Python **DECORATORS DEMYSTIFIED**

{
"event": "**PyCon ES 2013**"
"author": "**Pablo Enfedaque**"
"twitter": "**pablitoev56**"

Do you know what's happening each time you use the **@ (at) symbol** to decorate a function or class?

Today we are going to see how Python's **decorators syntactic sugar** works under the hood

Welcome!

{ "event": "**PyCon ES 2013**", "author": "**Pablo Enfedaque**", "twitter": "**pablitoev56**"}

And that’s why we will talk about Python **namespaces** and **scopes**

Welcome!

{ “event”: “**PyCon ES 2013**”, “author”: “**Pablo Enfedaque**”, “twitter”: “**pablitoev56**”}

And finally will **manually** implement and apply a **handcrafted decorator**

Welcome!

Let's start implementing some

useful stuff for the talk

```python
from collections import OrderedDict

CACHE = OrderedDict()
MAX_SIZE = 100

def set_key(key, value):
    "Set a key value, removing oldest key if MAX_SIZE exceeded"
    CACHE[key] = value
    if len(CACHE) > MAX_SIZE:
        CACHE.popitem(last=False)

def get_key(key):
    "Retrieve a key value from the cache, or None if not found"
    return CACHE.get(key, None)


>>> set_key("my_key", "the_value”)
>>> print(get_key("my_key"))
the_value

>>> print(get_key("not_found_key"))
None
```

A simple software cache

```python
def factorial(n):
    "Return n! (the factorial of n): n! = n * (n-1)!"
    if n < 2:
        return 1
    return n * factorial(n - 1)


>>> list(map(factorial, range(10)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]



def fibonacci(n):
    "Return nth fibonacci number: fib(n) = fib(n-1) + fib(n-2)"
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


>>> list(map(fibonacci, range(10)))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Factorial and fibonacci functions

Pretty **easy**, right?

However…

```
from collections import OrderedDict

CACHE = OrderedDict()
MAX_SIZE = 100

def set_key(key, value):
    "Set a key value, removing oldest key if MAX_SIZE exceeded"
    CACHE[key] = value
    if len(CACHE) > MAX_SIZE:
        CACHE.popitem(last=False)

def get_key(key):
    "Retrieve a key value from the cache, or None if not found"
    return CACHE.get(key, None)


>>> set_key("my_key", "the_value”)
>>> print(get_key("my_key"))
the_value

>>> print(get_key("not_found_key"))
None
```

How do we access this attribute?

```python
def factorial(n):
    "Return n! (the factorial of n): n! = n * (n-1)!"
    if n < 2:
        return 1
    return n * factorial(n - 1)


>>> list(map(factorial, range(10)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]



def fibonacci(n):
    "Return nth fibonacci number: fib(n) = fib(n-1) + fib(n-2)"
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


>>> list(map(fibonacci, range(10)))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

How are recursive calls possible?

```
>>> from os import path

>>> print(type(path), id(path))
<class 'module'> 4300435112




>>> from sys import path

>>> print(type(path), id(path))
<class 'list'> 4298480008




def split_path(path, sep="/"):
    print(type(path), id(path))
    return path.split(sep)

>>> split_path("/this/is/a/full/path")
<class 'str'> 4302038120
['', 'this', 'is', 'a', 'full', 'path']
```

A simpler case

```
>>> from os import path

>>> print(type(path), id(path))
<class 'module'> 4300435112



>>> from sys import path

>>> print(type(path), id(path))
<class 'list'> 4298480008



def split_path(path, sep="/"):
    print(type(path), id(path))
    return path.split(sep)

>>> split_path("/this/is/a/full/path")
<class 'str'> 4302038120
['', 'this', 'is', 'a', 'full', 'path']
```

The same name defined several times?

Let me introduce Python **namespaces**

A **namespace** is a mapping from names to objects

- The set of built-in names (functions, exceptions)
- Global names in a module (including imports)
- Local names in a function invocation
- Names defined in top-level invocation of interpreter

Python **namespaces** examples

There is no relation between names in different namespaces

Two modules or functions may define the same name without confusion

Python **namespaces**

Namespaces are created (and deleted) at different moments and **have different lifetimes**

Python **namespaces**

The **built-ins namespace** is created when the Python interpreter starts

And is never deleted

Python **namespaces** lifetimes

A **module global namespace** is created when the module definition is read-in (when it is imported)

Normally it lasts until the interpreter quits

Python **namespaces** lifetimes

A **function local namespace** is created each time it is called

It is deleted when the function returns or raises

Python **namespaces** lifetimes

And what about Python **scopes**?

A **scope** is a textual region of a program where a namespace is directly accessible

Scopes are **determined statically**

but **used dynamically**

Python **scopes**

At any time during execution, there are at least **three nested scopes** whose namespaces are directly accessible

Python **scopes**

Python **nested scopes**

1. The innermost scope contains the **local names**
   - The scopes of any **enclosing functions**, with non-local, but also non-global names
2. The next-to-last scope contains the **current module's global names**
3. The outermost scope is the namespace containing **built-in names**

Names are searched in nested scopes
from inside out

From locals to built-ins

Python nested **scopes**

```
$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0
(clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license"
for more information.
>>> import cache
>>> cache.set_key("my_key", 7)
>>> cache.get_key("my_key")
7
>>>
```

```python
"""
Simple cache implementation
"""
from collections import OrderedDict

CACHE = OrderedDict()
MAX_SIZE = 100

def set_key(key, value):
    "Set a key value, removing oldest key if MAX_SIZE exceeded"
    CACHE[key] = value
    if len(CACHE) > MAX_SIZE:
        CACHE.popitem(last=False)

def get_key(key):
    "Retrieve a key value from the cache, or None if not found"
    return CACHE.get(key, None)
```

Python **scopes**

{ “event”: “**PyCon ES 2013**”, “author”: “**Pablo Enfedaque**”, “twitter”: “**pablitoev56**”}

```
$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0
(clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license"
for more information
```

The outermost scope:
built-in names

```
>>> import cache
>>> cache.set_key("my_key", 7)
>>> cache.get_key("my_key")
7
>>>
```

The next-to-last scope:
current module's global names

```
"""
Simple cache implementation
"""
from collections import OrderedDict

CACHE = OrderedDict()
MAX_SIZE = 100

def set_key(key, value):
    "Set a key value, removing oldest key if MAX_SIZE exceeded"
    CACHE[key] = value
    if len(CACHE) > MAX_SIZE:
        CACHE.popitem(last=False)

def get_key(key):
    "Retrieve a key value from the cache, or None if not found"
    return CACHE.get(key, None)
```

The innermost scope:
current local names

Python **scopes**

{ "event": "**PyCon ES 2013**", "author": "**Pablo Enfedaque**", "twitter": "**pablitoev56**"}

```
def get_power_func(y):
    print("Creating function to raise to {}".format(y))

    def power_func(x):
        print("Calling to raise {} to power of {}".format(x, y))
        x = pow(x, y)
        return x

    return power_func


>>> raise_to_4 = get_power_func(4)
Creating function to raise to 3

>>> x = 3
>>> print(raise_to_4(x))
Calling to raise 3 to power of 4
81

>>> print(x)
3
```

Another more complex case

```
def get_power_func(y):
    print("Creating function to raise to {}".format(y))

    def power_func(x):
        print("Calling to raise {} to power of {}".format(x, y))
        x = pow(x, y)
        return x

    return power_func


>>> raise_to_4 = get_power_func(4)
Creating function to raise to 3

>>> x = 3
>>> print(raise_to_4(x))
Calling to raise 3 to power of 4
81

>>> print(x)
3
```

Where is **y** defined?

Nested scopes

```python
def get_power_func(y):
    print("Creating function to raise to {}".format(y))

    def power_func(x):
        print("Calling to raise {} to power of {}".format(x, y))
        x = pow(x, y)
        return x

    return power_func


>>> raise_to_4 = get_power_func(4)
Creating function to raise to 3

>>> x = 3
>>> print(raise_to_4(x))
Calling to raise 3 to power of 4
81

>>> print(x)
3
```

The innermost scope: local names

Enclosing function scope:
non-local non-global names

The next-to-last scope:
current module's global names

```
def get_power_func(y):
    print("Creating function to raise to {}".format(y))

    def power_func(x):
        print("Calling to raise {} to power of {}".format(x, y))
        x = pow(x, y)
        return x

    return power_func

>>> raise_to_4 = get_power_func(4)
Creating function to raise to 3

>>> print(raise_to_4.__globals__)
{'x': 3, 'raise_to_4': <function
get_power_func.<locals>.power_func at 0x100658488>,
'get_power_func': <function get_power_func at 0x1003b6048>, ...}


>>> print(raise_to_4.__closure__)
(<cell at 0x10065f048: int object at 0x10023b280>,)
```

There is a **closure!**

A function **closure** is a reference to each of the non-local variables of the function

```
def get_power_func(y):
    print("Creating function to raise to {}".format(y))

    def power_func(x):
        print("Calling to raise {} to power of {}".format(x, y))
        x = pow(x, y)
        return x

    return power_func

>>> raise_to_4 = get_power_func(4)
Creating function to raise to 3

>>> print(raise_to_4.__globals__)
{'x': 3, 'raise_to_4': <function
get_power_func.<locals>.power_func at 0x100658488>,
'get_power_func': <function get_power_func at 0x1003b6048>, ...}

>>> print(raise_to_4.__closure__)
(<cell at 0x10065f048: int object at 0x10023b280>,)
```

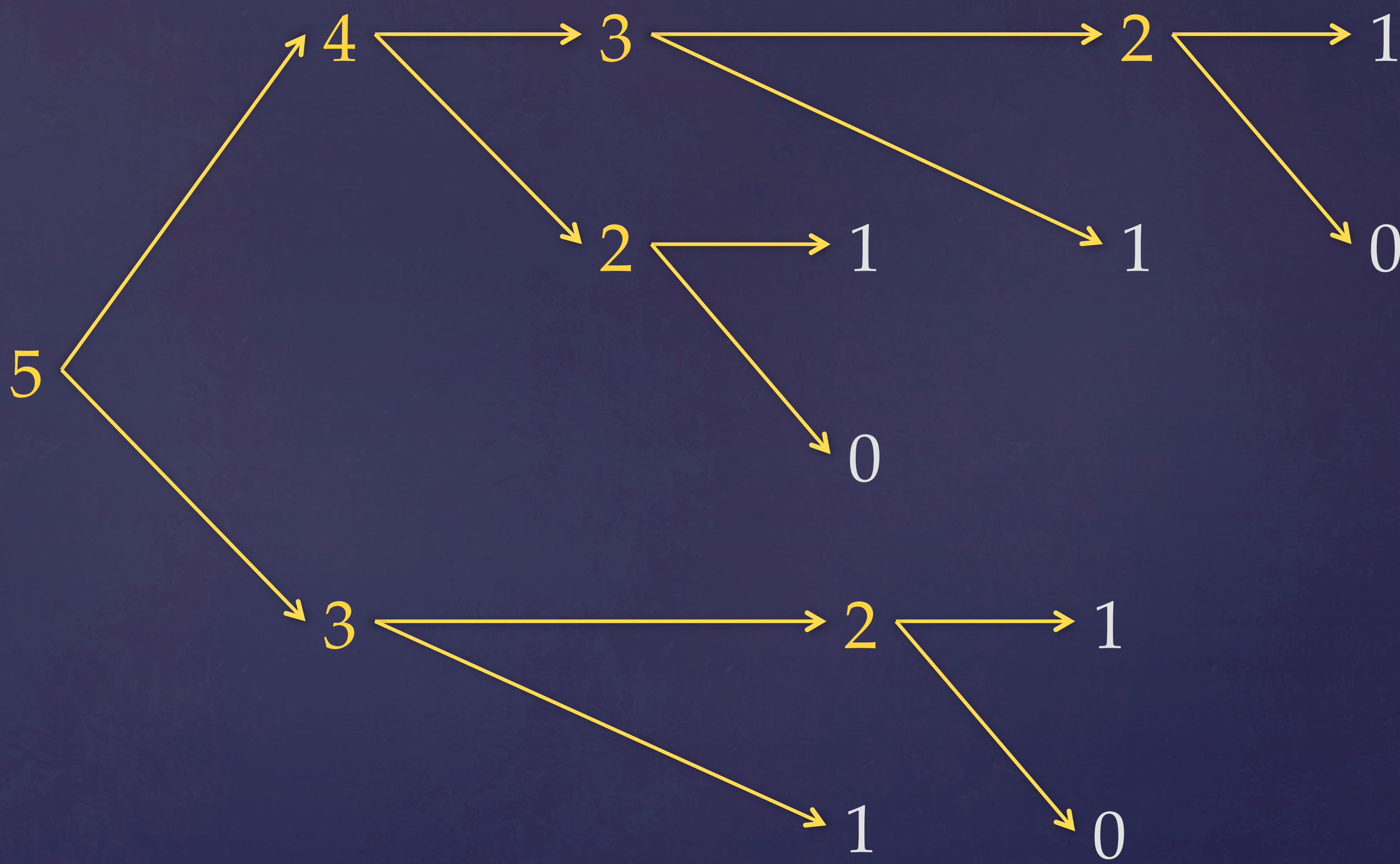
So, where is y defined?

Well, maybe you where are the **decorators** in this talk…

So, let's **manually** apply a decorator

```python
def factorial(n):
    "Return n! (the factorial of n): n! = n * (n-1)!"
    if n < 2:
        return 1
    return n * factorial(n - 1)

>>> start = time.time()
>>> factorial(35)
10333147966386144929666651337523200000000
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0007369518280029297

def fibonacci(n):
    "Return nth fibonacci number: fib(n) = fib(n-1) + fib(n-2)"
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> start = time.time()
>>> fibonacci(35)
9227465
>>> print("Elapsed:", time.time() - start)
Elapsed: 6.916048049926758
```

Let’s go back to these functions

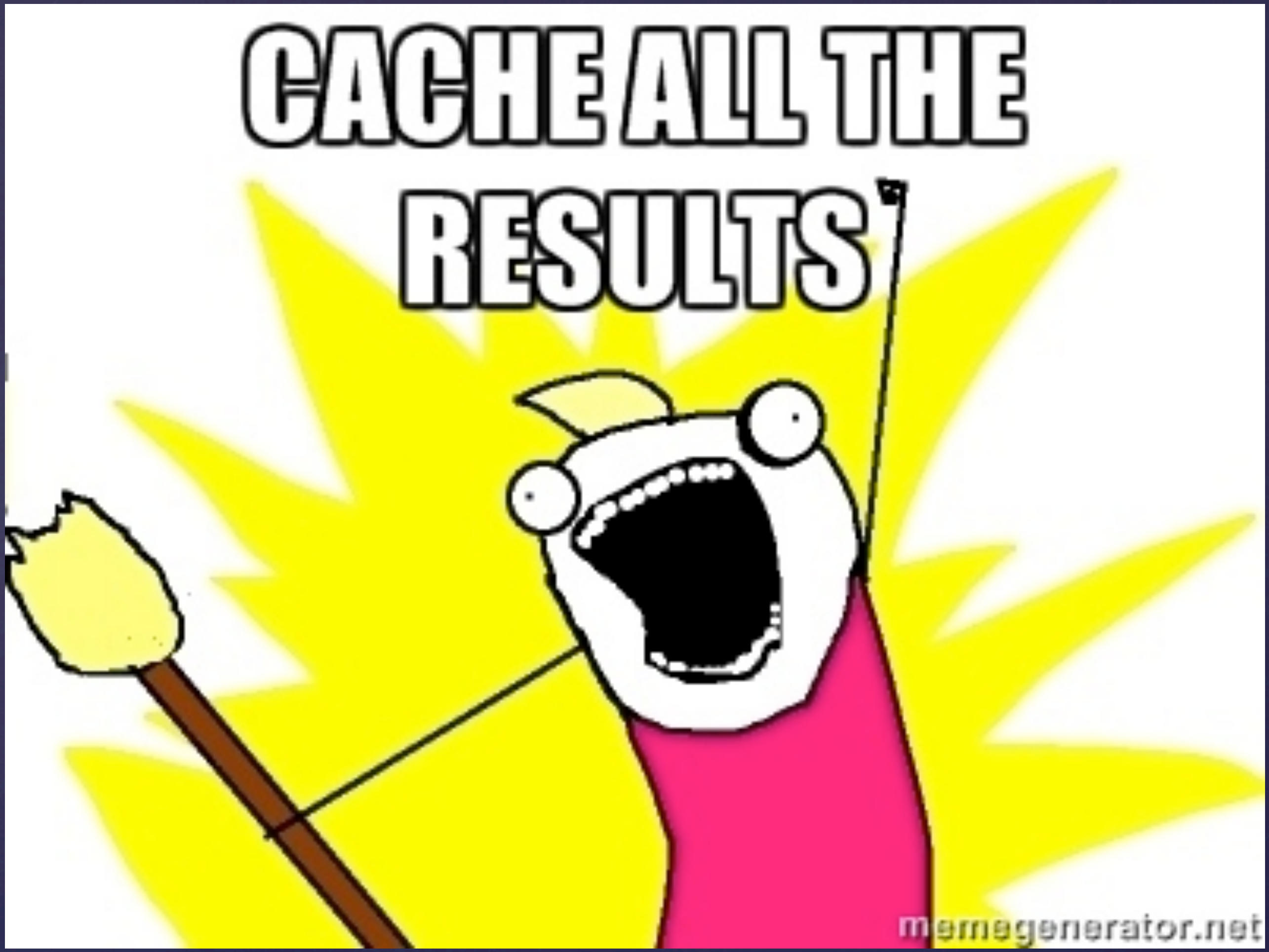


fibonacci(5) recursive calls graph

{ “event”: “**PyCon ES 2013**”, “author”: “**Pablo Enfedaque**”, “twitter”: “**pablitoev56**”}

```python
from collections import OrderedDict

CACHE = OrderedDict()
MAX_SIZE = 100

def set_key(key, value):
    "Set a key value, removing oldest key if MAX_SIZE exceeded"
    CACHE[key] = value
    if len(CACHE) > MAX_SIZE:
        CACHE.popitem(last=False)

def get_key(key):
    "Retrieve a key value from the cache, or None if not found"
    return CACHE.get(key, None)


>>> set_key("my_key", "the_value”)
>>> print(get_key("my_key"))
the_value

>>> print(get_key("not_found_key"))
None
```

Do you remember we have a cache?


CACHE ALL THE
RESULTS
memegenerator.net

```python
import cache

def fibonacci(n):
    "Return nth fibonacci number: fib(n) = fib(n-1) + fib(n-2)"
    if n < 2:
        return n
    fib = cache.get_key(n)
    if fib is None:
        fib = fibonacci(n - 1) + fibonacci(n - 2)
        cache.set_key(n, fib)
    return fib

>>> start = time.time()
>>> fibonacci(35)
9227465
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0007810592651367188

>>> start = time.time()
>>> fibonacci(100)
354224848179261915075
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0013179779052734375
```

What about this version?

```python
import cache

def fibonacci(n):
    "Return nth fibonacci number: fib(n) = fib(n-1) + fib(n-2)"
    if n < 2:
        return n
    fib = cache.get_key(n)
    if fib is None:
        fib = fibonacci(n - 1) + fibonacci(n - 2)
        cache.set_key(n, fib)
    return fib

>>> start = time.time()
>>> fibonacci(35)
9227465
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0007810592651367188

>>> start = time.time()
>>> fibonacci(100)
354224848179261915075
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0013179779052734375
```

DRY: Don’t Repeat Yourself!

Pay attention to the **magic trick**

```python
import time
import cache

def fibonacci(n):  # The function remains unchanged
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> real_fibonacci = fibonacci

def fibonacci(n):
    fib = cache.get_key(n)
    if fib is None:
        fib = real_fibonacci(n)
        cache.set_key(n, fib)
    return fib


>>> start = time.time()
>>> fibonacci(35)
9227465
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0010080337524414062
```

Original function is **not modified**

```
import time
import cache

def fibonacci(n):  # The function remains unchanged
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> real_fibonacci = fibonacci

def fibonacci(n):
    fib = cache.get_key(n)
    if fib is None:
        fib = real_fibonacci(n)
        cache.set_key(n, fib)
    return fib


>>> start = time.time()
>>> fibonacci(35)
9227465
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0010080337524414062
```

Which function is called here?

The next-to-last scope:
current module's global names

```
import time
import cache

def fibonacci(n):  # The function remains unchanged
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

The innermost scope:
current local names

```
>>> real_fibonacci = fibonacci

def fibonacci(n):
    fib = cache.get_key(n)
    if fib is None:
        fib = real_fibonacci(n)
        cache.set_key(n, fib)
    return fib


>>> start = time.time()
>>> fibonacci(35)
9227465
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0010080337524414062
```

Remember the **scopes**…

And now the trick in **slow motion**

```python
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> print(id(fibonacci))
4298858568
```

{
    fibonacci: 4298858568
}

Global names

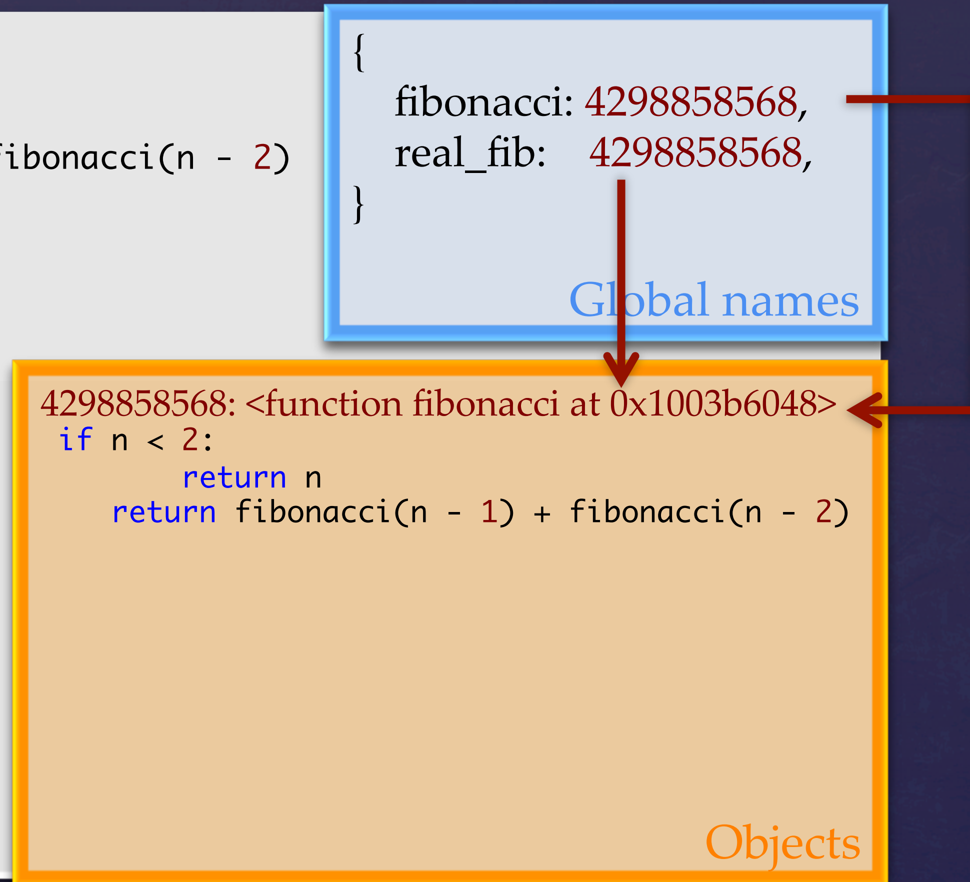
4298858568: <function fibonacci at 0x1003b6048>

```python
 if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Objects

1. Create original fibonacci function

```
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> print(id(fibonacci))
4298858568

>>> real_fib = fibonacci
```



2. Create alternative name pointing to the same function object

```python
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> print(id(fibonacci))
4298858568

>>> real_fib = fibonacci

def fibonacci(n):
    fib = cache.get_key(n)
    if fib is None:
        fib = real_fib (n)
        cache.set_key(n, fib)
    return fib

>>> print(id(fibonacci))
4302081696

>>> print(id(real_fib))
4298858568
```

```
{
    fibonacci: 4302081696,
    real_fib:  4298858568,
}
```

Global names

```
4298858568: <function fibonacci at 0x1003b6048>
 if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


4302081696: <function fibonacci at 0x1006c8ea0>
 fib = cache.get_key(n)
    if fib is None:
        fib = real_fib (n)
        cache.set_key(n, fib)
    return fib
```

Objects

3. Replace original name with new a function which calls the alternative name

```
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> print(id(fibonacci))
4298858568

>>> real_fib = fibonacci

def fibonacci(n):
    fib = cache.get_key(n)
    if fib is None:
        fib = real_fib (n)
        cache.set_key(n, fib)
    return fib

>>> print(id(fibonacci))
4302081696

>>> print(id(real_fib))
4298858568
```

```
{
  fibonacci: 4302081696,
  real_fib:  4298858568,
}
```

Global names

```
4298858568: <function fibonacci at 0x1003b6048>
 if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

4302081696: <function fibonacci at 0x1006c8ea0>
 fib = cache.get_key(n)
    if fib is None:
        fib = real_fib (n)
        cache.set_key(n, fib)
    return fib
```

Objects

This way we swap both functions

```python
import time
import cache

def fibonacci(n):  # The function remains unchanged
    if n < 2:
        return n
    print("Real fibonacci of", n, "calling", id(fibonacci))
    return fibonacci(n - 1) + fibonacci(n - 2)



>>> start = time.time()
>>> fibonacci(5)
Real fibonacci of 5 calling 4298858568
Real fibonacci of 4 calling 4298858568
Real fibonacci of 3 calling 4298858568
Real fibonacci of 2 calling 4298858568
Real fibonacci of 2 calling 4298858568
Real fibonacci of 3 calling 4298858568
Real fibonacci of 2 calling 4298858568
5
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0008890628814697266
```

Let’s repeat the trick with traces

```
>>> print(id(fibonacci))
4298858568
>>> real_fib = fibonacci

def fibonacci(n):
    fib = cache.get_key(n)
    if fib is None:
        print("Cached fibonacci of", n, "calling", id(real_fib))
        fib = real_fib(n)
        cache.set_key(n, fib)
    return fib

>>> start = time.time()
>>> fibonacci(5)
Cached fibonacci of 5 calling 4298858568
Real fibonacci of 5 calling 4302495392
Cached fibonacci of 4 calling 4298858568
Real fibonacci of 4 calling 4302495392
Cached fibonacci of 3 calling 4298858568
Real fibonacci of 3 calling 4302495392
Cached fibonacci of 2 calling 4298858568
Real fibonacci of 2 calling 4302495392
Cached fibonacci of 1 calling 4298858568
Cached fibonacci of 0 calling 4298858568
5
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0008230209350585938
```

Let's make this trick **fully reusable**

Let's make it work with **any* function**

```python
import cache

def memoize_any_function(func_to_memoize):
    "Return a wrapped version of the function using memoization"

    def memoized_version_of_func(n):
        "Wrapper using memoization"
        res = cache.get_key(n)
        if res is None:
            res = func_to_memoize(n)  # Call the real function
            cache.set_key(n, res)
        return res
    return memoized_version_of_func

def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci = memoize_any_function(fibonacci)
```

A factory of memoization functions

```python
import cache

def memoize_any_function(func_to_memoize):
    "Return a wrapped version of the function using memoization"

    def memoized_version_of_func(n):
        "Wrapper using memoization"
        res = cache.get_key(n)
        if res is None:
            res = func_to_memoize(n)  # Call the real function
            cache.set_key(n, res)
        return res
    return memoized_version_of_func

def factorial(n):
    if n < 2:
        return 1
    return n * factorial(n - 1)

>>> factorial= memoize_any_function(factorial)
```

A factory of memoization functions

```
import time

>>> start = time.time()
>>> fibonacci(250)
7896325826131730509282738943634332893686268675876375
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0009610652923583984

>>> start = time.time()
>>> factorial(250)
3232856260909107732320814552024368470994843717673780666747942427
1128237475551112094888179153710281994509285073531894329267309317
1280899082279103027907128192167652724018926473321804118626100683
2925365133678939089569935713530175040513178760077247933065402339
0061648255522488194365725860573992226412548329822048491377217766
5064127685880715312897877767295191399084437747870258917297325515
0283241787320658188482062478582659808848825548800000000000000000
000000000000000000000000000000000000000000000
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.00249481201171875
```

A factory of memoization functions

```python
import time
import cache

def memoize_any_function(func_to_memoize):
    "Return a wrapped version of the function using memoization"

    def memoized_version_of_func(n):
        "Wrapper using memoization"
        res = cache.get_key(n)
        if res is None:
            res = func_to_memoize(n)  # Call the real function
            cache.set_key(n, res)
        return res
    return memoized_version_of_func

def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci = memoize_any_function(fibonacci)

>>> start = time.time()
>>> fibonacci(35)
9227465
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0009610652923583984
```

```python
import time
import cache

def memoize_any_function(func_to_memoize):
    "Return a wrapped version of the function using memoization"

    def memoized_version_of_func(n):
        "Wrapper using memoization"
        res = cache.get_key(n)
        if res is None:
            res = func_to_memoize(n)  # Call the real function
            cache.set_key(n, res)
        return res
    return memoized_version_of_func

@memoize_any_function
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


>>> start = time.time()
>>> fibonacci(35)
9227465
>>> print("Elapsed:", time.time() - start)
Elapsed: 0.0009610652923583984
```

And finally, at long last, **decorators**

```python
import cache

def memoize_any_function(func_to_memoize):
    "Return a wrapped version of the function using memoization"

    def memoized_version_of_func(n):
        "Wrapper using memoization"
        res = cache.get_key(n)
        if res is None:
            res = func_to_memoize(n)  # Call the real function
            cache.set_key(n, res)
        return res
    return memoized_version_of_func


def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci = memoize_any_function(fibonacci)
```

Pay attention…

```python
import cache

def memoize_any_function(func_to_memoize):
    "Return a wrapped version of the function using memoization"

    def memoized_version_of_func(n):
        "Wrapper using memoization"
        res = cache.get_key(n)
        if res is None:
            res = func_to_memoize(n)  # Call the real function
            cache.set_key(n, res)
        return res
    return memoized_version_of_func

@memoize_any_function
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Did you spot the difference?

```
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci = memoize_any_function(fibonacci)
```

This is the only thing the @ does

Calls a decorator providing the decorated function, then makes the function name point to the result

```
@memoize_any_function
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n – 2)
```

Decorators demystified

Thanks for coming!

Slides:

Q&A