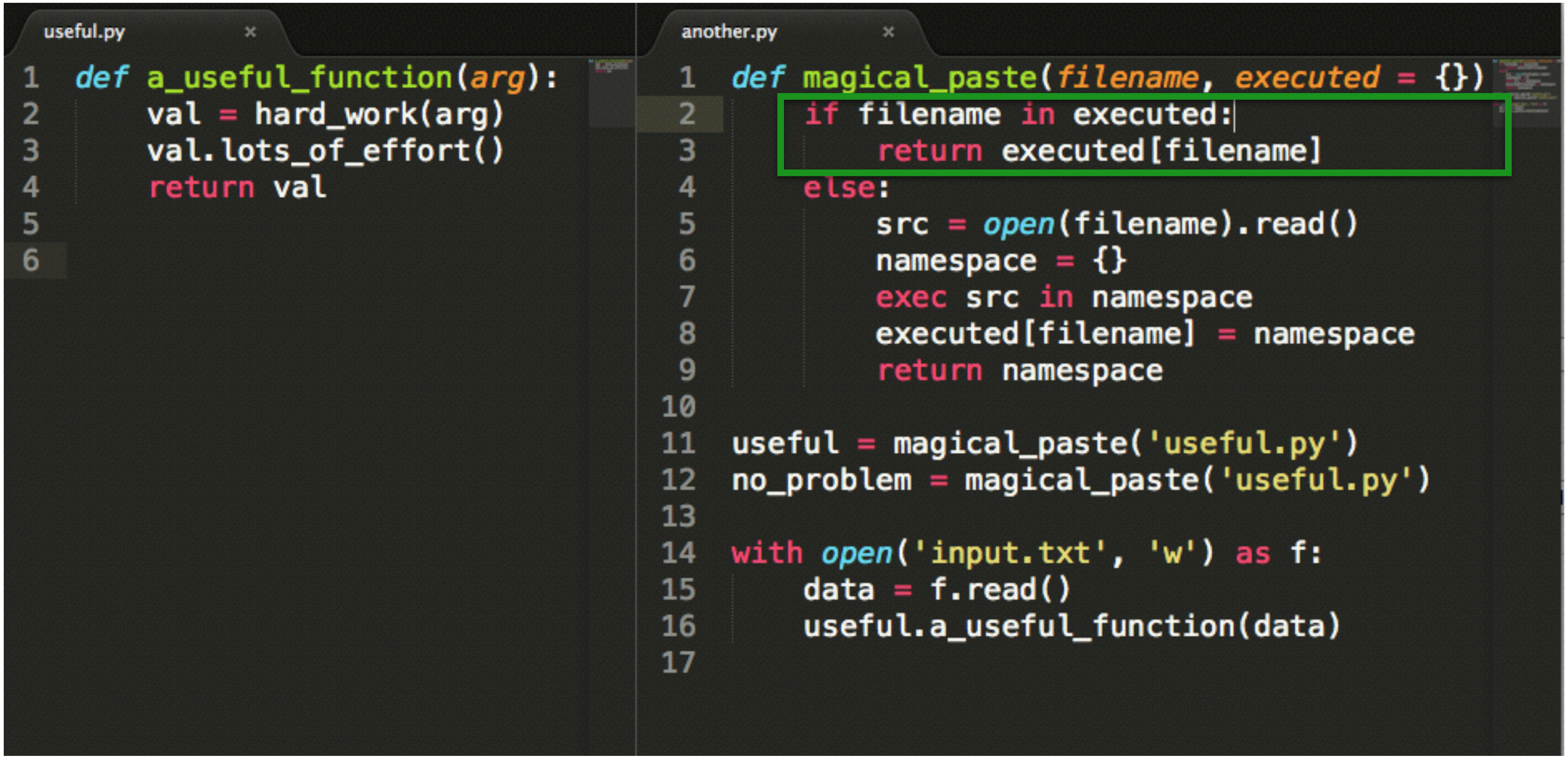
```python
def magical_paste(filename):
    src = open(filename).read()
    namespace = {}
    exec src in namespace
    return namespace

useful = magical_paste('useful.py')
bad = magical_paste('useful.py')

with open('input.txt', 'w') as f:
    data = f.read()
    useful.a_useful_function(data)

```

Memoize



```python
def fib(n):
    if n < 2:
        return n
    else:
        ans = fib(n - 1) + fib(n - 2)
        return ans

def fib_memo(n, cache={}):
    if n in cache:
        return cache[n]
    elif n < 2:
        return n
    else:
        ans = fib_memo(n - 1) + fib_memo(n - 2)
        cache[n] = ans
        return ans
```

Memoize



```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

```python
def magical_paste(filename, executed = {}):
    if filename in executed:
        return executed[filename]
    else:
        src = open(filename).read()
        namespace = {}
        exec src in namespace
        executed[filename] = namespace
        return namespace

useful = magical_paste('useful.py')
no_problem = magical_paste('useful.py')

with open('input.txt', 'w') as f:
    data = f.read()
    useful.a_useful_function(data)

```

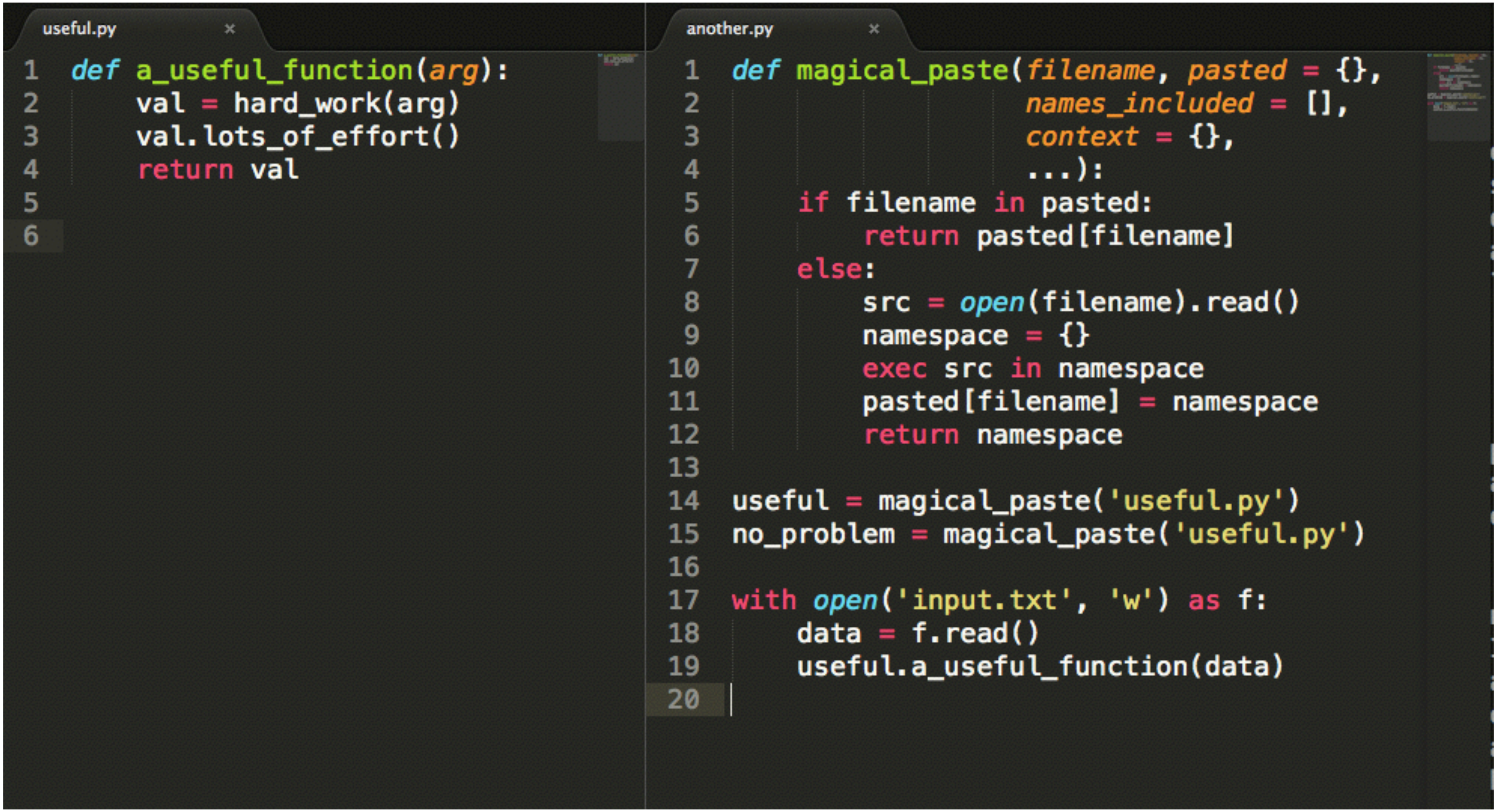
We have problems

1. Static: use Python at run-time
2. Name collisions: use namespaces and modules
3. Executes multiple times: memoize
4. Rigid

We have problems

1. Static: use Python at run-time
2. Name collisions: use namespaces and modules
3. Executes multiple times: memoize
4. Rigid

Options?



useful.py

```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val

```

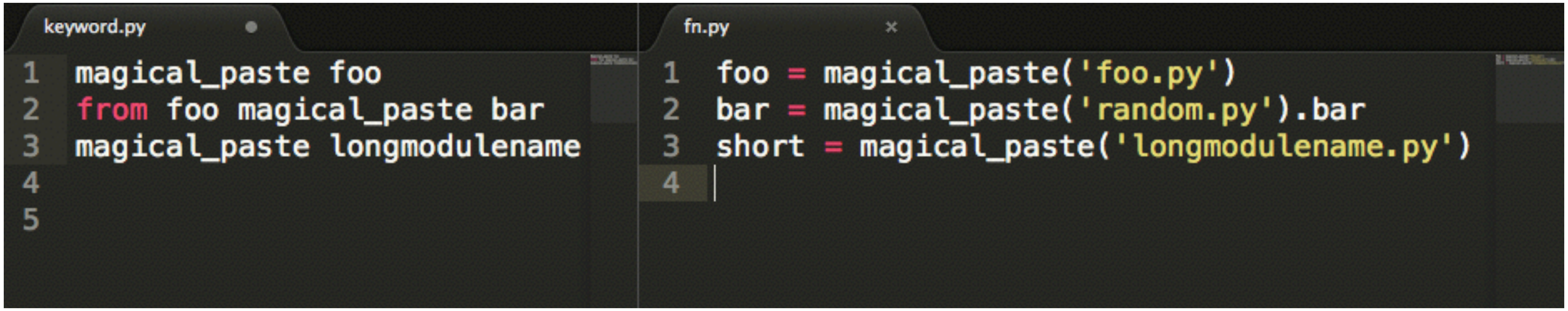
another.py

```python
def magical_paste(filename, pasted = {},
                            names_included = [],
                            context = {},
                            ...):
    if filename in pasted:
        return pasted[filename]
    else:
        src = open(filename).read()
        namespace = {}
        exec src in namespace
        pasted[filename] = namespace
        return namespace

useful = magical_paste('useful.py')
no_problem = magical_paste('useful.py')

with open('input.txt', 'w') as f:
    data = f.read()
    useful.a_useful_function(data)

```

We have problems

1. Static: use Python at run-time
2. Name collisions: use namespaces and modules
3. Executes multiple times: memoize
4. ~~Rigid~~ Ugly

Keywords: syntactic sugar



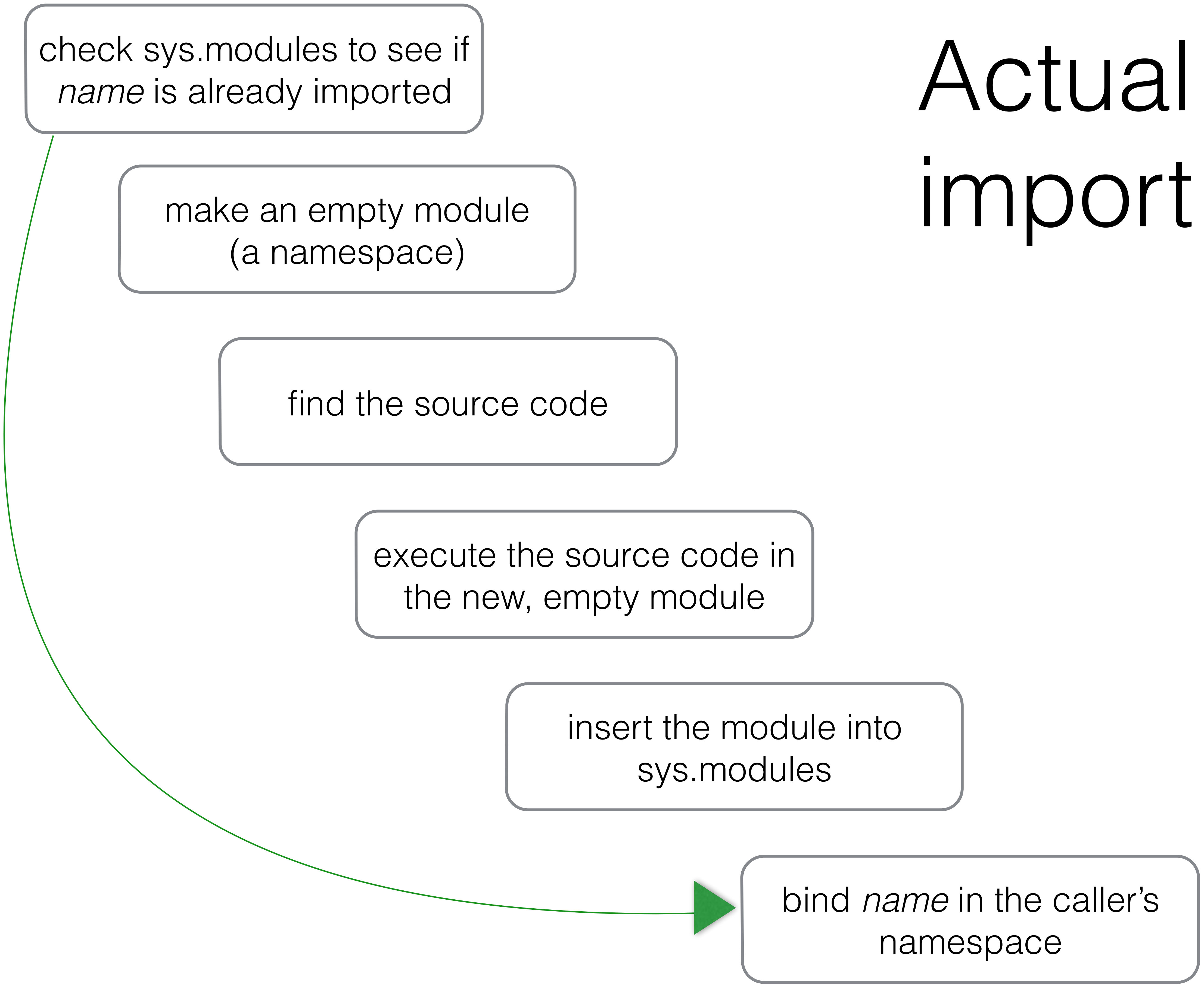
```
magical_paste foo
from foo magical_paste bar
magical_paste longmodulename
```

```
foo = magical_paste('foo.py')
bar = magical_paste('random.py').bar
short = magical_paste('longmodulename.py')
```

We have problems

1. Static: use Python at run-time
2. Name collisions: use namespaces and modules
3. Executes multiple times: memoize
4. Rigid: use keywords for flexible syntax

Actual import

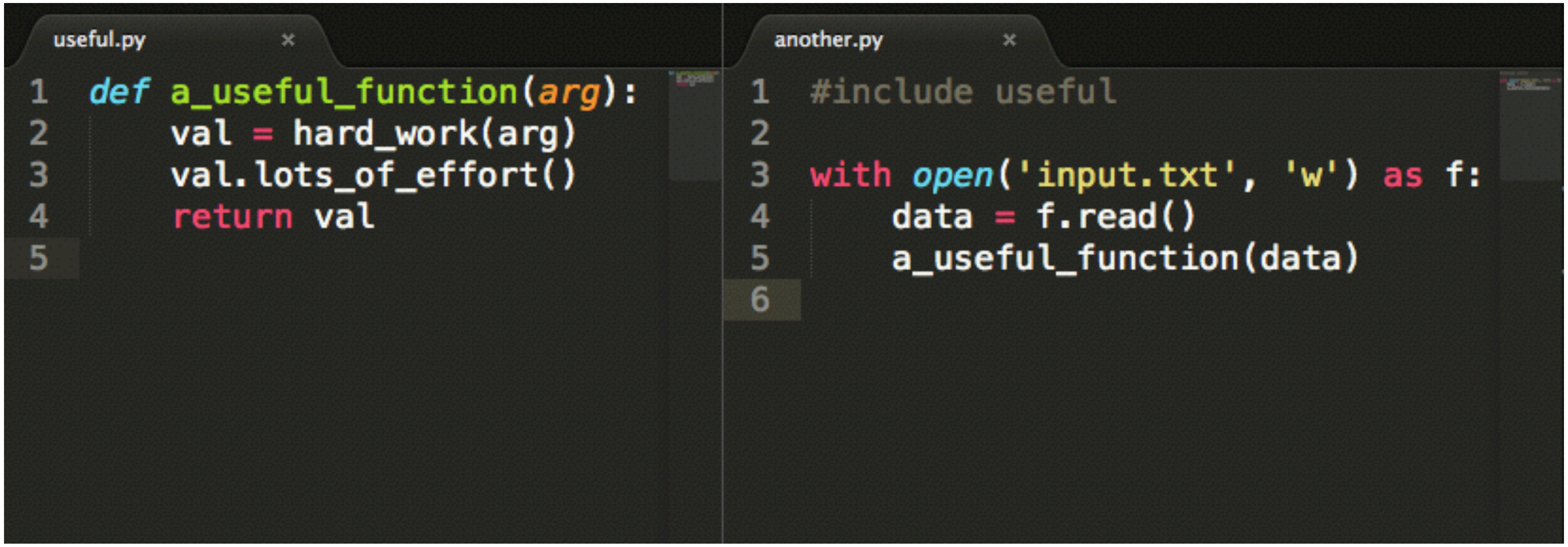

check sys.modules to see if *name* is already imported
make an empty module (a namespace)
find the source code
execute the source code in the new, empty module
insert the module into sys.modules
bind *name* in the caller's namespace

(Some) things we haven't done

1. Created real modules
2. Created packages
3. Worried about loaders
4. Worried about error handling

Other solutions

#include in C


useful.py

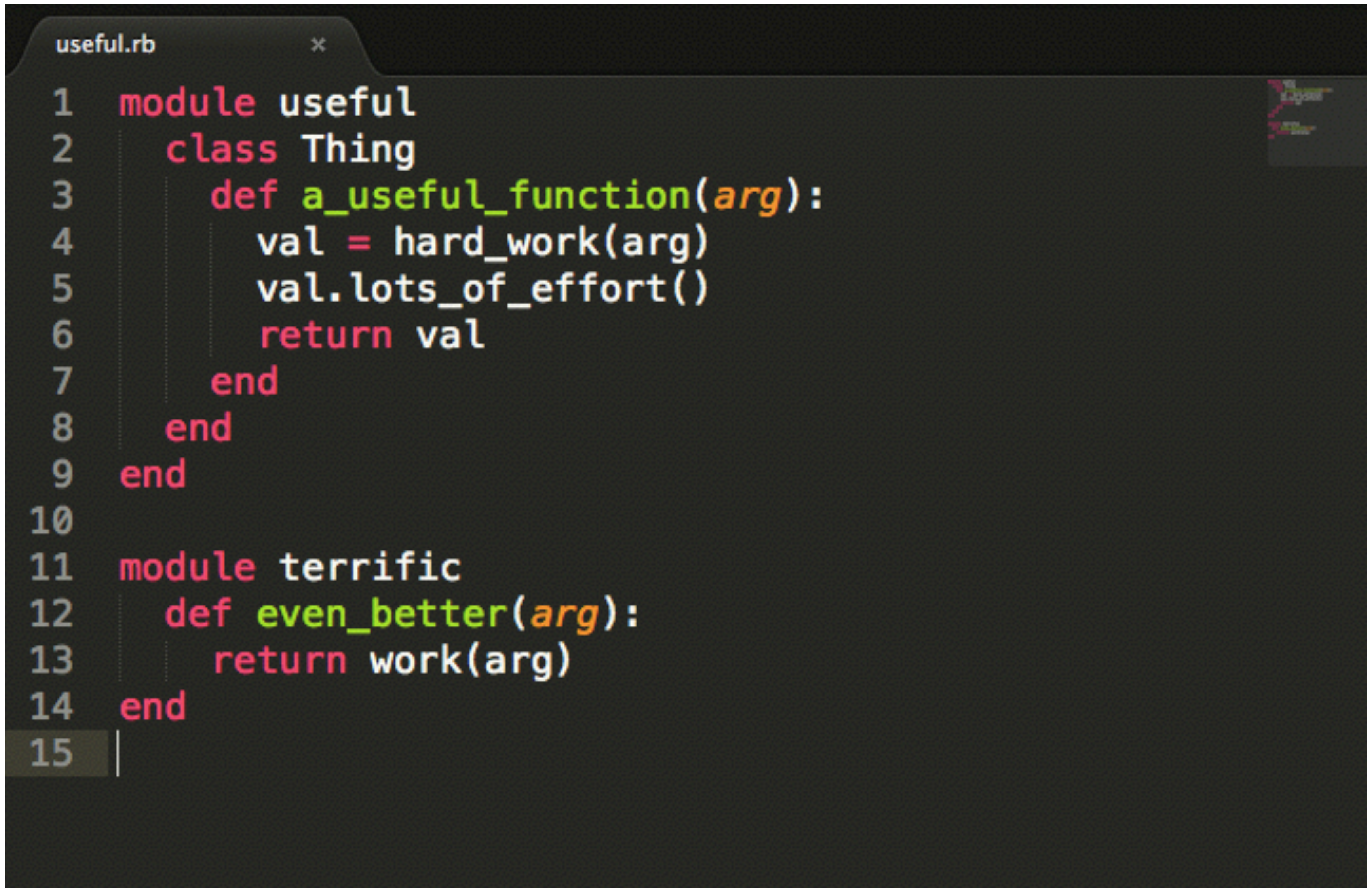
```python
def a_useful_function(arg):
    val = hard_work(arg)
    val.lots_of_effort()
    return val
```

another.py

```python
#include useful

with open('input.txt', 'w') as f:
    data = f.read()
    a_useful_function(data)
```

“magical copy & paste”

Ruby modules



```
module useful
  class Thing
    def a_useful_function(arg):
      val = hard_work(arg)
      val.lots_of_effort()
      return val
    end
  end
end

module terrific
  def even_better(arg):
    return work(arg)
end
```

Modula

“A module is a set of procedures, data types, and variables, where the programmer has precise control over the names that are imported from and exported to the environment.”

- N. Wirth, “Modula: A Language for Modular Multiprogramming” (1976)

There are these two young fish swimming along and they happen to meet an older fish swimming the other way, who nods at them and says "Morning, boys. How's the water?" And the two young fish swim on for a bit, and then eventually one of them looks over at the other and goes "What the hell is water?”

- David Foster Wallace, 2005

Invisible blind spots of Python programmers when it comes to ‘import’

Allison Kaptur
PyCon 2014
github.com/akaptur
@akaptur

Hacker School

Questions

Allison Kaptur
PyCon 2014
github.com/akaptur
@akaptur

Hacker School