

# Import-ant Decisions

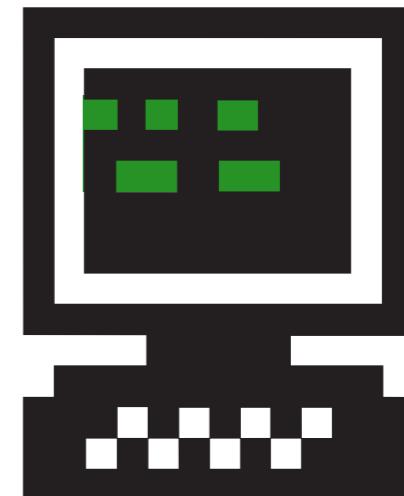
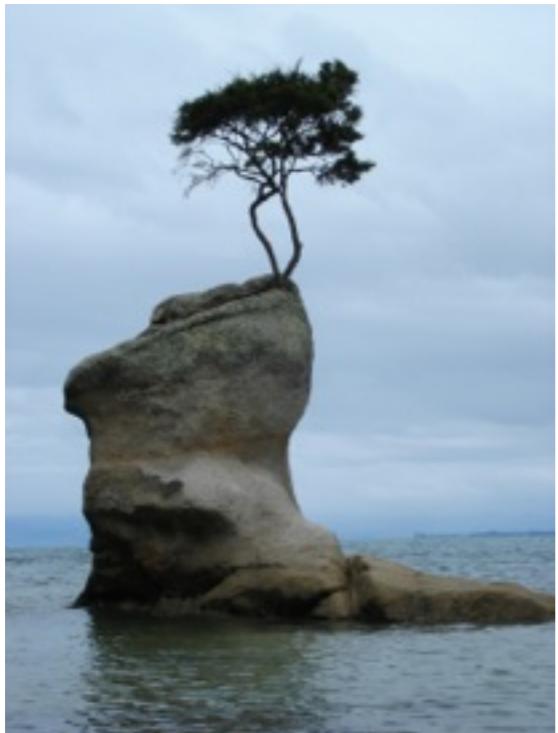
Allison Kaptur

PyCon 2014

[allison.kaptur@gmail.com](mailto:allison.kaptur@gmail.com)

[github.com/akaptur](https://github.com/akaptur)

@akaptur



**Hacker** School

THIS IS A  
COUP

import

# The problem

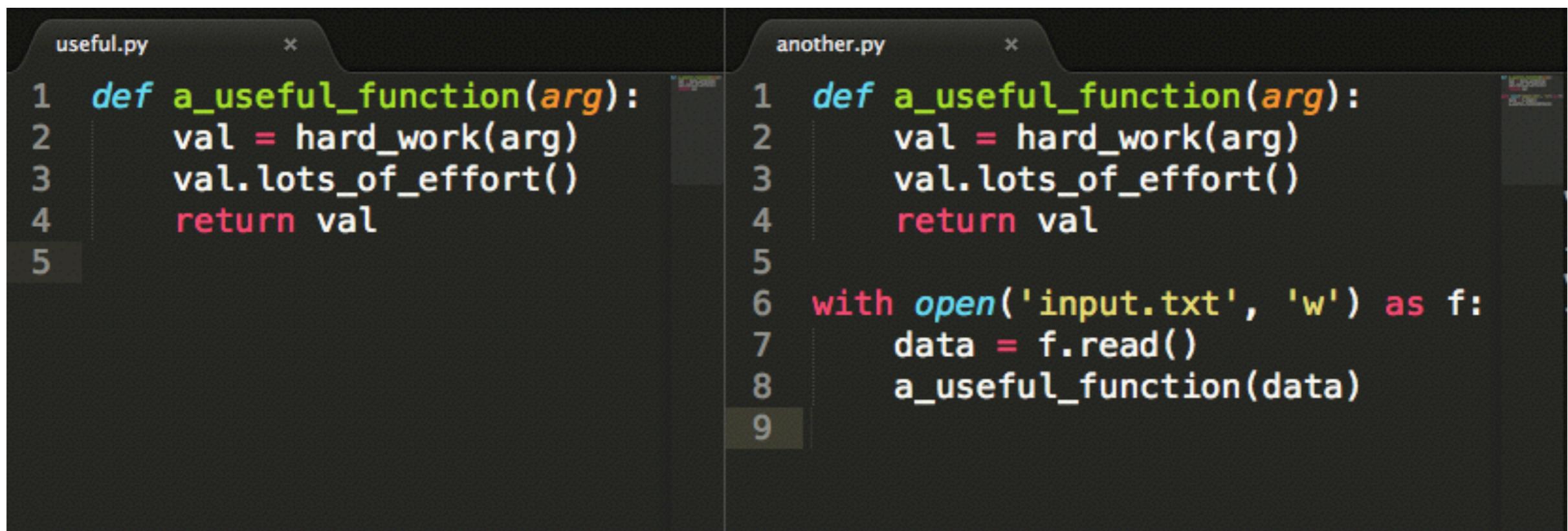
The image shows a code editor with two tabs open. The left tab is titled "useful.py" and contains the following Python code:

```
useful.py
1 def a_useful_function(arg):
2     val = hard_work(arg)
3     val.lots_of_effort()
4     return val
5
```

The right tab is titled "another.py" and contains the following Python code:

```
another.py
1 with open('input.txt', 'w') as f:
2     data = f.read()
3     # we need a_useful_function!
4
```

# Copy & paste



```
useful.py      *
1 def a_useful_function(arg):
2     val = hard_work(arg)
3     val.lots_of_effort()
4     return val
5

another.py      *
1 def a_useful_function(arg):
2     val = hard_work(arg)
3     val.lots_of_effort()
4     return val
5
6 with open('input.txt', 'w') as f:
7     data = f.read()
8     a_useful_function(data)
9
```

# A magical copy & paste

```
useful.py *  
1 def a_useful_function(arg):  
2     val = hard_work(arg)  
3     val.lots_of_effort()  
4     return val  
5  
another.py *  
1 %magical_paste useful  
2  
3 with open('input.txt', 'w') as f:  
4     data = f.read()  
5     a_useful_function(data)  
6
```

# 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

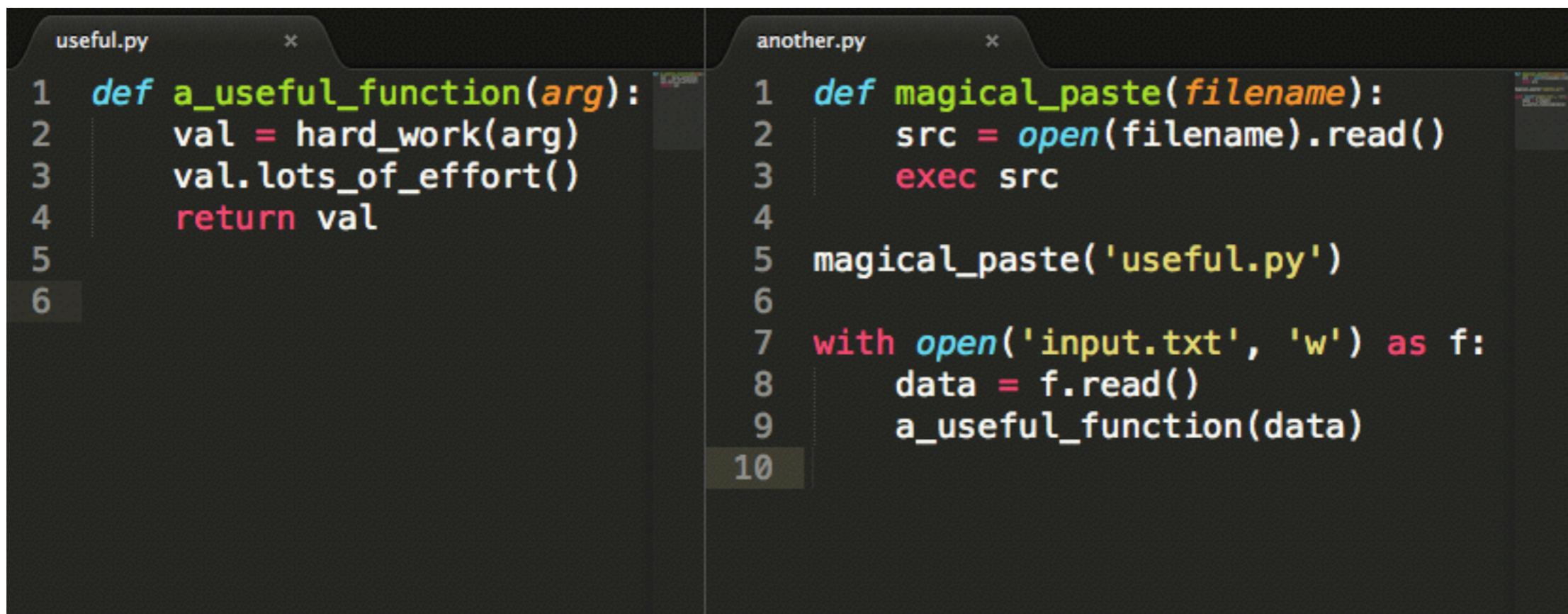
# We have problems

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



The image shows a dark-themed code editor with two tabs open: "useful.py" and "another.py".

**useful.py:**

```
1 def a_useful_function(arg):
2     val = hard_work(arg)
3     val.lots_of_effort()
4     return val
5
6
```

**another.py:**

```
1 def magical_paste(filename):
2     src = open(filename).read()
3     exec src
4
5 magical_paste('useful.py')
6
7 with open('input.txt', 'w') as f:
8     data = f.read()
9     a_useful_function(data)
10
```

# 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      another.py
1 def a_useful_function(arg):    1 def magical_paste(filename):
2     val = hard_work(arg)        2     src = open(filename).read()
3     val.lots_of_effort()       3     exec src
4     return val                4
5                               5     magical_paste('useful.py')
6                               6
7     with open('input.txt', 'w') as f: 7
8         data = f.read()           8     a_useful_function(data)
9         a_useful_function(data)   9
10                            10
```

# 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/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

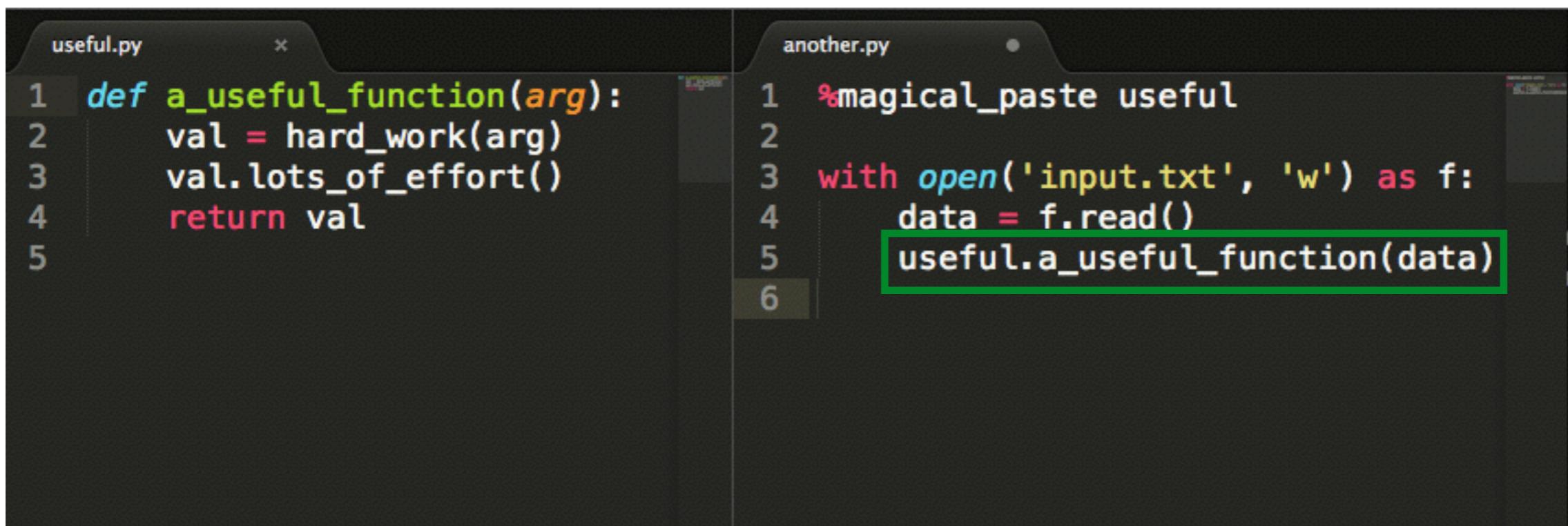
The image shows a dark-themed code editor with two tabs open. The left tab is named 'useful.py' and contains the following Python code:

```
1 module useful
2     def a_useful_function(arg):
3         val = hard_work(arg)
4         val.lots_of_effort()
5         return val
6 
```

The word 'module' in the first line is highlighted with a green rectangular selection. The right tab is named 'another.py' and contains the following Python code:

```
1 %magical_paste useful
2
3 with open('input.txt', 'w') as f:
4     data = f.read()
5     useful.a_useful_function(data)
6 
```

# Module



```
useful.py      *
1 def a_useful_function(arg):
2     val = hard_work(arg)
3     val.lots_of_effort()
4     return val
5

another.py
1 %magical_paste useful
2
3 with open('input.txt', 'w') as f:
4     data = f.read()
5     useful.a_useful_function(data)
6
```

# 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

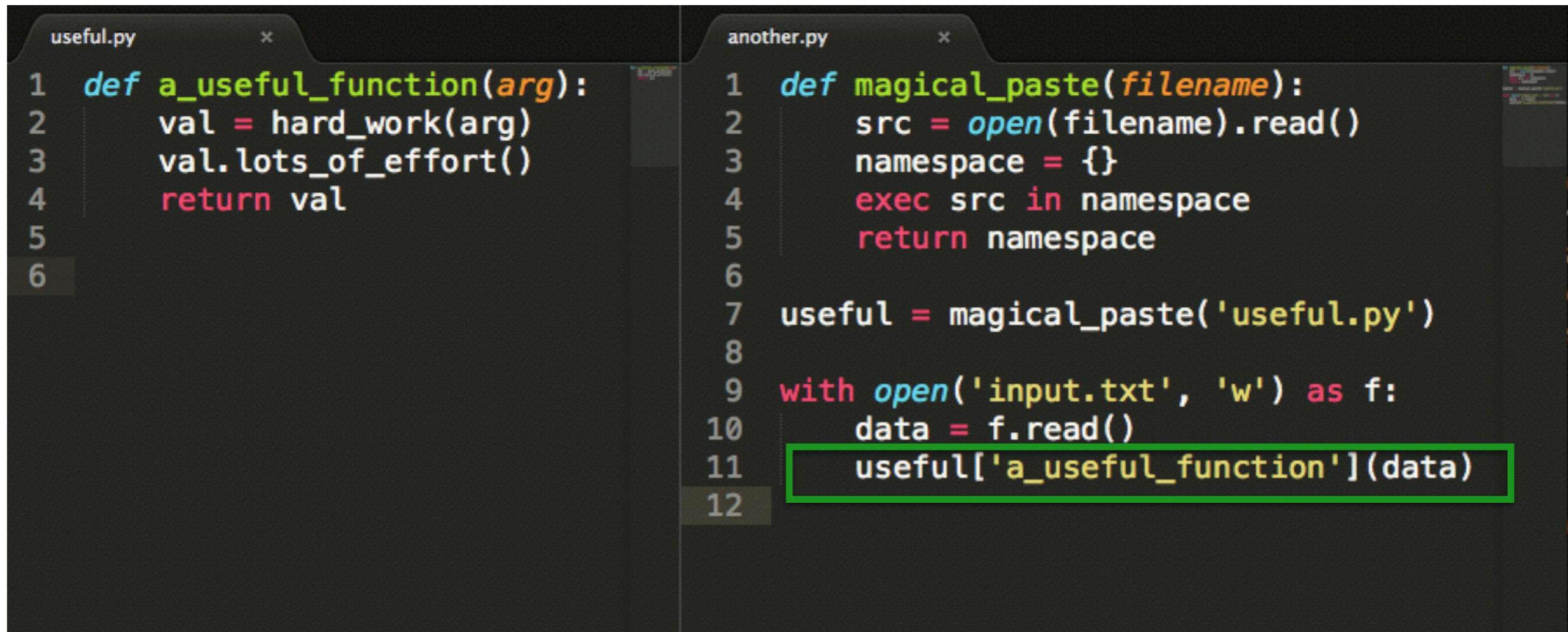
```
>>> more_code = """
... def hi():
...     print 'hello'
... hi()
...
...
>>> ns = {}
>>> exec more_code in ns
hello
>>> ns.keys()
['__builtins__', 'hi']
>>> hi()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'hi' is not defined
>>> ns['hi']
<function hi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

# Adding a namespace

```
useful.py
1 def a_useful_function(arg):
2     val = hard_work(arg)
3     val.lots_of_effort()
4     return val
5
6

another.py
1 def magical_paste(filename):
2     src = open(filename).read()
3     namespace = {}
4     exec src in namespace
5     return namespace
6
7     useful = magical_paste('useful.py')
8
9     with open('input.txt', 'w') as f:
10        data = f.read()
11        useful.a_useful_function(data)
12
```

# Adding a namespace



```
useful.py      another.py
1 def a_useful_function(arg):    1 def magical_paste(filename):
2     val = hard_work(arg)        2     src = open(filename).read()
3     val.lots_of_effort()       3     namespace = {}
4     return val                 4     exec src in namespace
5                           5     return namespace
6                           6
7     useful = magical_paste('useful.py') 7
8
9     with open('input.txt', 'w') as f: 8
10    data = f.read()                9
11    useful['a_useful_function'](data)10
12                                11
13                                12
```

(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

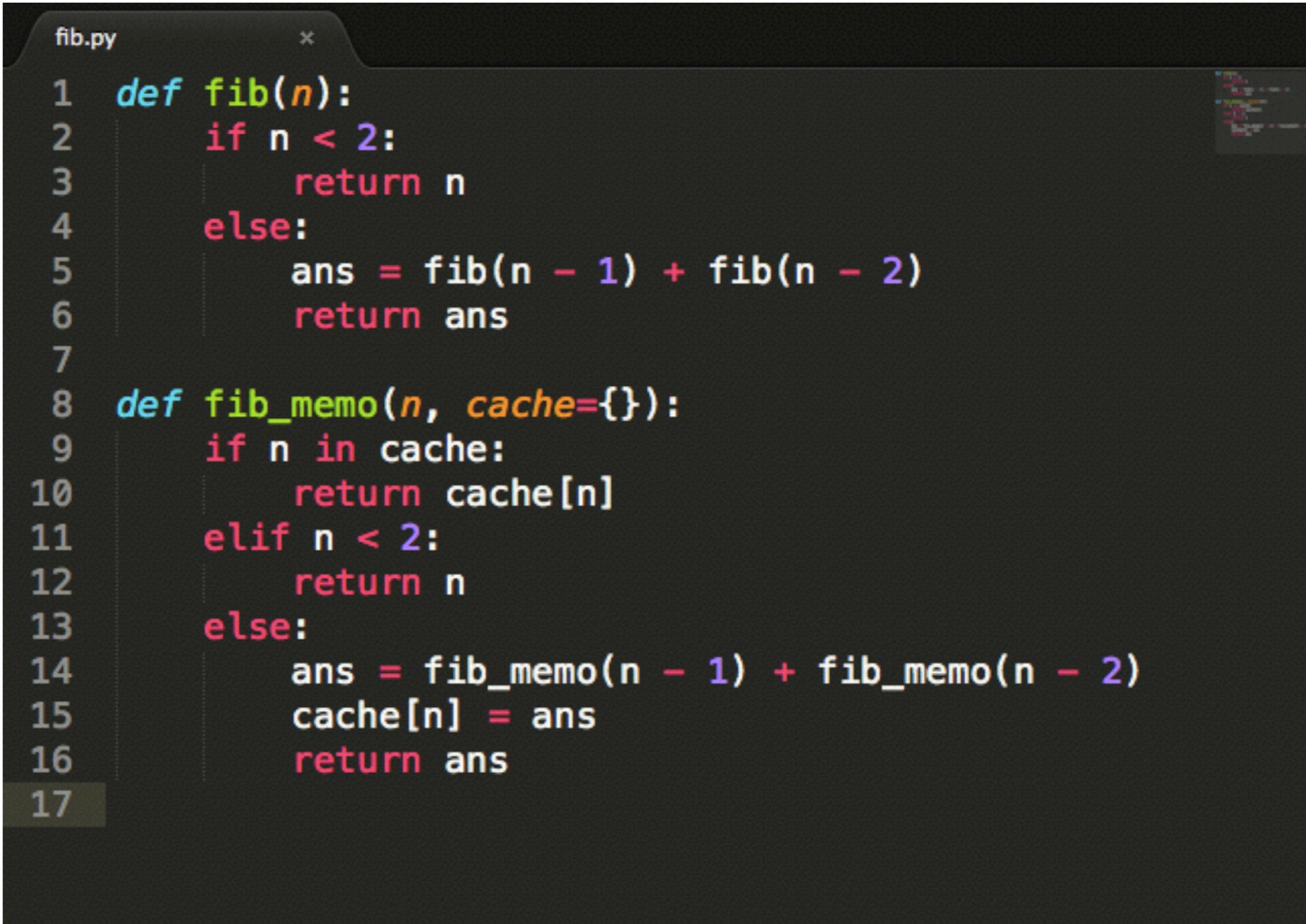
# We have problems

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 *  
1 def a_useful_function(arg):  
2     val = hard_work(arg)  
3     val.lots_of_effort()  
4     return val  
5  
6  
another.py *  
1 def magical_paste(filename):  
2     src = open(filename).read()  
3     namespace = {}  
4     exec src in namespace  
5     return namespace  
6  
7 useful = magical_paste('useful.py')  
8 bad = magical_paste('useful.py')  
9  
10 with open('input.txt', 'w') as f:  
11     data = f.read()  
12     useful.a_useful_function(data)  
13
```

# Memoize



The image shows a screenshot of a code editor with a dark theme. The file being edited is named "fib.py". The code contains two definitions: a recursive function "fib" and a memoized version "fib\_memo". The "fib" function is a simple recursive implementation of the Fibonacci sequence. The "fib\_memo" function uses a dictionary "cache" to store previously computed values, avoiding redundant calculations.

```
fib.py
1 def fib(n):
2     if n < 2:
3         return n
4     else:
5         ans = fib(n - 1) + fib(n - 2)
6         return ans
7
8 def fib_memo(n, cache={}):
9     if n in cache:
10        return cache[n]
11    elif n < 2:
12        return n
13    else:
14        ans = fib_memo(n - 1) + fib_memo(n - 2)
15        cache[n] = ans
16        return ans
17
```

# Memoize

```
useful.py *  
1 def a_useful_function(arg):  
2     val = hard_work(arg)  
3     val.lots_of_effort()  
4     return val  
5  
6  
another.py *  
1 def magical_paste(filename, executed = {})  
2     if filename in executed:  
3         return executed[filename]  
4     else:  
5         src = open(filename).read()  
6         namespace = {}  
7         exec src in namespace  
8         executed[filename] = namespace  
9         return namespace  
10  
11 useful = magical_paste('useful.py')  
12 no_problem = magical_paste('useful.py')  
13  
14 with open('input.txt', 'w') as f:  
15     data = f.read()  
16     useful.a_useful_function(data)  
17
```

# 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?

The image shows a code editor with two tabs open: "useful.py" and "another.py".

**useful.py:**

```
1 def a_useful_function(arg):
2     val = hard_work(arg)
3     val.lots_of_effort()
4     return val
5
6
```

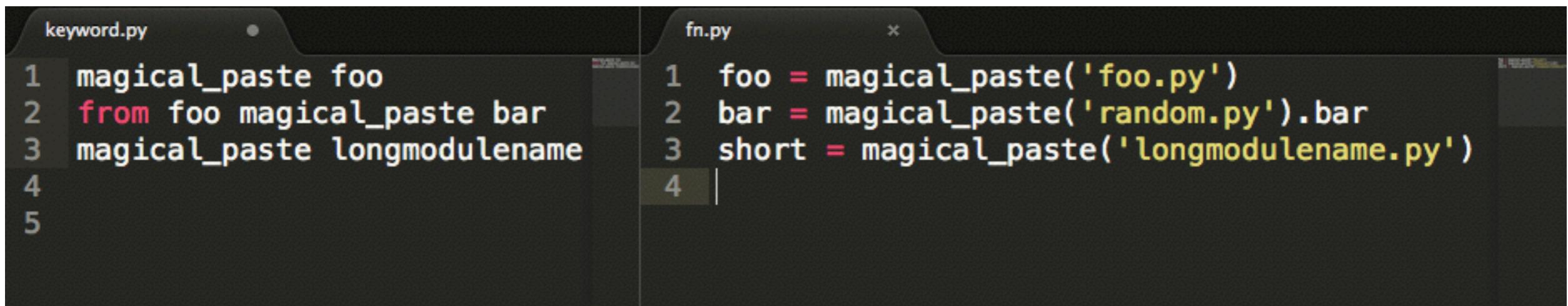
**another.py:**

```
1 def magical_paste(filename, pasted = {}, names_included = [], context = {}, ...):
2
3     if filename in pasted:
4         return pasted[filename]
5     else:
6         src = open(filename).read()
7         namespace = {}
8         exec src in namespace
9         pasted[filename] = namespace
10        return namespace
11
12
13
14 useful = magical_paste('useful.py')
15 no_problem = magical_paste('useful.py')
16
17 with open('input.txt', 'w') as f:
18     data = f.read()
19     useful.a_useful_function(data)
20
```

# 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



```
keyword.py
```

```
1 magical_paste foo
2 from foo magical_paste bar
3 magical_paste longmodulename
4
5
```

```
fn.py
```

```
1 foo = magical_paste('foo.py')
2 bar = magical_paste('random.py').bar
3 short = magical_paste('longmodulename.py')
4 |
```

# 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

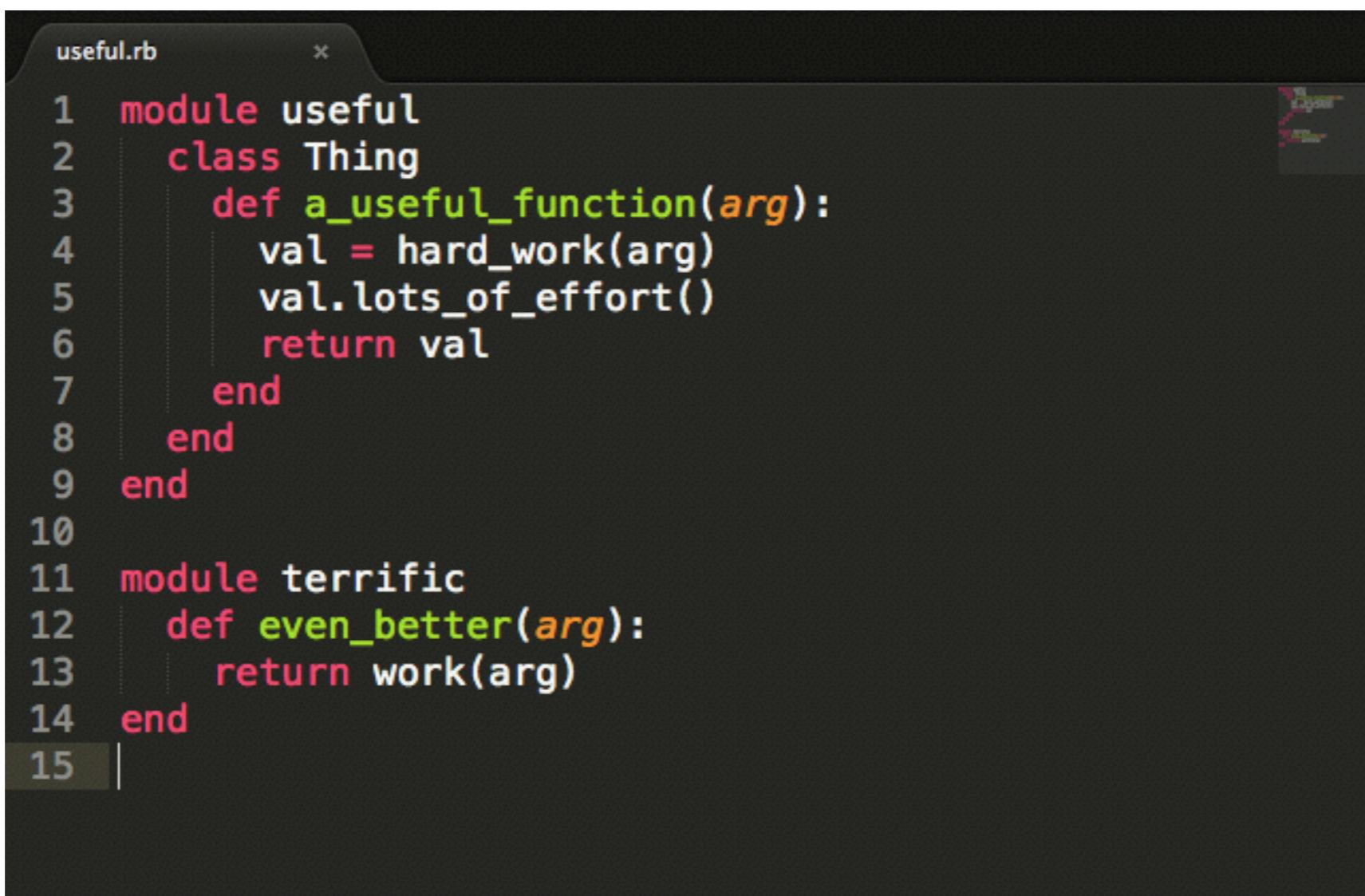
# #include in C

```
useful.py      *
1 def a_useful_function(arg):
2     val = hard_work(arg)
3     val.lots_of_effort()
4     return val
5

another.py      *
1 #include useful
2
3 with open('input.txt', 'w') as f:
4     data = f.read()
5     a_useful_function(data)
6
```

“magical copy & paste”

# Ruby modules



A screenshot of a code editor window titled "useful.rb". The code defines two modules: "useful" and "terrific". The "useful" module contains a class "Thing" with a method "a\_useful\_function" that performs some work and returns a value. The "terrific" module contains a method "even\_better" that returns a value. The code is color-coded, and the editor has a dark theme.

```
useful.rb
1 module useful
2   class Thing
3     def a_useful_function(arg):
4       val = hard_work(arg)
5       val.lots_of_effort()
6       return val
7     end
8   end
9 end
10
11 module terrific
12   def even_better(arg):
13     return work(arg)
14 end
15 |
```

# Modula

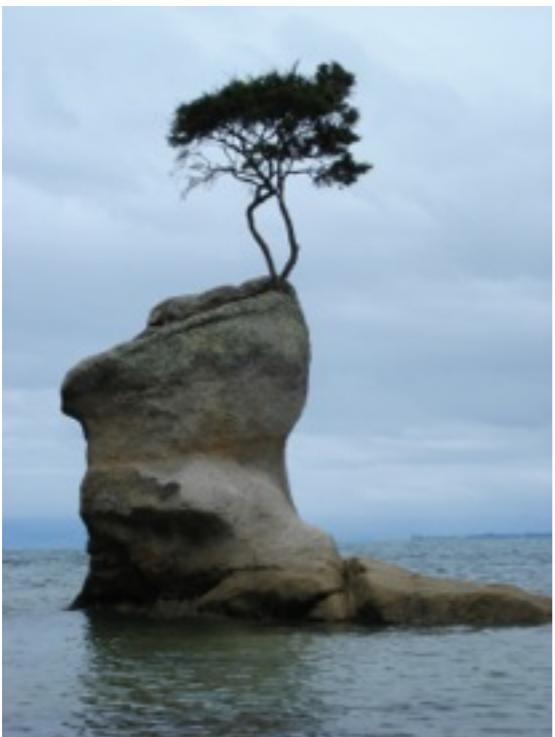
“A module is a set of procedures, data types, and variables, where the programmer has precise control over the names that are imported from and exported to the environment.”

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

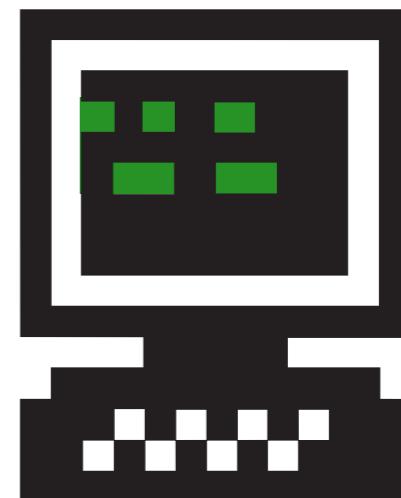
There are these two young fish swimming along and they happen to meet an older fish swimming the other way, who nods at them and says "Morning, boys. How's the water?" And the two young fish swim on for a bit, and then eventually one of them looks over at the other and goes "What the hell is water?"

- David Foster Wallace, 2005

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

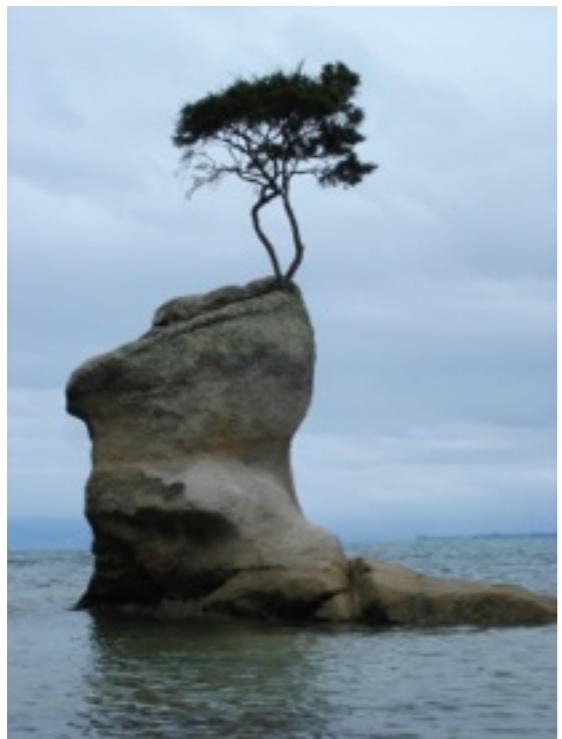


Allison Kaptur  
PyCon 2014  
[github.com/akaptur](https://github.com/akaptur)  
@akaptur

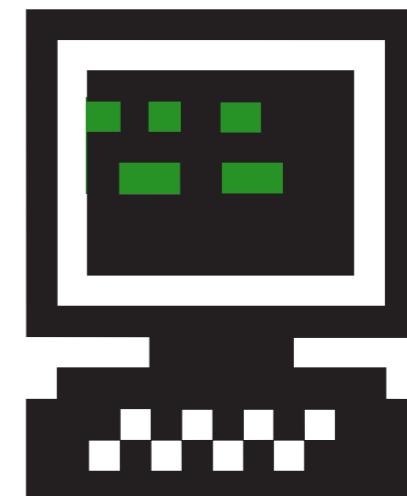


**Hacker** School

# Questions



Allison Kaptur  
PyCon 2014  
[github.com/akaptur](https://github.com/akaptur)  
@akaptur



**Hacker** School