merge6

June 14, 2023

0.0.1 Global and Local Scopes

In Python the **global** scope refers to the **module** scope.

The scope of a variable is normally defined by **where** it is (lexically) defined in the code.

[1]:
```
a = 10
```

In this case, **a** is defined inside the main module, so it is a global variable.

[2]:
```
def my_func(n):
    c = n ** 2
    return c
```

In this case, **c** was defined inside the function **my_func**, so it is **local** to the function **my_func**. In this example, **n** is also **local** to **my_func**

Global variables can be accessed from any inner scope in the module, for example:

[3]:
```
def my_func(n):
    print('global:', a)
    c = a ** n
    return c
```

[4]:
```
my_func(2)
```

```
global: 10
```

[4]:
```
100
```

As you can see, **my_func** was able to reference the global variable **a**.

But remember that the scope of a variable is determined by where it is assigned. In particular, any variable defined (i.e. assigned a value) inside a function is local to that function, even if the variable name happens to be global too!

[5]:
```
def my_func(n):
    a = 2
    c = a ** 2
    return c
```

[6]:
```
print(a)
print(my_func(3))
print(a)
```

```
10
4
10
```

In order to change the value of a global variable within an inner scope, we can use the **global** keyword as follows:

[7]:
```
def my_func(n):
    global a
    a = 2
    c = a ** 2
    return c
```

[8]:
```
print(a)
print(my_func(3))
print(a)
```

```
10
4
2
```

As you can see, the value of the global variable **a** was changed from within **my__func**.

In fact, we can **create** global variables from within an inner function - Python will simply create the variable and place it in the **global** scope instead of the **local scope**:

[9]:
```
def my_func(n):
    global var
    var = 'hello world'
    return n ** 2
```

Now, **var** does not exist yet, since the function has not run:

[10]:
```
print(var)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-571cba235a7f> in <module>()
----> 1 print(var)

NameError: name 'var' is not defined
```

Once we call the function though, it will create that global **var**:

[11]:
```
my_func(2)
```

```
[11]: 4
```

```
[12]: print(var)
```

```
hello world
```

Beware!! Remember that whenever you assign a value to a variable without having specified the variable as **global**, it is **local** in the current scope. **Moreover**, it does not matter **where** the assignment in the code takes place, the variable is considered local in the **entire** scope - Python determines the scope of objects at compile-time, not at run-time.

Let's see an example of this:

```
[13]: a = 10
      b = 100
```

```
[14]: def my_func():
          print(a)
          print(b)
```

```
[15]: my_func()
```

```
10
100
```

So, this works as expected - **a** and **b** are taken from the global scope since they are referenced **before** being assigned a value in the local scope.

But now consider the following example:

```
[16]: a = 10
      b = 100

      def my_func():
          print(a)
          print(b)
          b = 1000
```

```
[17]: my_func()
```

```
10
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-17-d82eda95de40> in <module>()
----> 1 my_func()

<ipython-input-16-a2b60f95cac1> in my_func()
      4 def my_func():
```

```
      5     print(a)
----> 6     print(b)
      7     b = 1000

UnboundLocalError: local variable 'b' referenced before assignment
```

As you can see, **b** in the line `print(b)` is considered a **local** variable - that's because the **next** line **assigns** a value to **b** - hence **b** is scoped as local by Python for the **entire** function.

Of course, functions are also objects, and scoping applies equally to function objects too. For example, we can "mask" the built-in `print` Python function:

[18]:
```
print = lambda x: 'hello {0}!'.format(x)

def my_func(name):
        return print(name)

my_func('world')
```

[18]:
```
'hello world!'
```

You may be wondering how we get our **real** `print` function back!

[19]:
```
del print
```

[20]:
```
print('hello')
```

```
hello
```

Yay!!

If you have experience in some other programming languages you may be wondering if loops and other code "blocks" have their own local scope too. For example in Java, the following would not work:

`for (int i=0; i<10; i++) {     int x = 2 * i; } system.out.println(x);`

But in Python it works perfectly fine:

[21]:
```
for i in range(10):
    x = 2 * i
print(x)
```

```
18
```

In this case, when we assigned a value to `x`, Python put it in the global (module) scope, so we can reference it after the `for` loop has finished running.

0.0.2 Nonlocal Scopes

Functions defined inside anther function can reference variables from that enclosing scope, just like functions can reference variables from the global scope.

[1]:
```
def outer_func():
    x = 'hello'

    def inner_func():
        print(x)

    inner_func()
```

[2]:
```
outer_func()
```

```
hello
```

In fact, any level of nesting is supported since Python just keeps looking in enclosing scopes until it finds what it needs (or fails to find it by the time it finishes looking in the built-in scope, in which case a runtime error occurrs.)

[3]:
```
def outer_func():
    x = 'hello'
    def inner1():
        def inner2():
            print(x)
        inner2()
    inner1()
```

[4]:
```
outer_func()
```

```
hello
```

But if we **assign** a value to a variable, it is considered part of the local scope, and potentially **masks** enclsogin scope variable names:

[5]:
```
def outer():
    x = 'hello'
    def inner():
        x = 'python'
    inner()
    print(x)
```

[6]:
```
outer()
```

```
hello
```

As you can see, **x** in **outer** was not changed.

To achieve this, we can use the **nonlocal** keyword:

[7]:
```python
def outer():
    x = 'hello'
    def inner():
        nonlocal x
        x = 'python'
    inner()
    print(x)
```

[8]:
```python
outer()
```

```
python
```

Of course, this can work at any level as well:

[9]:
```python
def outer():
    x = 'hello'

    def inner1():
        def inner2():
            nonlocal x
            x = 'python'
        inner2()
    inner1()
    print(x)
```

[10]:
```python
outer()
```

```
python
```

How far Python looks up the chain depends on the first occurrence of the variable name in an enclosing scope.

Consider the following example:

[11]:
```python
def outer():
    x = 'hello'
    def inner1():
        x = 'python'
        def inner2():
            nonlocal x
            x = 'monty'
        print('inner1 (before):', x)
        inner2()
        print('inner1 (after):', x)
    inner1()
    print('outer:', x)
```

[12]:
```python
outer()
```

```
inner1 (before): python
inner1 (after): monty
outer: hello
```

What happened here, is that `x` in `inner1` **masked** `x` in `outer`. But `inner2` indicated to Python that `x` was nonlocal, so the first local variable up in the enclosing scope chain Python found was the one in `inner1`, hence `x` in `inner2` is actually referencing `x` that is local to `inner1`

We can change this behavior by making the variable `x` in `inner` nonlocal as well:

[13]:
```python
def outer():
    x = 'hello'
    def inner1():
        nonlocal x
        x = 'python'
        def inner2():
            nonlocal x
            x = 'monty'
        print('inner1 (before):', x)
        inner2()
        print('inner1 (after):', x)
    inner1()
    print('outer:', x)
```

[14]:
```python
outer()
```

```
inner1 (before): python
inner1 (after): monty
outer: monty
```

[15]:
```python
x = 100
def outer():
    x = 'python'  # masks global x
    def inner1():
        nonlocal x  # refers to x in outer
        x = 'monty' # changed x in outer scope
        def inner2():
            global x  # refers to x in global scope
            x = 'hello'
        print('inner1 (before):', x)
        inner2()
        print('inner1 (after):', x)
    inner1()
    print('outer', x)
```

[16]:
```python
outer()
print(x)
```

```
inner1 (after): monty
```

```
outer monty
100
```

But this will not work. In `inner` Python is looking for a local variable called `x`. `outer` has a label called `x`, but it is a global variable, not a local one - hence Python does not find a local variable in the scope chain.

[17]:
```
x = 100
def outer():
    global x
    x = 'python'

    def inner():
        nonlocal x
        x = 'monty'
    inner()
```

```
  File "<ipython-input-17-3ccaec905318>", line 7
    nonlocal x
    ^
SyntaxError: no binding for nonlocal 'x' found
```

[ ]:

0.0.3 Closures

Let's examine that concept of a cell to create an indirect reference for variables that are in multiple scopes.

[1]:
```
def outer():
    x = 'python'
    def inner():
        print(x)
    return inner
```

[2]:
```
fn = outer()
```

[3]:
```
fn.__code__.co_freevars
```

[3]:
```
('x',)
```

As we can see, `x` is a free variable in the closure.

[4]:
```
fn.__closure__
```

[4]:
```
(<cell at 0x0000015F5299B4C8: str object at 0x0000015F51092068>,)
```

Here we see that the free variable x is actually a reference to a cell object that is itself a reference to a string object.

Let's see what the memory address of `x` is in the outer function and the inner function. To be sure string interning does not play a role, I am going to use an object that we know Python will not automatically intern, like a list.

[5]:
```
def outer():
    x = [1, 2, 3]
    print('outer:', hex(id(x)))
    def inner():
        print('inner:', hex(id(x)))
        print(x)
    return inner
```

[6]:
```
fn = outer()
```

```
outer: 0x15f52907988
```

[7]:
```
fn.__closure__
```

[7]:
```
(<cell at 0x0000015F5299B768: list object at 0x0000015F52907988>,)
```

[8]:
```
fn()
```

```
inner: 0x15f52907988
[1, 2, 3]
```

As you can see, each the memory address of `x` in `outer`, `inner` and the cell all point to the same object.

Modifying the Free Variable We know we can modify nonlocal variables by using the `nonlocal` keyword. So the following will work:

[9]:
```
def counter():
    count = 0 # local variable

    def inc():
        nonlocal count  # this is the count variable in counter
        count += 1
        return count
    return inc
```

[10]:
```
c = counter()
```

[11]:
```
c()
```

[11]:
```
1
```

[12]:
```
c()
```

[12]:
```
2
```

Shared Extended Scopes As we saw in the lecture, we can set up nonlocal variables in different inner functions that reference the same outer scope variable, i.e. we have a free variable that is shared between two closures. This works because both non local variables and the outer local variable all point back to the same cell object.

[13]:
```
def outer():
    count = 0
    def inc1():
        nonlocal count
        count += 1
        return count

    def inc2():
        nonlocal count
        count += 1
        return count

    return inc1, inc2
```

[14]:
```
fn1, fn2 = outer()
```

[15]:
```
fn1.__closure__, fn2.__closure__
```

[15]:
```
((<cell at 0x0000015F5299B738: int object at 0x00000000506FEC50>,),
 (<cell at 0x0000015F5299B738: int object at 0x00000000506FEC50>,))
```

As you can see here, the `count` label points to the same cell.

[16]:
```
fn1()
```

[16]:
```
1
```

[17]:
```
fn1()
```

[17]:
```
2
```

[18]:
```
fn2()
```

[18]:
```
3
```

0.0.4 Multiple Instances of Closures

Recall that **every** time a function is called, a **new** local scope is created.

[19]:
```
from time import perf_counter

def func():
    x = perf_counter()
    print(x, id(x))
```

[20]:
```
func()
```

```
2.7089464582150425e-07 1508916709680
```

[21]:
```
func()
```

```
0.011222623387093279 1508916709680
```

The same thing happens with closures, they have their own extended scope every time the closure is created:

[22]:
```
def pow(n):
    # n is local to pow
    def inner(x):
        # x is local to inner
        return x ** n
    return inner
```

In this example, `n`, in the function `inner` is a free variable, so we have a closure that contains `inner` and the free variable `n`

[23]:
```
square = pow(2)
```

[24]:
```
square(5)
```

[24]:
```
25
```

[25]:
```
cube = pow(3)
```

[26]:
```
cube(5)
```

[26]:
```
125
```

We can see that the cell used for the free variable in both cases is **different**:

[27]:
```
square.__closure__
```

[27]:
```
(<cell at 0x0000015F5299B8B8: int object at 0x00000000506FEC90>,)
```

[28]:
```
cube.__closure__
```

[28]:
```
(<cell at 0x0000015F5299BAC8: int object at 0x00000000506FECB0>,)
```

In fact, these functions (`square` and `cube`) are **not** the same functions, even though they were "created" from the same `power` function:

[29]:
```
id(square), id(cube)
```

[29]:
```
(1508919294560, 1508919295784)
```

0.0.5 Beware!

Remember when I said the captured variable is a reference established when the closure is created, but the value is looked up only once the function is called?

This can create very subtle bugs in your program.

Consider the following example where we want to create some functions that can add 1, 2, 3, 4 and to whatever is passed to them.

We could do the following:

[30]:
```
def adder(n):
    def inner(x):
        return x + n
    return inner
```

[31]:
```
add_1 = adder(1)
add_2 = adder(2)
add_3 = adder(3)
add_4 = adder(4)
```

[32]:
```
add_1(10), add_2(10), add_3(10), add_4(10)
```

[32]:
```
(11, 12, 13, 14)
```

But suppose we want to get a little fancier and do it as follows:

[33]:
```
def create_adders():
    adders = []
    for n in range(1, 5):
        adders.append(lambda x: x + n)
    return adders
```

[34]:
```
adders = create_adders()
```

Now technically we have 4 functions in the `adders` list:

[35]:
```
adders
```

[35]:
```
[<function __main__.create_adders.<locals>.<lambda>>,
 <function __main__.create_adders.<locals>.<lambda>>,
 <function __main__.create_adders.<locals>.<lambda>>,
<function __main__.create_adders.<locals>.<lambda>>]
```

The first one should add 1 to the value we pass it, the second should add 2, and so on.

```
[36]: adders[3](10)
```

```
[36]: 14
```

Yep, that works for the 4th function.

```
[37]: adders[0](10)
```

```
[37]: 14
```

Uh Oh - what happened? In fact we get the same behavior from every one of those functions:

```
[38]: adders[0](10), adders[1](10), adders[2](10), adders[3](10)
```

```
[38]: (14, 14, 14, 14)
```

Remember what I said about when the variable is captured and when the value is looked up?

When the lambdas are **created** their `n` is the `n` used in the loop - the **same** `n`!!

```
[39]: adders[0].__code__.co_freevars
```

```
[39]: ('n',)
```

```
[40]: adders[0].__closure__
```

```
[40]: (<cell at 0x0000015F5299B3D8: int object at 0x00000000506FECD0>,)
```

```
[41]: adders[1].__closure__
```

```
[41]: (<cell at 0x0000015F5299B3D8: int object at 0x00000000506FECD0>,)
```

```
[42]: adders[2].__closure__
```

```
[42]: (<cell at 0x0000015F5299B3D8: int object at 0x00000000506FECD0>,)
```

```
[43]: adders[3].__closure__
```

```
[43]: (<cell at 0x0000015F5299B3D8: int object at 0x00000000506FECD0>,)
```

So, by the time we call `adder[i]`, the free variable `n` (shared between all adders) is set to 4.

```
[44]: hex(id(4))
```

```
[44]: '0x506fecd0'
```

As we can see the memory address of the singleton integer 4, is what that cell is pointint to.

If you want to use a loop to do this and not end up using the same cell for each of the free variables, we can use a simple trick that forces the evaluation of `n` at the time the closure is **created**, instead of when the closure function is evaluated.

We can do this by creating a parameter for `n` in our lambda whose default value is the current value of `n` - remember from an earlier video that parameter defaults are avaluated when the function is created, not called.

[45]:
```
def create_adders():
    adders = []
    for n in range(1, 5):
        adders.append(lambda x, step=n: x + step)
    return adders
```

[46]:
```
adders = create_adders()
```

[47]:
```
adders[0].__closure__
```

Why aren't we getting anything in the closure? What about free variables?

[48]:
```
adders[0].__code__.co_freevars
```

[48]:
```
()
```

Hmm, nothing either… Why?

Well, look at the lambda in that loop. Does it reference the variable `n` (other than in the default value)? No. Hence, `n` is **not** a free variable in this case, and our lambda is just a plain lambda, not a closure.

And this code will now work as expected:

[49]:
```
adders[0](10)
```

[49]:
```
11
```

[50]:
```
adders[1](10)
```

[50]:
```
12
```

[51]:
```
adders[2](10)
```

[51]:
```
13
```

[52]:
```
adders[3](10)
```

[52]:
```
14
```

You just need to understand that since the default values are evaluated when the function (lambda in this case) is **created**, the then-current `n` value is assigned to the local variable `step`. So `step` will not change every time the lambda is called, and since n is not referenced inside the function (and therefore evaluated when the lambda is called), `n` is not a free variable.

Nested Closures We can also nest closures, as can be seen in this example:

[53]:
```
def incrementer(n):
    def inner(start):
        current = start
        def inc():
            a = 10  # local var
            nonlocal current
            current += n
            return current
        return inc
    return inner
```

[54]:
```
fn = incrementer(2)
```

[55]:
```
fn
```

[55]:
```
<function __main__.incrementer.<locals>.inner>
```

[56]:
```
fn.__code__.co_freevars
```

[56]:
```
('n',)
```

[57]:
```
fn.__closure__
```

[57]:
```
(<cell at 0x0000015F5299B798: int object at 0x00000000506FEC90>,)
```

[58]:
```
inc_2 = fn(100)
```

[59]:
```
inc_2
```

[59]:
```
<function __main__.incrementer.<locals>.inner.<locals>.inc>
```

[60]:
```
inc_2.__code__.co_freevars
```

[60]:
```
('current', 'n')
```

[61]:
```
inc_2.__closure__
```

[61]:
```
(<cell at 0x0000015F5299B318: int object at 0x00000000506FF8D0>,
 <cell at 0x0000015F5299B798: int object at 0x00000000506FEC90>)
```

Here you can see that the second free variable `n`, is pointing to the same cell as the free variable in `fn`.

Note that **a** is a local variable, and is not considered a free variable.

And we can call the closures as follows:

```
[62]: inc_2()
```

```
[62]: 102
```

```
[63]: inc_2()
```

```
[63]: 104
```

```
[64]: inc_3 = incrementer(3)(200)
```

```
[65]: inc_3()
```

```
[65]: 203
```

```
[66]: inc_3()
```

```
[66]: 206
```

0.0.6 Closure Applications (Part 1)

In this example we are going to build an averager function that can average multiple values.

The twist is that we want to simply be able to feed numbers to that function and get a running average over time, not average a list which requires performing the same calculations (sum and count) over and over again.

```
[1]: class Averager:
         def __init__(self):
             self.numbers = []

         def add(self, number):
             self.numbers.append(number)
             total = sum(self.numbers)
             count = len(self.numbers)
             return total / count
```

```
[2]: a = Averager()
```

```
[3]: a.add(10)
```

```
[3]: 10.0
```

[4]:
```
a.add(20)
```

[4]:
```
15.0
```

[5]:
```
a.add(30)
```

[5]:
```
20.0
```

We can do this using a closure as follows:

[6]:
```
def averager():
    numbers = []
    def add(number):
        numbers.append(number)
        total = sum(numbers)
        count = len(numbers)
        return total / count
    return add
```

[7]:
```
a = averager()
```

[8]:
```
a(10)
```

[8]:
```
10.0
```

[9]:
```
a(20)
```

[9]:
```
15.0
```

[10]:
```
a(30)
```

[10]:
```
20.0
```

Now, instead of storing a list and reclaculating `total` and `count` every time wer need the new average, we are going to store the running total and count and update each value each time a new value is added to the running average, and then return `total / count`.

Let's start with a class approach first, where we will use instance variables to store the running total and count and provide an instance method to add a new number and return the current average.

[11]:
```
class Averager:
    def __init__(self):
        self._count = 0
        self._total = 0

    def add(self, value):
        self._total += value
        self._count += 1
        return self._total / self._count
```

[12]:
```
a = Averager()
```

[13]:
```
a.add(10)
```

[13]:
```
10.0
```

[14]:
```
a.add(20)
```

[14]:
```
15.0
```

[15]:
```
a.add(30)
```

[15]:
```
20.0
```

Now, let's see how we might use a closure to achieve the same thing.

[16]:
```
def averager():
    total = 0
    count = 0

    def add(value):
        nonlocal total, count
        total += value
        count += 1
        return 0 if count == 0 else total / count

    return add
```

[17]:
```
a = averager()
```

[18]:
```
a(10)
```

[18]:
```
10.0
```

[19]:
```
a(20)
```

[19]:
```
15.0
```

[20]:
```
a(30)
```

[20]:
```
20.0
```

Generalizing this example We saw that we were essentially able to convert a class to an equivalent functionality using closures. This is actually true in a much more general sense - very often, classes that define a single method (other than initializers) can be implemented using a closure instead.

Let's look at another example of this.

Suppose we want something that can keep track of the running elapsed time in seconds.

[21]:
```
from time import perf_counter
```

[22]:
```
class Timer:
    def __init__(self):
        self._start = perf_counter()

    def __call__(self):
        return (perf_counter() - self._start)
```

[23]:
```
a = Timer()
```

Now wait a bit before running the next line of code:

[24]:
```
a()
```

[24]:
```
0.011695334544051804
```

Let's start another "timer":

[25]:
```
b = Timer()
```

[26]:
```
print(a())
print(b())
```

```
0.035282944039667645
0.011656054820407689
```

Now let's rewrite this using a closure instead:

[27]:
```
def timer():
    start = perf_counter()

    def elapsed():
        # we don't even need to make start nonlocal
        # since we are only reading it
        return perf_counter() - start

    return elapsed
```

[28]:
```
x = timer()
```

[29]:
```
x()
```

[29]: 0.011068213438975016

[30]:
```
y = timer()
```

[31]:
```
print(x())
print(y())
```

```
0.03419096772236116
0.01164738619174141
```

[32]:
```
print(a())
print(b())
print(x())
print(y())
```

```
0.10822159832175349
0.08475345336494494
0.0462381944113351
0.023573252079387305
```

[ ]:
```

```

0.0.7 Closure Applications (Part 2)

Example 1 Let's write a small function that can increment a counter for us - we don't have an incrementor in Python (the ++ operator in Java or C++ for example):

[2]:
```
def counter(initial_value):
    # initial_value is a local variable here

    def inc(increment=1):
        nonlocal initial_value
        # initial_value is a nonlocal (captured) variable here
        initial_value += increment
        return initial_value

    return inc
```

[3]:
```
counter1 = counter(0)
```

[4]:
```
print(counter1(0))
```

```
0
```

[5]:
```
print(counter1())
```

```
1
```

[6]:
```
print(counter1())
```

```
2
```

[7]:
```
print(counter1(8))
```

```
10
```

[8]:
```
counter2 = counter(1000)
```

[9]:
```
print(counter2(0))
```

```
1000
```

[10]:
```
print(counter2(1))
```

```
1001
```

[11]:
```
print(counter2())
```

```
1002
```

[12]:
```
print(counter2(220))
```

```
1222
```

As you can see, each closure maintains a reference to the **initial_value** variable that was created when the **counter** function was **called** - each time that function was called, a new local variable **initial_value** was created (with a value assigned from the argument), and it became a nonlocal (captured) variable in the inner scope.

Example 2 Let's modify this example to now build something that can run, and maintain a count of how many times we have run some function.

[13]:
```
def counter(fn):
    cnt = 0  # initially fn has been run zero times

    def inner(*args, **kwargs):
        nonlocal cnt
        cnt = cnt + 1
        print('{0} has been called {1} times'.format(fn.__name__, cnt))
        return fn(*args, **kwargs)

    return inner
```

[14]:
```
def add(a, b):
    return a + b
```

[15]:
```
counted_add = counter(add)
```

And the free variables are:

[16]:
```
counted_add.__code__.co_freevars
```

[16]:
```
('cnt', 'fn')
```

We can now call the `counted_add` function:

[17]:
```
counted_add(1, 2)
```

```
add has been called 1 times
```

[17]:
```
3
```

[18]:
```
counted_add(2, 3)
```

```
add has been called 2 times
```

[18]:
```
5
```

[19]:
```
def mult(a, b, c):
    return a * b * c
```

[20]:
```
counted_mult = counter(mult)
```

[21]:
```
counted_mult(1, 2, 3)
```

```
mult has been called 1 times
```

[21]:
```
6
```

[22]:
```
counted_mult(2, 3, 4)
```

```
mult has been called 2 times
```

[22]:
```
24
```

Example 3 Let's take this one step further, and actually store the function name and the number of calls in a global dictionary instead of just printing it out all the time.

[35]:
```
counters = dict()

def counter(fn):
    cnt = 0  # initially fn has been run zero times

    def inner(*args, **kwargs):
        nonlocal cnt
        cnt = cnt + 1
        counters[fn.__name__] = cnt  # counters is global
        return fn(*args, **kwargs)

    return inner
```

[26]:
```
counted_add = counter(add)
counted_mult = counter(mult)
```

Note that `counters` is a **global** variable, and therefore **not** a free variable:

[27]:
```
counted_add.__code__.co_freevars
```

[27]:
```
('cnt', 'fn')
```

[28]:
```
counted_mult.__code__.co_freevars
```

[28]:
```
('cnt', 'fn')
```

We can now call out functions:

[29]:
```
counted_add(1, 2)
```

[29]:
```
3
```

[30]:
```
counted_add(2, 3)
```

[30]:
```
5
```

[31]:
```
counted_mult(1, 2, 'a')
```

[31]:
```
'aa'
```

[32]:
```
counted_mult(2, 3, 'b')
```

[32]:
```
'bbbbbb'
```

[33]:
```
counted_mult(1, 1, 'abc')
```

[33]:
```
'abc'
```

[34]:
```
print(counters)
```

```
{'add': 2, 'mult': 3}
```

Of course this relies on us creating the **counters** global variable first and making sure we are naming it that way, so instead, we're going to pass it as an argument to the **counter** function:

[36]:
```
def counter(fn, counters):
    cnt = 0  # initially fn has been run zero times

    def inner(*args, **kwargs):
        nonlocal cnt
        cnt = cnt + 1
        counters[fn.__name__] = cnt  # counters is nonlocal
        return fn(*args, **kwargs)

    return inner
```

[33]:
```
func_counters = dict()
counted_add = counter(add, func_counters)
counted_mult = counter(mult, func_counters)
```

[34]:
```
counted_add.__code__.co_freevars
```

[34]:
```
('cnt', 'counters', 'fn')
```

As you can see, `counters` is now a free variable.

We can now call our functions:

[35]:
```
for i in range(5):
    counted_add(i, i)

for i in range(10):
    counted_mult(i, i, i)
```

[36]:
```
print(func_counters)
```

```
{'add': 5, 'mult': 10}
```

Of course, we don't have to assign the "counted" version of our functions a new name - we can simply assign it to the same name!

[37]:
```
def fact(n):
    product = 1
    for i in range(2, n+1):
        product *= i
    return product
```

[38]:
```
fact = counter(fact, func_counters)
```

[39]:
```
fact(0)
```

[39]:
```
1
```

[40]:
```
fact(3)
```

[40]: 6

[41]:
```python
fact(4)
```

[41]: 24

[42]:
```python
print(func_counters)
```

```
{'add': 5, 'mult': 10, 'fact': 3}
```

Notice, how we essentially **added** some functionality to our `fact` function, without modifying what the `fact` function actually returns.

This leads us straight into our next topic: decorators!

[ ]:
```python

```

0.0.8 Decorators (Part 1)

Recall the example in the last section where we wrote a simple closure to count how many times a function had been run:

[1]:
```python
def counter(fn):
    count = 0

    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print('Function {0} was called {1} times'.format(fn.__name__, count))
        return fn(*args, **kwargs)
    return inner
```

[2]:
```python
def add(a, b=0):
    """
    returns the sum of a and b
    """
    return a + b
```

[3]:
```python
help(add)
```

```
Help on function add in module __main__:

add(a, b=0)
    returns the sum of a and b
```

Here's the memory address that `add` points to:

[4]:
```python
id(add)
```

[4]: 2352389334696

Now we create a closure using the `add` function as an argument to the `counter` function:

[5]:
```
add = counter(add)
```

And you'll note that `add` is no longer the same function as before. Indeed the memory address `add` points to is no longer the same:

[6]:
```
id(add)
```

[6]: 2352404346128

[7]:
```
add(1, 2)
```

```
Function add was called 1 times
```

[7]: 3

[8]:
```
add(2, 2)
```

```
Function add was called 2 times
```

[8]: 4

What happened is that we put our **add** function 'through' the **counter** function - we usually say that we **decorated** our function **add**.

And we call that **counter** function a **decorator**.

There is a shorthand way of decorating our function without having to type:

`func = counter(func)`

[9]:
```
@counter
def mult(a: float, b: float=1, c: float=1) -> float:
    """
    returns the product of a, b, and c
    """
    return a * b * c
```

[10]:
```
mult(1, 2, 3)
```

```
Function mult was called 1 times
```

[10]: 6

[11]:
```
mult(2, 2, 2)
```

```
Function mult was called 2 times
```

```
[11]: 8
```

Let's do a little bit of introspection on our two decorated functions:

```
[12]: add.__name__
```

```
[12]: 'inner'
```

```
[13]: mult.__name__
```

```
[13]: 'inner'
```

As you can see, the name of the function is no longer **add** or **mult**, but instead it is the name of that **inner** function in our decorator.

```
[14]: help(add)
```

```
Help on function inner in module __main__:

inner(*args, **kwargs)
```

```
[15]: help(mult)
```

```
Help on function inner in module __main__:

inner(*args, **kwargs)
```

As you can see, we've also lost our docstring and parameter annotations!

What about introspecting the parameters of **add** and **mult**:

```
[16]: import inspect
```

```
[17]: inspect.getsource(add)
```

```
[17]: "    def inner(*args, **kwargs):\n        nonlocal count\n        count += 1\n
print('Function {0} was called {1} times'.format(fn.__name__, count))\n
return fn(*args, **kwargs)\n"
```

```
[18]: inspect.getsource(mult)
```

```
[18]: "    def inner(*args, **kwargs):\n        nonlocal count\n        count += 1\n
print('Function {0} was called {1} times'.format(fn.__name__, count))\n
return fn(*args, **kwargs)\n"
```

Even the signature is gone:

```
[19]: inspect.signature(add)
```

[19]:
```
<Signature (*args, **kwargs)>
```

[20]:
```
inspect.signature(mult)
```

[20]:
```
<Signature (*args, **kwargs)>
```

Even the parameter defaults documentation is are gone:

[21]:
```
inspect.signature(add).parameters
```

[21]:
```
mappingproxy({'args': <Parameter "*args">, 'kwargs': <Parameter "**kwargs">})
```

In general, when we create decorated functions, we end up “losing” a lot of the metadata of our original function!

However, we **can** put that information back in - it can get quite complicated.

Let’s see how we might be able to do that for some simple things, like the docstring and the function name.

[22]:
```
def counter(fn):
    count = 0

    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print("{0} was called {1} times".format(fn.__name__, count))
    inner.__name__ = fn.__name__
    inner.__doc__ = fn.__doc__
    return inner
```

[23]:
```
@counter
def add(a: int, b: int=10) -> int:
    """
    returns sum of two integers
    """
    return a + b
```

[24]:
```
help(add)
```

```
Help on function add in module __main__:

add(*args, **kwargs)
    returns sum of two integers
```

[25]:
```
add.__name__
```

[25]:
```
'add'
```

At least we have the docstring and function name back... But what about the parameters? Our real **add** function takes two positional parameters, but because the closure used a generic way of accepting ***args** and ****kwargs**, we lose this information

We can use a special function in the **functools** module, called **wraps**. In fact, that function is a decorator itself!

[26]:
```
from functools import wraps
```

[27]:
```
def counter(fn):
    count = 0

    @wraps(fn)
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print("{0} was called {1} times".format(fn.__name__, count))

    return inner
```

[28]:
```
@counter
def add(a: int, b: int=10) -> int:
    """
    returns sum of two integers
    """
    return a + b
```

[29]:
```
help(add)
```

```
Help on function add in module __main__:

add(a:int, b:int=10) -> int
    returns sum of two integers
```

Yay!!! Everything is back to normal.

[30]:
```
inspect.getsource(add)
```

[30]:
```
'@counter\ndef add(a: int, b: int=10) -> int:\n    """\n    returns sum of two
integers\n    """\n    return a + b\n'
```

[31]:
```
inspect.signature(add)
```

[31]:
```
<Signature (a:int, b:int=10) -> int>
```

[32]:
```
inspect.signature(add).parameters
```

[32]:
```
mappingproxy({'a': <Parameter "a:int">, 'b': <Parameter "b:int=10">})
```

0.0.9 Decorators Application (Timing)

Here we go back to an example we have seen in the past - timing how long it takes to run a certain function.

[2]:
```
def timed(fn):
    from time import perf_counter
    from functools import wraps

    @wraps(fn)
    def inner(*args, **kwargs):
        start = perf_counter()
        result = fn(*args, **kwargs)
        end = perf_counter()
        elapsed = end - start

        args_ = [str(a) for a in args]
        kwargs_ = ['{0}={1}'.format(k, v) for (k, v) in kwargs.items()]
        all_args = args_ + kwargs_
        args_str = ','.join(all_args)
        print('{0}({1}) took {2:.6f}s to run.'.format(fn.__name__,
                                                      args_str,
                                                      elapsed))

        return result

    return inner
```

Let's write a function that calculates the n-th Fibonacci number:

`1, 1, 2, 3, 5, 8, ...`

We will implement this using three different methods: 1. recursion 2. a loop 3. functional programming (reduce)

We use a 1-based system, e.g. first Fibonnaci number has index 1, etc.

Using Recursion

[3]:
```
def calc_recursive_fib(n):
    if n <=2:
        return 1
    else:
        return calc_recursive_fib(n-1) + calc_recursive_fib(n-2)
```

[4]:
```
calc_recursive_fib(3)
```

[4]:
```
2
```

[5]:
```
calc_recursive_fib(6)
```

[5]: 8

[6]:
```
@timed
def fib_recursed(n):
    return calc_recursive_fib(n)
```

[7]:
```
fib_recursed(33)
```

```
fib_recursed(33) took 1.060477s to run.
```

[7]: 3524578

[8]:
```
fib_recursed(34)
```

```
fib_recursed(34) took 1.715229s to run.
```

[8]: 5702887

[9]:
```
fib_recursed(35)
```

```
fib_recursed(35) took 2.773638s to run.
```

[9]: 9227465

There's a reason we did not decorate our recursive function directly!

[10]:
```
@timed
def fib_recursed_2(n):
    if n <=2:
        return 1
    else:
        return fib_recursed_2(n-1) + fib_recursed_2(n-2)
```

[11]:
```
fib_recursed_2(10)
```

```
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000001s to run.
fib_recursed_2(3) took 0.000409s to run.
fib_recursed_2(2) took 0.000001s to run.
fib_recursed_2(4) took 0.000460s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000038s to run.
fib_recursed_2(5) took 0.000535s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000038s to run.
fib_recursed_2(2) took 0.000000s to run.
```

```
fib_recursed_2(4) took 0.000075s to run.
fib_recursed_2(6) took 0.000646s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(2) took 0.000001s to run.
fib_recursed_2(4) took 0.000071s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000035s to run.
fib_recursed_2(5) took 0.000143s to run.
fib_recursed_2(7) took 0.000837s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000001s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(4) took 0.000072s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(5) took 0.000142s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000037s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(4) took 0.000073s to run.
fib_recursed_2(6) took 0.000251s to run.
fib_recursed_2(8) took 0.001125s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000041s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(4) took 0.000076s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000035s to run.
fib_recursed_2(5) took 0.000146s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000001s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(4) took 0.000072s to run.
fib_recursed_2(6) took 0.000253s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000001s to run.
fib_recursed_2(3) took 0.000048s to run.
fib_recursed_2(2) took 0.000001s to run.
fib_recursed_2(4) took 0.000085s to run.
```

```
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(5) took 0.000156s to run.
fib_recursed_2(7) took 0.000444s to run.
fib_recursed_2(9) took 0.001604s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000001s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(2) took 0.000001s to run.
fib_recursed_2(4) took 0.000071s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(5) took 0.000142s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(4) took 0.000071s to run.
fib_recursed_2(6) took 0.000248s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000040s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(4) took 0.000075s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000036s to run.
fib_recursed_2(5) took 0.000145s to run.
fib_recursed_2(7) took 0.000429s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000035s to run.
fib_recursed_2(2) took 0.000001s to run.
fib_recursed_2(4) took 0.000075s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000035s to run.
fib_recursed_2(5) took 0.000145s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(1) took 0.000000s to run.
fib_recursed_2(3) took 0.000041s to run.
fib_recursed_2(2) took 0.000000s to run.
fib_recursed_2(4) took 0.000076s to run.
fib_recursed_2(6) took 0.000256s to run.
fib_recursed_2(8) took 0.000720s to run.
fib_recursed_2(10) took 0.002367s to run.
```

```
[11]: 55
```

Since we are calling the function recursively, we are actually calling the **decorated** function recursively. In this case I wanted the total time to calculate the n-th number, not the time for each recursion.

You will notice from the above how inefficient the recursive method is: the same fibonacci numbers are calculated repeatedly! This is why as the value of `n` start increasing beyond 30 we start seeing considerable slow downs.

Using a Loop

```
[12]: @timed
      def fib_loop(n):
          fib_1 = 1
          fib_2 = 1
          for i in range(3, n+1):
              fib_1, fib_2 = fib_2, fib_1 + fib_2
          return fib_2
```

```
[13]: fib_loop(3)
```

```
fib_loop(3) took 0.000003s to run.
```

```
[13]: 2
```

```
[14]: fib_loop(6)
```

```
fib_loop(6) took 0.000002s to run.
```

```
[14]: 8
```

```
[15]: fib_loop(34)
```

```
fib_loop(34) took 0.000004s to run.
```

```
[15]: 5702887
```

```
[16]: fib_loop(35)
```

```
fib_loop(35) took 0.000005s to run.
```

```
[16]: 9227465
```

As you can see this method is much more efficient!

Using Reduce We first need to understand how we are going to calculate the Fibonnaci sequence using reduce:

In general each step in the reduction is as follows:

If we start our reduction with an initial value of `(1, 0)`, we need to run our "loop" n times.

We therefore use a "dummy" sequence of length `n` to create `n` steps in our reduce.

[33]:
```python
from functools import reduce

@timed
def fib_reduce(n):
    initial = (1, 0)
    dummy = range(n-1)
    fib_n = reduce(lambda prev, n: (prev[0] + prev[1], prev[0]),
                   dummy,
                   initial)
    return fib_n[0]
```

[34]:
```python
fib_reduce(3)
```

```
fib_reduce(3) took 0.000004s to run.
```

[34]:
```
2
```

[35]:
```python
fib_reduce(6)
```

```
fib_reduce(6) took 0.000005s to run.
```

[35]:
```
8
```

[36]:
```python
fib_reduce(34)
```

```
fib_reduce(34) took 0.000013s to run.
```

[36]:
```
5702887
```

[37]:
```python
fib_reduce(35)
```

```
fib_reduce(35) took 0.000014s to run.
```

[37]:
```
9227465
```

Now we can run a quick comparison between the various timed implementations:

[22]:
```python
fib_recursed(35)
fib_loop(35)
fib_reduce(35)
```

```
fib_recursed(35) took 2.771373s to run.
fib_loop(35) took 0.000007s to run.
fib_reduce(35) took 0.000013s to run.
```

[22]:
```
9227465
```

Even though the recursive algorithm is by far the easiest to understand, it is also the slowest. We'll see how to fix this in an upcoming video using a technique called **memoization**.

First let's focus on the loop and reduce variants. Our timing is not very effective since we only time a single calculation for each - there could be some variance if we run these tests multiple times:

[23]:
```
for i in range(10):
    result =  fib_loop(10000)
```

```
fib_loop(10000) took 0.002114s to run.
fib_loop(10000) took 0.002109s to run.
fib_loop(10000) took 0.002072s to run.
fib_loop(10000) took 0.002072s to run.
fib_loop(10000) took 0.002075s to run.
fib_loop(10000) took 0.002078s to run.
fib_loop(10000) took 0.002049s to run.
fib_loop(10000) took 0.002064s to run.
fib_loop(10000) took 0.002533s to run.
fib_loop(10000) took 0.002109s to run.
```

[24]:
```
for i in range(10):
    result = fib_reduce(10000)
```

```
fib_reduce(10000) took 0.004234s to run.
fib_reduce(10000) took 0.003961s to run.
fib_reduce(10000) took 0.004363s to run.
fib_reduce(10000) took 0.004459s to run.
fib_reduce(10000) took 0.003895s to run.
fib_reduce(10000) took 0.003847s to run.
fib_reduce(10000) took 0.004342s to run.
fib_reduce(10000) took 0.003908s to run.
fib_reduce(10000) took 0.003970s to run.
fib_reduce(10000) took 0.003970s to run.
```

In general it is better to time the same function call multiple times and generate and average of the run times.

We'll see in an upcoming video how we can do this from within our decorator.

In the meantime observe that the simple loop approach seems to perform about twice as fast as the reduce approach!!

The moral of this side note is that simply because you **can** do something in Python using some fancy or cool technique does not mean you **should**!

We technically could write our reduce-based function as a one liner:

[25]:
```
from functools import reduce
```

```
fib_1 = timed(lambda n: reduce(lambda prev, n: (prev[0] + prev[1], prev[0]),
                               range(n),
                               (0, 1))[0])
```

[26]:
```
fib_loop(100)
```

```
fib_loop(100) took 0.000009s to run.
```

[26]:
```
354224848179261915075
```

[27]:
```
fib_1(100)
```

```
<lambda>(100) took 0.000031s to run.
```

[27]:
```
354224848179261915075
```

So yes, it's cool that you can write this using a single line of code, but consider two things here: 1. Is it as efficient as another method? 2. Is the code **readable**?

Code readability is something I cannot emphasize enough. Given similar efficiencies (cpu / memory), give preference to code that is more easily understandable!

Sometimes, if the efficiency is not greatly impacted (or does not matter in absolute terms), I might even give preference to less efficient, but more readable (i.e. understanbdable), code.

But enough of the soapbox already :-)

0.0.10 Decorators Application (Logger, Stacked Decorators)

In this example we're going to create a utility decorator that will log function calls (to the console, but in practice you would be writing your logs to a file (e.g. using Python's built-in logger), or to a database, etc.

[1]:
```
def logged(fn):
    from functools import wraps
    from datetime import datetime, timezone

    @wraps(fn)
    def inner(*args, **kwargs):
        run_dt = datetime.now(timezone.utc)
        result = fn(*args, **kwargs)
        print('{0}: called {1}'.format(fn.__name__, run_dt))
        return result

    return inner
```

[2]:
```
@logged
def func_1():
    pass
```

[3]:
```
@logged
def func_2():
    pass
```

[4]:
```
func_1()
```

```
func_1: called 2017-12-10 00:09:19.443657+00:00
```

[5]:
```
func_2()
```

```
func_2: called 2017-12-10 00:09:19.460691+00:00
```

Now we may additionaly also want to time the function. We can certainly include the code to do so in our `logged` decorator, but we could also just use the `@timed` decorator we already wrote by **stacking** our decorators.

[6]:
```
def timed(fn):
    from functools import wraps
    from time import perf_counter

    @wraps(fn)
    def inner(*args, **kwargs):
        start = perf_counter()
        result = fn(*args, **kwargs)
        end = perf_counter()
        print('{0} ran for {1:.6f}s'.format(fn.__name__, end-start))
        return result

    return inner
```

[7]:
```
@timed
@logged
def factorial(n):
    from operator import mul
    from functools import reduce

    return reduce(mul, range(1, n+1))
```

[8]:
```
factorial(10)
```

```
factorial: called 2017-12-10 00:09:19.496762+00:00
factorial ran for 0.000130s
```

[8]:
```
3628800
```

Note that the order in which we stack the decorators can make a difference!

Remember that this is because our stacked decorators essentially amounted to:

[9]:
```
def factorial(n):
    from operator import mul
    from functools import reduce

    return reduce(mul, range(1, n+1))

factorial = timed(logged(factorial))
```

So in this case the `timed` decorator will be called first, followed by the `logged` decorator.

You may wonder why the printed output seems reversed. Look at how the decorators were defined - they first ran the function passed in, and **then** printed the result.

So in the above example, a simplified look at what happens in each decorator:

- `timed(fn)(*args, **kwargs)`:
  1. calls `fn(*args, **kwargs)`
  2. prints timing
- `logged(fn)(*args, **kwargs)`:
  1. calls `fn(*args, **kwargs)`
  2. prints log info

So, calling `factorial = timed(logged(factorial))`

is equivalent to:

So as you can see, the `timed` decorator ran first, but it called the logged decorated function first, then printed the result - hence why the print output seems reversed.

[10]:
```
factorial(10)
```

```
factorial: called 2017-12-10 00:09:19.525820+00:00
factorial ran for 0.000147s
```

[10]:
```
3628800
```

But in the following case, the `logged` decorator will run first, followed by the `timed` decorator:

[11]:
```
def factorial(n):
    from operator import mul
    from functools import reduce

    return reduce(mul, range(1, n+1))

factorial = logged(timed(factorial))
```

[12]:
```
factorial(10)
```

```
factorial ran for 0.000015s
factorial: called 2017-12-10 00:09:19.547866+00:00
```

[12]: 3628800

Or, using the **@** notation:

[13]:
```
@logged
@timed
def factorial(n):
    from operator import mul
    from functools import reduce

    return reduce(mul, range(1, n+1))
```

[14]:
```
factorial(10)
```

```
factorial ran for 0.000016s
factorial: called 2017-12-10 00:09:19.572914+00:00
```

[14]: 3628800

[15]:
```
@timed
@logged
def factorial(n):
    from operator import mul
    from functools import reduce

    return reduce(mul, range(1, n+1))
```

[16]:
```
factorial(10)
```

```
factorial: called 2017-12-10 00:09:19.608237+00:00
factorial ran for 0.000153s
```

[16]: 3628800

To make this clearer, let's write two very simple decorators as follows:

[17]:
```
def dec_1(fn):
    def inner():
        print('running dec_1')
        return fn()
    return inner
```

[18]:
```
def dec_2(fn):
    def inner():
        print('running dec_2')
        return fn()
    return inner
```

[19]:
```python
@dec_1
@dec_2
def my_func():
    print('running my_func')
```

[20]:
```python
my_func()
```

```
running dec_1
running dec_2
running my_func
```

But if we change the order of the decorators:

[21]:
```python
@dec_2
@dec_1
def my_func():
    print('running my_func')
```

[22]:
```python
my_func()
```

```
running dec_2
running dec_1
running my_func
```

You may wonder whether this really matters in practice. And yes, it can.

Consider an API that contains various functions that can be called. However, endpoints are secured and can only be run by authenticated users who have some specific role(s). If they do not have the role you want to return an unauthorized error. But if they do, then you want to log that they called the endpoint.

In this case you may have one decorator that is used to check authentication and permissions (and immediately return an unauthorized error from the API if applicable), and the other to log the call.

If you decorated it this way:

then the call would always be logged.

But, in this instance:

your endpoint would only get logged if the user passed the `authorize` test.

0.0.11 Decorators Application (Memoization)

Let's go back to our Fibonacci example:

[1]:
```python
def fib(n):
    print ('Calculating fib({0})'.format(n))
    return 1 if n < 3 else fib(n-1) + fib(n-2)
```

When we run this, we see that it is quite inefficient, as the same Fibonacci numbers get calculated multiple times:

[2]:
```
fib(6)
```

```
Calculating fib(6)
Calculating fib(5)
Calculating fib(4)
Calculating fib(3)
Calculating fib(2)
Calculating fib(1)
Calculating fib(2)
Calculating fib(3)
Calculating fib(2)
Calculating fib(1)
Calculating fib(4)
Calculating fib(3)
Calculating fib(2)
Calculating fib(1)
Calculating fib(2)
```

[2]: 8

It would be better if we could somehow "store" these results, so if we have calculated `fib(4)` and `fib(3)` before, we could simply recall the these values when calculating `fib(5) = fib(4) + fib(3)` instead of recalculating them.

This concept of improving the efficiency of our code by caching pre-calculated values so they do not need to be re-calcualted every time, is called "memoization"

We can approach this using a simple class and a dictionary that stores any Fibonacci number that's already been calculated:

[3]:
```
class Fib:
    def __init__(self):
        self.cache = {1: 1, 2: 1}

    def fib(self, n):
        if n not in self.cache:
            print('Calculating fib({0})'.format(n))
            self.cache[n] = self.fib(n-1) + self.fib(n-2)
        return self.cache[n]
```

[4]:
```
f = Fib()
```

[5]:
```
f.fib(1)
```

[5]: 1

[6]:
```
f.fib(6)
```

```
Calculating fib(6)
Calculating fib(5)
Calculating fib(4)
Calculating fib(3)
```

```
[6]: 8
```

```
[7]: f.fib(7)
```

```
Calculating fib(7)
```

```
[7]: 13
```

Let’s see how we could rewrite this using a closure:

```
[8]: def fib():
         cache = {1: 1, 2: 2}

         def calc_fib(n):
             if n not in cache:
                 print('Calculating fib({0})'.format(n))
                 cache[n] = calc_fib(n-1) + calc_fib(n-2)
             return cache[n]

         return calc_fib
```

```
[9]: f = fib()
```

```
[10]: f(10)
```

```
Calculating fib(10)
Calculating fib(9)
Calculating fib(8)
Calculating fib(7)
Calculating fib(6)
Calculating fib(5)
Calculating fib(4)
Calculating fib(3)
```

```
[10]: 89
```

Now let’s see how we would implement this using a decorator:

```
[11]: from functools import wraps

      def memoize_fib(fn):
          cache = dict()

          @wraps(fn)
```

```
    def inner(n):
        if n not in cache:
            cache[n] = fn(n)
        return cache[n]

    return inner
```

[12]:
```
@memoize_fib
def fib(n):
    print ('Calculating fib({0})'.format(n))
    return 1 if n < 3 else fib(n-1) + fib(n-2)
```

[13]:
```
fib(3)
```

```
Calculating fib(3)
Calculating fib(2)
Calculating fib(1)
```

[13]:
```
2
```

[14]:
```
fib(10)
```

```
Calculating fib(10)
Calculating fib(9)
Calculating fib(8)
Calculating fib(7)
Calculating fib(6)
Calculating fib(5)
Calculating fib(4)
```

[14]:
```
55
```

[15]:
```
fib(6)
```

[15]:
```
8
```

As you can see, we are hitting the cache when the values are available.

Now, we made our memoization decorator "hardcoded" to single argument functions - we could make it more generic.

For example, to handle an arbitrary number of positional arguments and keyword-only arguments we could do the following:

[44]:
```
def memoize(fn):
    cache = dict()

    @wraps(fn)
    def inner(*args):
        if args not in cache:
            cache[args] = fn(*args)
        return cache[args]

    return inner
```

```
[17]: @memoize
      def fib(n):
          print ('Calculating fib({0})'.format(n))
          return 1 if n < 3 else fib(n-1) + fib(n-2)
```

```
[18]: fib(6)
```

```
Calculating fib(6)
Calculating fib(5)
Calculating fib(4)
Calculating fib(3)
Calculating fib(2)
Calculating fib(1)
```

```
[18]: 8
```

```
[19]: fib(7)
```

```
Calculating fib(7)
```

```
[19]: 13
```

Of course, with this rather generic memoization decorator we can memoize other functions too:

```
[20]: def fact(n):
          print('Calculating {0}!'.format(n))
          return 1 if n < 2 else n * fact(n-1)
```

```
[21]: fact(5)
```

```
Calculating 5!
Calculating 4!
Calculating 3!
Calculating 2!
Calculating 1!
```

```
[21]: 120
```

```
[22]: fact(5)
```

```
Calculating 5!
Calculating 4!
Calculating 3!
Calculating 2!
Calculating 1!
```

[22]:
```
120
```

And memoizing it:

[23]:
```python
@memoize
def fact(n):
    print('Calculating {0}!'.format(n))
    return 1 if n < 2 else n * fact(n-1)
```

[24]:
```python
fact(6)
```

```
Calculating 6!
Calculating 5!
Calculating 4!
Calculating 3!
Calculating 2!
Calculating 1!
```

[24]:
```
720
```

[25]:
```python
fact(6)
```

[25]:
```
720
```

Our simple memoizer has a drawback however: * the cache size is unbounded - probably not a good thing! In general we want to limit the cache to a certain number of entries, balancing computational efficiency vs memory utilization. * we are not handling **kwargs

Memoization is such a common thing to do that Python actually has a memoization decorator built for us!

It's in the, you guessed it, **functools** module, and is called **lru_cache** and is going to be quite a bit more efficient compared to the rudimentary memoization example we did above.

[LRU Cache = Least Recently Used caching: since the cache is not unlimited, at some point cached entries need to be discarded, and the least recently used entries are discarded first]

[26]:
```python
from functools import lru_cache
```

[27]:
```python
@lru_cache()
def fact(n):
    print("Calculating fact({0})".format(n))
    return 1 if n < 2 else n * fact(n-1)
```

[28]:
```python
fact(5)
```

```
Calculating fact(5)
Calculating fact(4)
Calculating fact(3)
Calculating fact(2)
Calculating fact(1)
```

[28]:
```
120
```

[29]:
```python
fact(4)
```

[29]:
```
24
```

As you can see, `fact(4)` was returned via a cached entry!

Same thing with our Fibonacci function:

[30]:
```python
@lru_cache()
def fib(n):
    print("Calculating fib({0})".format(n))
    return 1 if n < 3 else fib(n-1) + fib(n-2)
```

[31]:
```python
fib(6)
```

```
Calculating fib(6)
Calculating fib(5)
Calculating fib(4)
Calculating fib(3)
Calculating fib(2)
Calculating fib(1)
```

[31]:
```
8
```

[32]:
```python
fib(5)
```

[32]:
```
5
```

Recall from a few videos back that we timed the calculation for Fibonacci numbers. Calculating fib(35) took several seconds - every time...

[33]:
```python
from time import perf_counter
```

[34]:
```python
def fib_no_memo(n):
    return 1 if n < 3 else fib_no_memo(n-1) + fib_no_memo(n-2)
```

[35]:
```python
start = perf_counter()
result = fib_no_memo(35)
print("result={0}, elapsed: {1}s".format(result, perf_counter() - start))
```

```
result=9227465, elapsed: 2.939012289158911s
```

[36]:
```
@lru_cache()
def fib_memo(n):
    return 1 if n < 3 else fib_memo(n-1) + fib_memo(n-2)
```

[37]:
```
start = perf_counter()
result = fib_memo(35)
print("result={0}, elapsed: {1}s".format(result, perf_counter() - start))
```

```
result=9227465, elapsed: 9.83349429017899e-05s
```

And if we make the calls again:

[38]:
```
start = perf_counter()
result = fib_no_memo(35)
print("result={0}, elapsed: {1}s".format(result, perf_counter() - start))
```

```
result=9227465, elapsed: 2.782454120518548s
```

[39]:
```
start = perf_counter()
result = fib_memo(35)
print("result={0}, elapsed: {1}s".format(result, perf_counter() - start))
```

```
result=9227465, elapsed: 5.66170883375960440e-05s
```

You may have noticed that the `lru_cache` decorator was implemented using `()` - we'll see more on this later, but that's because decorators can themselves have parameters (beyond the function being decorated).

One of the arguments to the `lru_cache` decorator is the size of the cache - it defaults to 128 items, but we can easily change this - for performance reasons use powers of 2 for the cache size (or None for unbounded cache):

[40]:
```
@lru_cache(maxsize=8)
def fib(n):
    print("Calculating fib({0})".format(n))
    return 1 if n < 3 else fib(n-1) + fib(n-2)
```

[41]:
```
fib(10)
```

```
Calculating fib(10)
Calculating fib(9)
Calculating fib(8)
Calculating fib(7)
Calculating fib(6)
Calculating fib(5)
Calculating fib(4)
Calculating fib(3)
Calculating fib(2)
Calculating fib(1)
```

```
[41]: 55
```

```
[42]: fib(20)
```

```
Calculating fib(20)
Calculating fib(19)
Calculating fib(18)
Calculating fib(17)
Calculating fib(16)
Calculating fib(15)
Calculating fib(14)
Calculating fib(13)
Calculating fib(12)
Calculating fib(11)
```

```
[42]: 6765
```

```
[43]: fib(10)
```

```
Calculating fib(10)
Calculating fib(9)
Calculating fib(8)
Calculating fib(7)
Calculating fib(6)
Calculating fib(5)
Calculating fib(4)
Calculating fib(3)
Calculating fib(2)
Calculating fib(1)
```

```
[43]: 55
```

Note how Python had to recalculate `fib` for `10, 9,` etc. This is because the cache can only contain 10 items, so when we calculated `fib(20)`, it stored fib for `20, 19, ..., 11` (10 items) and therefore the oldest items fib `10, 9, ..., 1` were removed from the cache to make space.

```
[ ]:
```

0.0.12 Decorators 2

We have seen how to create some simple and not so simple decorators.

However we have also been using built-in decorators that can accept parameters, such as `wraps` and `lru_cache`.

This can be quite useful and we can accomplish the same thing ourselves.

First recall our original timer decorator from an earlier video (Decorator Application - Timer):

[1]:
```
def timed(fn):
    from time import perf_counter

    def inner(*args, **kwargs):
        start = perf_counter()
        result = fn(*args, **kwargs)
        end = perf_counter()
        elapsed = end - start
        print('Run time: {0:.6f}s'.format(elapsed))
        return result

    return inner
```

[2]:
```
def calc_fib_recurse(n):
    return 1 if n < 3 else calc_fib_recurse(n-1) + calc_fib_recurse(n-2)

def fib(n):
    return calc_fib_recurse(n)
```

We can decorate our Fibonacci function using the **@** syntax, or the longer syntax as follows:

[3]:
```
fib = timed(fib)
```

[4]:
```
fib(30)
```

```
Run time: 0.255260s
```

[4]:
```
832040
```

Let's modify this so the timer runs the function multiple times and calculates the average run time:

[5]:
```
def timed(fn):
    from time import perf_counter

    def inner(*args, **kwargs):
        total_elapsed = 0
        for i in range(10):
            start = perf_counter()
            result = fn(*args, **kwargs)
            end = perf_counter()
            total_elapsed += (perf_counter() - start)
        avg_elapsed = total_elapsed / 10
        print('Avg Run time: {0:.6f}s'.format(avg_elapsed))
        return result

    return inner
```

And again we decorate it using the long syntax:

[6]:
```python
def fib(n):
    return calc_fib_recurse(n)

fib = timed(fib)
```

[7]:
```python
fib(28)
```

```
Avg Run time: 0.098860s
```

[7]:
```
317811
```

But that value of 10 has been hardcoded. Let's make it a parameter instead.

[8]:
```python
def timed(fn, num_reps):
    from time import perf_counter

    def inner(*args, **kwargs):
        total_elapsed = 0
        for i in range(num_reps):
            start = perf_counter()
            result = fn(*args, **kwargs)
            end = perf_counter()
            total_elapsed += (perf_counter() - start)
        avg_elapsed = total_elapsed / num_reps
        print('Avg Run time: {0:.6f}s ({1} reps)'.format(avg_elapsed,
                                                        num_reps))
        return result

    return inner
```

Now to decorate our Fibonacci function we **have** to use the long syntax (as we saw in the lecture, the **@** syntax will not work):

[9]:
```python
def fib(n):
    return calc_fib_recurse(n)

fib = timed(fib, 5)
```

[10]:
```python
fib(28)
```

```
Avg Run time: 0.095708s (5 reps)
```

[10]:
```
317811
```

The problem is that we cannot use the `@` decorator syntax because when using that syntax Python passes a **single** argument to the decorator: the function we are decorating - nothing else.

Of course we could just use what we did above, but the decorator syntax is kind of neat, so it would be nice to retain the ability to use it.

We just need to change our thinking a little bit to do this:

First, when we see the following syntax:

`@dec def my_func():     pass`

we see that `dec` must be a function that takes a single argument, the function being decorated.

You'll note that `dec` is just a function, but we do not **call** `dec` when we decorate `my_func`, we simply use the label `dec`.

Then Python does:

`my_func = dec(my_func)`

Let's try a concrete example:

[11]:
```python
def dec(fn):
    print ("running dec")

    def inner(*args, **kwargs):
        print("running inner")
        return fn(*args, **kwargs)

    return inner
```

[12]:
```python
@dec
def my_func():
    print('running my_func')
```

```
running dec
```

As we can see, when we decorated `my_func`, the `dec` function was **called** at that time.

(Because Python did this:

`my_func = dec(my_func)`

so `dec` was called)

And when we now call `my_func`, we see that the `inner` function is called, followed by the original `my_func`

[13]:
```python
my_func()
```

```
running inner
running my_func
```

But what if `dec` was not the decorator itself, but instead created and returned a decorator?

Let's see how we might do this:

[14]:
```python
def dec_factory():
    print('running dec_factory')
    def dec(fn):
        print('running dec')
        def inner(*args, **kwargs):
            print('running inner')
            return fn(*args, **kwargs)
        return inner
    return dec
```

So as you can see, calling `dec_generator()` will return that `dec` function which is our decorator:

[15]:
```
@dec_factory()
def my_func(a, b):
    print(a, b)
```

```
running dec_factory
running dec
```

You can see that both `dec_generator` and `dec` were already called.

[16]:
```
my_func(10, 20)
```

```
running inner
10 20
```

And there you go, all we did is basically create a decorator by calling a function (`dec_factory`) and use the return value of that call (the `dec` function) as our actual decorator.

We could have done the decoration this way too:

[17]:
```
dec = dec_factory()
```

```
running dec_factory
```

[18]:
```
@dec
def my_func():
    print('running my_func')
```

```
running dec
```

[19]:
```
my_func()
```

```
running inner
running my_func
```

Or even this way:

[20]:
```
dec = dec_factory()

def my_func():
    print('running my_func')

```

```
my_func = dec(my_func)
```

```
running dec_factory
running dec
```

[21]:
```
my_func()
```

```
running inner
running my_func
```

Of course we could even decorate it this way using a single statement:

[22]:
```
def my_func():
    print('running my_func')

my_func = dec_factory()(my_func)
```

```
running dec_factory
running dec
```

[23]:
```
my_func()
```

```
running inner
running my_func
```

OK, so now we have decorated our function using, not a decorator, but a decorator factory as follows:

[24]:
```
def dec_factory():
    def dec(fn):
        def inner(*args, **kwargs):
            print('running decorator inner')
            return fn(*args, **kwargs)
        return inner
    return dec
```

[25]:
```
@dec_factory()
def my_func(a, b):
    return a + b
```

[26]:
```
my_func(10, 20)
```

```
running decorator inner
```

[26]:
```
30
```

You should note that in this approach, we are **calling** `dec_factory()`, [note the parentheses `()`], and **then** using the return value (a decorator) to decorate our function.

So, we could pass arguments as we do so without affecting the final outcome. In fact we can even access them from anywhere inside `dec_factory`, including any of the nested functions!

Let's try this:

[27]:
```python
def dec_factory(a, b):
    def dec(fn):
        def inner(*args, **kwargs):
            print('running decorator inner')
            print('free vars: ', a, b)  # a and b are free variables!
            return fn(*args, **kwargs)
        return inner
    return dec
```

[28]:
```python
@dec_factory(10, 20)
def my_func():
    print('python rocks')
```

[29]:
```python
my_func()
```

```
running decorator inner
free vars:  10 20
python rocks
```

And this is how we can create decorators with parameters. We do not directly create a decorator, instead we use an outer function that returns a decorator when called, and pass arguments to that outer function, which the decorator and its inner function can of course access as nonlocal (free) variables.

So now, let's go back to our original problem where we wanted our timing decorator to run a number of loops which could be specified as a parameter when decorating the function we want to time.

Here it is again:

[30]:
```python
def timed(fn, num_reps):
    from time import perf_counter

    def inner(*args, **kwargs):
        total_elapsed = 0
        for i in range(num_reps):
            start = perf_counter()
            result = fn(*args, **kwargs)
            end = perf_counter()
            total_elapsed += (perf_counter() - start)
        avg_elapsed = total_elapsed / num_reps
        print('Avg Run time: {0:.6f}s ({1} reps)'.format(avg_elapsed,
                                                        num_reps))
        return result
    return inner
```

So, all we need to do is create an outer function around our timed decorator, and pass the `num_reps` argument to that outer function instead:

[31]:
```
def timed_factory(num_reps=1):
    def timed(fn):
        from time import perf_counter

        def inner(*args, **kwargs):
            total_elapsed = 0
            for i in range(num_reps):
                start = perf_counter()
                result = fn(*args, **kwargs)
                end = perf_counter()
                total_elapsed += (perf_counter() - start)
            avg_elapsed = total_elapsed / num_reps
            print('Avg Run time: {0:.6f}s ({1} reps)'.format(avg_elapsed,
                                                            num_reps))

            return result
        return inner
    return timed
```

[32]:
```
@timed_factory(5)
def fib(n):
    return calc_fib_recurse(n)
```

[33]:
```
fib(30)
```

```
Avg Run time: 0.249934s (5 reps)
```

[33]: 832040

Just to put the finishing touch on this, we probably don't want to have our outer function named the way it is (`timed_factory`). Instead we probably just want to call it `timed`. So lets just do this final part:

[34]:
```
from functools import wraps

def timed(num_reps=1):
    def decorator(fn):
        from time import perf_counter

        @wraps(fn)
        def inner(*args, **kwargs):
            total_elapsed = 0
            for i in range(num_reps):
                start = perf_counter()
                result = fn(*args, **kwargs)
                end = perf_counter()
                total_elapsed += (perf_counter() - start)
            avg_elapsed = total_elapsed / num_reps
            print('Avg Run time: {0:.6f}s ({1} reps)'.format(avg_elapsed,
                                                              num_reps))

            return result
        return inner
    return decorator
```

[35]:
```
@timed(5)
def fib(n):
    return calc_fib_recurse(n)
```

[36]:
```
fib(30)
```

```
Avg Run time: 0.253744s (5 reps)
```

[36]:
```
832040
```

0.0.13 Decorator Application (Decorator Class)

If you recalls how we wrote a parameterized decorator, we had to write a decorator factory that took in the arguments for our decorator and then returned the decorator (which could reference the arguments as free variables).

Very simply:

[1]:
```
def my_dec(a, b):
    def dec(fn):
        def inner(*args, **kwargs):
            print('decorated function called: a={0}, b={1}'.format(a, b))
            return fn(*args, **kwargs)
        return inner
    return dec
```

[2]:
```
@my_dec(10, 20)
def my_func(s):
    print('hello {0}'.format(s))
```

[3]:
```
my_func('world')
```

```
decorated function called: a=10, b=20
hello world
```

So, our decorator factory was passed some arguments, and returned a callable which took one single parameter, the function being decorated, but also had access to the arguments passed to the factory.

Now, recall that we can make our class instances callable, simply by implementing the `__call__` method.

Here's a simple example:

[4]:
```
class MyClass:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self):
        print('MyClass instance called: a={0}, b={1}'.format(self.a, self.b))
```

[5]:
```
my_class = MyClass(10, 20)
```

[6]:
```
my_class()
```

```
MyClass instance called: a=10, b=20
```

So let's modify this just a bit, and have the `__call__` method be our decorator!

[7]:
```
class MyClass:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self, fn):
        def inner(*args, **kwargs):
            print('MyClass instance called: a={0}, b={1}'.format(self.a, self.
 ↪b))
            return fn(*args, **kwargs)
        return inner
```

So, we can decorate our functions this way:

[8]:
```
@MyClass(10, 20)
def my_func(s):
    print('Hello {0}!'.format(s))
```

Remember that `@MyClass(10, 20)` returned an object of type `MyClass`. But that object is itself callable, so we could do something like:

`my_func = MyClass(10, 20)(my_func)`

or, more simply

`@MyClass(10, 20) def my_func(s):     print(s)`

[9]:
```
my_func('Python')
```

```
MyClass instance called: a=10, b=20
Hello Python!
```

So as you can see, we can also use callable classes to decorate functions!

[ ]:
```python

```

0.0.14 Decorator Application: Decorating Classes

We have so far worked with decorating functions. This means we can decorate functions defined with a `def` statement (we can use the `@` syntax, or the long form). Since class methods are functions, they can be decorated too. Lambda expressions can also be decorated (using the long form).

But if you think about how our decorators work, they take a single parameter, a function, and return some other function - usually a closure that uses the original function that was passed as an argument.

We could use the same concept to accept, not a function, but a class instead. We could reference that class inside our decorator, modify it, and then return that modified class.

First we look at something called **monkey patching**. It boils down to modifying or extending our code at **run time**.

For example we can modify or add attributes to classes at run time. Modules too.

In Python, many of the classes we use can be modified at run time (built-ins like strings, lists, and so on, cannot).

But classes written in Python, such as the ones we write, and even library classes, as long as they are written in Python, not C, can. For example `Fraction` in the `fractions` module can be monkey patched.

Just because we can do something however, does not mean we should! Monkey patching can be extremely useful, but don't do it just because you can - as always there should be a real reason to do it, as we'll see in a bit.

Also, in general it is a bad idea to monkey patch the special methods `__???__` (such as `__len__`) as this will often not work due to how these methods are searched for by Python.

[1]:
```python
from fractions import Fraction
```

[2]:
```python
Fraction.speak = lambda self: 'This is a late parrot.'
```

[3]:
```python
f = Fraction(2, 3)
```

[4]:
```python
f
```

[4]:
```
Fraction(2, 3)
```

[5]:
```python
f.speak()
```

[5]:
```
'This is a late parrot.'
```

Yes, this is obviously nonsense, but you get the idea that you can add attributes to classes even if you do not have direct control over the class, or after your class has been defined.

If you want a more useful method, how about one that tells us if the Fraction is an integral number? (i.e. denominator is 1)

```
[6]: Fraction.is_integral = lambda self: self.denominator == 1
```

```
[7]: f1 = Fraction(1, 2)
     f2 = Fraction(10, 5)
```

```
[8]: f1.is_integral()
```

```
[8]: False
```

```
[9]: f2.is_integral()
```

```
[9]: True
```

Now, we can make this change to the class by calling a function to do it instead:

```
[10]: def dec_speak(cls):
          cls.speak = lambda self: 'This is a very late parrot.'
          return cls
```

```
[11]: Fraction = dec_speak(Fraction)
```

(Hopefully the above code reminds you of decorators.)

```
[12]: f = Fraction(10, 2)
```

```
[13]: f.speak()
```

```
[13]: 'This is a very late parrot.'
```

We can use that function to decorate our custom classes too, using the short **@** syntax too.

```
[14]: @dec_speak
      class Parrot:
          def __init__(self):
              self.state = 'late'
```

```
[15]: polly = Parrot()
```

```
[16]: polly.speak()
```

```
[16]: 'This is a very late parrot.'
```

Using this technique we could for example add a useful *reciprocal* attribute to the Fraction class, but of course since it would probably be a one time kind of thing (how many Fraction classes are there that you will want to add a reciprocal to after all), there's no need for decorators. Decorators are useful when they are able to be reused in more general ways.

[17]:
```
Fraction.recip = lambda self: Fraction(self.denominator, self.numerator)
```

[18]:
```
f = Fraction(2,3)
```

[19]:
```
f
```

[19]:
```
Fraction(2, 3)
```

[20]:
```
f.recip()
```

[20]:
```
Fraction(3, 2)
```

These example are quite trivial, and not very useful.

So why bring this up?

Because this same technique can be used for more interesting things.

As a first example, let's say you typically like to inspect various properties of an object for debugging purposes, maybe the memory address, it's current state (property values), and the time at which the debug info was generated.

[21]:
```
from datetime import datetime, timezone
```

[22]:
```
def debug_info(cls):
    def info(self):
        results = []
        results.append('time: {0}'.format(datetime.now(timezone.utc)))
        results.append('class: {0}'.format(self.__class__.__name__))
        results.append('id: {0}'.format(hex(id(self))))

        if vars(self):
            for k, v in vars(self).items():
                results.append('{0}: {1}'.format(k, v))

        # we have not covered lists, the extend method and generators,
        # but note that a more Pythonic way to do this would be:
        #if vars(self):
        #    results.extend('{0}: {1}'.format(k, v)
        #                   for k, v in vars(self).items())

        return results

    cls.debug = info
```

```python
    return cls
```

[23]:
```python
@debug_info
class Person:
    def __init__(self, name, birth_year):
        self.name = name
        self.birth_year = birth_year

    def say_hi():
        return 'Hello there!'
```

[24]:
```python
p1 = Person('John', 1939)
```

[25]:
```python
p1.debug()
```

[25]:
```
['time: 2018-02-09 04:44:02.893951+00:00',
 'class: Person',
 'id: 0x2dfe29a4630',
 'name: John',
 'birth_year: 1939']
```

And of course we can decorate other classes this way too, not just a single class:

[26]:
```python
@debug_info
class Automobile:
    def __init__(self, make, model, year, top_speed_mph):
        self.make = make
        self.model = model
        self.year = year
        self.top_speed_mph = top_speed_mph
        self.current_speed = 0

    @property
    def speed(self):
        return self.current_speed

    @speed.setter
    def speed(self, new_speed):
        self.current_speed = new_speed
```

[27]:
```python
s = Automobile('Ford', 'Model T', 1908, 45)
```

[28]:
```python
s.debug()
```

[28]:
```
['time: 2018-02-09 04:44:03.562898+00:00',
 'class: Automobile',
 'id: 0x2dfe29b3a58',
 'make: Ford',
 'model: Model T',
 'year: 1908',
 'top_speed_mph: 45',
 'current_speed: 0']
```

[29]:
```
s.speed = 20
```

[30]:
```
s.debug()
```

[30]:
```
['time: 2018-02-09 04:44:03.898085+00:00',
 'class: Automobile',
 'id: 0x2dfe29b3a58',
 'make: Ford',
 'model: Model T',
 'year: 1908',
 'top_speed_mph: 45',
 'current_speed: 20']
```

Let's look at another example where decorating an entire class could be useful.

[31]:
```
from math import sqrt
```

[32]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __abs__(self):
        return sqrt(self.x**2 + self.y**2)

    def __repr__(self):
        return 'Point({0},{1})'.format(self.x, self.y)
```

[33]:
```
p1, p2, p3 = Point(2, 3), Point(2, 3), Point(0,0)
```

[34]:
```
abs(p1)
```

[34]:
```
3.605551275463989
```

[35]:
```
p1, p2
```

[35]:
```
(Point(2,3), Point(2,3))
```

[36]:
```
p1 == p2
```

```
[36]: False
```

Hmm, we probably would have expected `p1` to be equal to `p2` since it has the same coordinates. But by default Python will compare memory addresses, since our class does not implement the `__eq__` method used for `==` comparisons.

```
[37]: p2, p3
```

```
[37]: (Point(2,3), Point(0,0))
```

```
[38]: p2 > p3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-b46639986960> in <module>()
----> 1 p2 > p3

TypeError: '>' not supported between instances of 'Point' and 'Point'
```

So, that class does not support the comparison operators such as `<`, `<=`, etc.

Even `==` does not work as expected - it will use the memory address instead of using a comparison of the `x` and `y` coordinates as we might probably expect.

For the `<` operator, we need our class to implement the `__lt__` method, and for `==` we need the `__eq__` method.

Other comparison operators are supported by implementing a variety of functions such as `__le__` (`<=`), `__gt__` (`>`), `__ge__` (`>=`).

We are going to add the `__lt__` and `__eq__` methods to our Point class.

We will consider a Point object to be smaller than another one if it is closer to the origin (i.e. smaller magnitude).

```
[39]: del Point

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __abs__(self):
        return sqrt(self.x**2 + self.y**2)

    def __eq__(self, other):
        if isinstance(other, Point):
            return self.x == other.x and self.y == other.y
        else:
            return NotImplemented

    def __lt__(self, other):
        if isinstance(other, Point):
            return abs(self) < abs(other)
        else:
            return NotImplemented

    def __repr__(self):
        return '{0}({1},{2})'.format(self.__class__.__name__, self.x, self.y)
```

[40]:
```
p1, p2, p3 = Point(2, 3), Point(2, 3), Point(0,0)
```

[41]:
```
p1, p2, p1==p2
```

[41]:
```
(Point(2,3), Point(2,3), True)
```

[42]:
```
p2, p3, p2==p3
```

[42]:
```
(Point(2,3), Point(0,0), False)
```

As we can see, `==` now works as expected

[43]:
```
p4 = Point(1, 2)
```

[44]:
```
abs(p1), abs(p4), p1 < p4
```

[44]:
```
(3.605551275463989, 2.23606797749979, False)
```

Great, so now we have `<` and `==` implemented. What about the rest of the operators: `<=`, `>`, `>=`?

[45]:
```
p1 > p4
```

[45]:
```
True
```

Ooh, since we have implemented `<` and `==`, does this mean Python magically implemented a `>` operator (i.e. not $<$ and not $==$)?

Not exactly! What happened is that since `p1` and `p4` are both points, running the comparison `p1 > p4` is really the same as evaluating `p4 < p1` - and Python did do that automatically for us.

But it has not implemented any of the others, such as `>=` and `<=`:

[46]:
```
p1 <= p4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-80f9ea228203> in <module>()
----> 1 p1 <= p4
```

```
TypeError: '<=' not supported between instances of 'Point' and 'Point'
```

Now, although we could proceed in a similar way and define `>=`, `<=` and `>` using the same technique, observe that if `<` and `==` is defined then:

- `a <= b` iff `a < b or a == b`
- `a > b` iff `not(a<b) and a != b`
- `a >= b` iff `not(a<b)`

So, to be quite generic we could create a decorator that will implement these last three operators as long as `==` and `<` are defined. We could then decorate **any** class that implements just those two operators.

```
[47]: def complete_ordering(cls):
          if '__eq__' in dir(cls) and '__lt__' in dir(cls):
              cls.__le__ = lambda self, other: self < other or self == other
              cls.__gt__ = lambda self, other: not(self < other) and not (self ==␣
       ↪other)
              cls.__ge__ = lambda self, other: not (self < other)
          return cls
```

In reality, the code above is **NOT** a good implementation at all. We are not checking that the types are compatible and returning a `NotImplemented` result if appropriate. I am also using inline operators (`<` and `==`) instead of the dunder functions (`__lt__` and `__eq__`). I just kept it simple because we'll use a better alternative in a bit.

For example, a better way to implement `__ge__` would be as follows:

```
[48]: def ge_from_lt(self, other):
          # self >= other iff not(other < self)
          result = self.__lt__(other)
          if result is NotImplemented:
              return NotImplemented
          else:
              return not result
```

You may be wondering why I used `__lt__` instead of just using the `<` operator. This is because I want to actually look at the result of the operation without raising an exception if the operation is not implemented. The way I have the total ordering decorator implemented could cause an infinite loop because when I evaluate `self < other`, if an exception is raised, Python will reflect the evaluation to `other > self`, and if that raises an error as well, Python will try to reflect that operation too, and we get into an infinite loop (which eventually terminates in a stack overflow). This was actually a bug in Python's standard library implementation of a `complete_ordering` decorator (called `total_ordering`) that was resolved in 3.4.

```
[49]: class Point:
          def __init__(self, x, y):
              self.x = x
```

```
        self.y = y

    def __abs__(self):
        return sqrt(self.x**2 + self.y**2)

    def __eq__(self, other):
        if isinstance(other, Point):
            return self.x == other.x and self.y == other.y
        else:
            return NotImplemented

    def __lt__(self, other):
        if isinstance(other, Point):
            return abs(self) < abs(other)
        else:
            return NotImplemented

    def __repr__(self):
        return '{0}({1},{2})'.format(self.__class__, self.x, self.y)
```

[50]:
```
Point = complete_ordering(Point)
```

[51]:
```
p1, p2, p3 = Point(1, 1), Point(3, 4), Point(3, 4)
```

[52]:
```
abs(p1), abs(p2), abs(p3)
```

[52]:
```
(1.4142135623730951, 5.0, 5.0)
```

[53]:
```
p1 < p2, p1 <= p2, p1 > p2, p1 >= p2, p2 > p2, p2 >= p3
```

[53]:
```
(True, True, False, False, False, True)
```

Now the `complete_ordering` decorator can also be directly applied to any class that defines `__eq__` and `__lt__`.

[54]:
```
@complete_ordering
class Grade:
    def __init__(self, score, max_score):
        self.score = score
        self.max_score = max_score
        self.score_percent = round(score / max_score * 100)

    def __repr__(self):
        return 'Grade({0}, {1})'.format(self.score, self.max_score)

    def __eq__(self, other):
        if isinstance(other, Grade):
            return self.score_percent == other.score_percent
        else:
            return NotImplemented

    def __lt__(self, other):
        if isinstance(other, Grade):
            return self.score_percent < other.score_percent
        else:
            return NotImplemented
```

[55]:
```
g1 = Grade(10, 100)
g2 = Grade(20, 30)
g3 = Grade(5, 50)
```

[56]:
```
g1 <= g2, g1 == g3, g2 > g3
```

[56]:
```
(True, True, True)
```

Often, given the `==` operator and just **one** of the other comparison operators (`<`, `<=`, `>`, `>=`), then all the rest can be derived.

Our decorator insisted on `==` and `<`. but we could make it better by insisting on `==` and any one of the other operators. This will of course make our decorator more complicated, and in fact, Python has this precise functionality built in to the, you guessed it, `functools` module!

It is a decorator called `total_ordering`.

Let's see it in action:

[57]:
```
from functools import total_ordering
```

[58]:
```
@total_ordering
class Grade:
    def __init__(self, score, max_score):
        self.score = score
        self.max_score = max_score
        self.score_percent = round(score / max_score * 100)

    def __repr__(self):
        return 'Grade({0}, {1})'.format(self.score, self.max_score)

    def __eq__(self, other):
        if isinstance(other, Grade):
            return self.score_percent == other.score_percent
        else:
            return NotImplemented

    def __lt__(self, other):
        if isinstance(other, Grade):
            return self.score_percent < other.score_percent
        else:
            return NotImplemented
```

[59]:
```
g1, g2 = Grade(80, 100), Grade(60, 100)
```

[60]:
```
g1 >= g2, g1 > g2
```

[60]:
```
(True, True)
```

Or we could also do it this way:

[61]:
```
@total_ordering
class Grade:
    def __init__(self, score, max_score):
        self.score = score
        self.max_score = max_score
        self.score_percent = round(score / max_score * 100)

    def __repr__(self):
        return 'Grade({0}, {1})'.format(self.score, self.max_score)

    def __eq__(self, other):
        if isinstance(other, Grade):
            return self.score_percent == other.score_percent
        else:
            return NotImplemented

    def __gt__(self, other):
        if isinstance(other, Grade):
            return self.score_percent > other.score_percent
        else:
            return NotImplemented
```

[62]:
```
g1, g2 = Grade(80, 100), Grade(60, 100)
```

[63]:
```
g1 >= g2, g1 > g2, g1 <= g2, g1 < g2
```

[63]:
```
(True, True, False, False)
```

0.0.15 Decorator Application: Single Dispatch Generic Functions

Consider an application where we want to provide similar functionality but that varies slightly depending on the argument types passed in.

In this set of examples we consider this problem where functionality differs based on a single argument's type (hence single dispatch) instead of the type of multiple arguments (which would be multi dispatch)

If you have a background in some other OO languages such as Java or C#, you'll know that we can easily do something like this by basically **overloading** functions: using a different data type for the function parameter, hence changing the function signature. Then although the name of the function is the same, calling `do_something(100)` and `do_something('java')` would call a different function, the first one would call the `do_something(int)` function, and the second would call the `do_something(String)` function.

Of course, Python is not statically typed, so even if Python had function overloading built-in, we would not be able to make such a distinction in our function signatures since there is nothing that says that a parameter must be of a specific type, so in a best case scenario we would have to "distinguish" functions with the same name only by the number of parameters they take. And then we'd have to somehow deal with variable numbers of positional and keyword arguments too… Uuugh! In any event, single dispatch could never work.

Instead we have to come up with a different solution.

Let's say we want to display various data types in html format, with different presentations for integers (we want both base 10 and hex values), floats (we always want it rounded to 2 decimal points), strings (we want the string html-escaped, and all newline characters replaced by `<br/>`), lists and tuples should be implemented using bulleted lists, and the same with dictionaries except we want the name/value pair to be displayed in the bulleted list.

For starters, let's just implement individual functions to do each of those things.

I am going to keep the functions very simple, but in practice you should handle situations like None objects, empty lists and dictionaries, possibly the wrong type being passed to the function, etc.

[1]:
```python
from html import escape

def html_escape(arg):
    return escape(str(arg))

def html_int(a):
    return '{0}(<i>{1}</i)'.format(a, str(hex(a)))

def html_real(a):
    return '{0:.2f}'.format(round(a, 2))

def html_str(s):
    return html_escape(s).replace('\n', '<br/>\n')

def html_list(l):
    items = ('<li>{0}</li>'.format(html_escape(item))
             for item in l)
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'

def html_dict(d):
    items = ('<li>{0}={1}</li>'.format(html_escape(k), html_escape(v))
             for k, v in d.items())
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'
```

[2]:
```python
print(html_str("""this is
a multi line string
with special characters: 10 < 100"""))
```

```
this is <br/>
a multi line string<br/>
with special characters: 10 &lt; 100
```

[3]:
```python
print(html_int(255))
```

```
255(<i>0xff</i)
```

[4]:
```python
print(html_escape(3+10j))
```

```
(3+10j)
```

Ideally we would want to just have to call a single function, maybe `htmlize` that would figure out which particular flavor of the `html_xxx` function to call depending on the argument type.

We could try it as follows:

[5]:
```python
from decimal import Decimal

def htmlize(arg):
    if isinstance(arg, int):
        return html_int(arg)
    elif isinstance(arg, float) or isinstance(arg, Decimal):
        return html_real(arg)
    elif isinstance(arg, str):
        return html_str(arg)
    elif isinstance(arg, list) or isinstance(arg, tuple):
        return html_list(arg)
    elif isinstance(arg, dict):
        return html_dict(arg)
    else:
        # default behavior - just html escape string representation
        return html_escape(str(arg))
```

Now we can essentially use the same function call to handle different types - the `htmlize` function is a dispatcher - it dispatches the request to a different function based on the argument type. (There's a much better way to do some of this, but we'll have to wait until we cover abstract base classes to do so).

[6]:
```python
print(htmlize([1, 2, 3]))
```

```
<ul>
<li>1</li>
<li>2</li>
```

```
<li>3</li>
</ul>
```

[7]:
```
print(htmlize(dict(key1=1, key2=2)))
```

```
<ul>
<li>key1=1</li>
<li>key2=2</li>
</ul>
```

[8]:
```
print(htmlize(255))
```

```
255(<i>0xff</i)
```

But there are a number of shortcomings here:

[9]:
```
print(htmlize(["""first element is
a multi-line string""", (1, 2, 3)]))
```

```
<ul>
<li>first element is
a multi-line string</li>
<li>(1, 2, 3)</li>
</ul>
```

As you can see, the multi-line string did not get the newline characters replaced, the tuple was not rendered as an html list, and the integers do not have their hex representation.

So we just need to redefine the `html_list` and `html_dict` functions to use the `htmlize` function:

[10]:
```
def html_list(l):
    items = ['<li>{0}</li>'.format(htmlize(item)) for item in l]
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'
```

[11]:
```
def html_dict(d):
    items = ['<li>{0}={1}</li>'.format(html_escape(k), htmlize(v)) for k, v in␣
  ↪d.items()]
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'
```

[12]:
```
print(htmlize(["""first element is
a multi-line string""", (1, 2, 3)]))
```

```
<ul>
<li>first element is <br/>
a multi-line string</li>
<li><ul>
<li>1(<i>0x1</i)</li>
<li>2(<i>0x2</i)</li>
<li>3(<i>0x3</i)</li>
```

```
</ul></li>
</ul>
```

Much better, but hopefully you spotted something that might seem problematic!

Do we not have a circular reference?

In order to define `html_list` and `html_dict` we needed to call `htmlize`, but in order to define `htmlize` we needed to call `html_list` and `html_dict`.

Remember that in Python we can reference a function **inside** the body of another function **before** the function has been defined, as long as by the time we **call** the first function, the second one has been defined. SO this is actually OK.

If you don't believe me and want to make sure of this yourself, go ahead and reset your Kernel (click on the Kernel | Restart menu option), and run the following code without running anything prior to this.

The `htmlize` function body makes calls to other functions such as `html_escape`, `html_int`, etc that have not actually been defined yet

[1]:
```
from html import escape
from decimal import Decimal

def htmlize(arg):
    if isinstance(arg, int):
        return html_int(arg)
    elif isinstance(arg, float) or isinstance(arg, Decimal):
        return html_real(arg)
    elif isinstance(arg, str):
        return html_str(arg)
    elif isinstance(arg, list) or isinstance(arg, tuple) or isinstance(arg,␣
 ↪set):
        return html_list(arg)
    elif isinstance(arg, dict):
        return html_dict(arg)
    else:
        # default behavior - just html escape string representation
        return html_escape(str(arg))
```

Now we define all the functions that `htmlize` uses before we actually call `htmlize` and all is good:

[14]:
```
def html_escape(arg):
    return escape(str(arg))

def html_int(a):
    return '{0}(<i>{1}</i)'.format(a, str(hex(a)))

def html_real(a):
    return '{0:.2f}'.format(round(a, 2))
```

```
def html_str(s):
    return html_escape(s).replace('\n', '<br/>\n')

def html_list(l):
    items = ['<li>{0}</li>'.format(htmlize(item)) for item in l]
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'

def html_dict(d):
    items = ['<li>{0}={1}</li>'.format(html_escape(k), htmlize(v)) for k, v in␣
 ↪d.items()]
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'
```

[15]:
```
print(htmlize(["""first element is
a multi-line string""", (1, 2, 3)]))
```

```
<ul>
<li>first element is <br/>
a multi-line string</li>
<li><ul>
<li>1(<i>0x1</i)</li>
<li>2(<i>0x2</i)</li>
<li>3(<i>0x3</i)</li>
</ul></li>
</ul>
```

As you can see this works just fine.

But we still have something undesirable. You'll notice that the dispatch function `htmlize` needs to have this big `if...elif...else` statement that will just keep growing as we need to handle more and more types (including potentially custom types).

This will just get unwieldy, and not very flexible (every time someone creates a new type that has to have a special html representation they will need to go into the `htmlize` function and modify it.

So instead, we are going to try a more flexible approach using decorators.

The way we are going to approach this is to create a dispatcher function, and then separately "register" each type-specific function with the dispatcher.

First, we are going to create a decorator that will do something that may seem kind of silly - it is going to take the decorated function and store it in a dictionary, using a key consisting of the **type** `object`.

Then when the returned closure is called, the closure will call the function stored in that dictionary.

[16]:
```
def singledispatch(fn):
    registry = dict()
    registry[object] = fn

    def inner(arg):
        return registry[object](arg)
```

```python
    return inner
```

[17]:
```python
@singledispatch
def htmlizer(arg):
    return escape(str(arg))
```

[18]:
```python
htmlizer('a < 10')
```

[18]:
```
'a &lt; 10'
```

Next, we are going to add some functions to that `registry` dictionary, and modify our inner function to choose the correct function from the registry, or pick a default based on the type of the argument:

[19]:
```python
def singledispatch(fn):
    registry = dict()

    registry[object] = fn
    registry[int] = lambda arg: '{0}(<i>{1}</i)'.format(arg, str(hex(arg)))
    registry[float] = lambda arg: '{0:.2f}'.format(round(arg, 2))

    def inner(arg):
        fn = registry.get(type(arg), registry[object])
        return fn(arg)
    return inner
```

[20]:
```python
@singledispatch
def htmlize(a):
    return escape(str(a))
```

[21]:
```python
htmlize(10)
```

[21]:
```
'10(<i>0xa</i)'
```

[22]:
```python
htmlize(3.1415)
```

[22]:
```
'3.14'
```

Now, we want a way to add the specialized functions to the `registry` dictionary from **outside** the `singledispatch` function - to do so we will create a parametrized decorator that will (1) take the type as a parameter, and (2) return a closure that will decorate the function associated with the type:

[23]:
```python
def singledispatch(fn):
    registry = dict()

    registry[object] = fn
    def register(type_):
        def inner(fn):
            registry[type_] = fn
        return inner


    def decorator(arg):
        fn = registry.get(type(arg), registry[object])
        return fn(arg)

    return decorator
```

But of course this is not good enough - how do we get a hold of the `register` function from outside `singledispatch`? Remember, `singledispatch` is a decorator that returns the `decorated` closure, not the `register` closure.

We can do this by adding the `register` function as an **attribute** of the `decorated` function before we return it.

While we're at it we're also going to:

- add the `registry` dictionary as an attribute as so we can look into it to see what it contains.
- add another function that given a type will return the function associated with that type (or the default function if the type is not found in the dictionary)

[24]:
```
def singledispatch(fn):
    registry = dict()

    registry[object] = fn

    def register(type_):
        def inner(fn):
            registry[type_] = fn
            return fn  # we do this so we can stack register decorators!
        return inner

    def decorator(arg):
        fn = registry.get(type(arg), registry[object])
        return fn(arg)

    def dispatch(type_):
        return registry.get(type_, registry[object])

    decorator.register = register
    decorator.registry = registry.keys()
    decorator.dispatch = dispatch
    return decorator
```

[25]:
```python
@singledispatch
def htmlize(arg):
    return escape(str(arg))
```

And we can see that `htmlize` (that returned `inner`) function has an attribute called `register`:

[26]:
```python
htmlize.register
```

[26]:
```
<function __main__.singledispatch.<locals>.register>
```

as well as that `registry` attribute that we put in just we could see what keys are in the `registry` dictionary:

[27]:
```python
htmlize.registry
```

[27]:
```
dict_keys([<class 'object'>])
```

We can also ask it what function it is going to use for any specific type (currently we only have one registered, the default, for the most general `object` type):

[28]:
```python
htmlize.dispatch(str)
```

[28]:
```
object
```

And you'll note that the extended scope of `register` and `dispatch` is the same as the extended scope of `htmlize`.

So now we can register some functions (it will store the function with associated data type in the `registry` dictionary):

[29]:
```python
@htmlize.register(int)
def html_int(a):
    return '{0}(<i>{1}</i)'.format(a, str(hex(a)))
```

We can peek into the registered types:

[30]:
```python
htmlize.registry
```

[30]:
```
dict_keys([<class 'object'>, <class 'int'>])
```

and we can ask the decorated `htmlize` function what function it is going to use for the `int` type:

[31]:
```python
htmlize.dispatch(int)
```

[31]:
```
<function __main__.html_int>
```

and we can actually call it as well:

[32]:
```python
htmlize(100)
```

[32]:
```
'100(<i>0x64</i)'
```

The huge advantage now is that we can keep registering new handlers from anywhere in our module, or even from outside our module!

[33]:
```
@htmlize.register(float)
def html_real(a):
    return '{0:.2f}'.format(round(a, 2))

@htmlize.register(str)
def html_str(s):
    return escape(s).replace('\n', '<br/>\n')

@htmlize.register(tuple)
@htmlize.register(list)
def html_list(l):
    items = ['<li>{0}</li>'.format(htmlize(item)) for item in l]
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'

@htmlize.register(dict)
def html_dict(d):
    items = ['<li>{0}={1}</li>'.format(htmlize(k), htmlize(v)) for k, v in d.
 ↪items()]
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'
```

[34]:
```
htmlize.registry
```

[34]:
```
dict_keys([<class 'object'>, <class 'int'>, <class 'float'>, <class 'str'>,
<class 'list'>, <class 'tuple'>, <class 'dict'>])
```

[35]:
```
print(htmlize([1, 2, 3]))
```

```
<ul>
<li>1(<i>0x1</i)</li>
<li>2(<i>0x2</i)</li>
<li>3(<i>0x3</i)</li>
</ul>
```

[36]:
```
print(htmlize((1, 2, 3)))
```

```
<ul>
<li>1(<i>0x1</i)</li>
<li>2(<i>0x2</i)</li>
<li>3(<i>0x3</i)</li>
</ul>
```

[37]:
```
print(htmlize("""this
is a multi line string with
a < 10"""))
```

```
this<br/>
is a multi line string with<br/>
a &lt; 10
```

Our single dispatch decorator works quite well - but it has some limitations. For example it cannot handle functions that take in more than one argument (in which case dispatching would be based on the type of the **first** argument), and we also are not allowing for types based on parent classes - for example, integers and booleans are both integral numbers - i.e. they both inherit from the Integral base class. Similarly lists and tuples are both more generic Sequence types. We'll see this in more detail when we get to the topic of abstract base classes (ABC's).

```
[38]: from numbers import Integral
```

```
[39]: isinstance(100, Integral)
```

```
[39]: True
```

```
[40]: isinstance(True, Integral)
```

```
[40]: True
```

```
[41]: isinstance(100.5, Integral)
```

```
[41]: False
```

```
[42]: type(100) is Integral
```

```
[42]: False
```

```
[43]: type(True) is Integral
```

```
[43]: False
```

```
[44]: (100).__class__
```

```
[44]: int
```

```
[45]: (True).__class__
```

```
[45]: bool
```

The way we have implement our decorator, if we register an Integral generic function, it won't pick up either integers or Booleans.

We can certainly fix this shortcoming ourselves, but of course…

We can can use Python's built-in single dispatch support, in …

you guessed it!

the `functools` module.

[46]:
```
from functools import singledispatch
from numbers import Integral
from collections.abc import Sequence
```

[47]:
```
@singledispatch
def htmlize(a):
    return escape(str(a))
```

The `singledispatch` returned closure has a few attributes we can use: 1. A `register` decorator (just like ours did) 2. A `registry` property that is the registry dictionary 3. A `dispatch` function that can be used to determine which registry key (registered type) it will use for the specified type.

[48]:
```
@htmlize.register(Integral)
def htmlize_int(a):
    return '{0}(<i>{1}</i)'.format(a, str(hex(a)))
```

[49]:
```
htmlize.dispatch(int)
```

[49]:
```
<function __main__.htmlize_int>
```

[50]:
```
htmlize.dispatch(bool)
```

[50]:
```
<function __main__.htmlize_int>
```

[51]:
```
htmlize(100)
```

[51]:
```
'100(<i>0x64</i)'
```

[52]:
```
htmlize(True)
```

[52]:
```
'True(<i>0x1</i)'
```

[53]:
```
@htmlize.register(Sequence)
def html_sequence(l):
    items = ['<li>{0}</li>'.format(htmlize(item)) for item in l]
    return '<ul>\n' + '\n'.join(items) + '\n</ul>'
```

[54]:
```
htmlize.dispatch(list)
```

[54]:
```
<function __main__.html_sequence>
```

[55]:
```
htmlize.dispatch(tuple)
```

[55]:
```
<function __main__.html_sequence>
```

```
[56]: htmlize.dispatch(str)
```

```
[56]: <function __main__.html_sequence>
```

You'll note that a string is also a sequence type, hence our dispatcher will call the `html_sequence` function on a string.

In fact, at this point things would not even run properly.

If we were to call

`htmlize('abc')`

we'd get an infinite recursion!

The call to `htmlize` the string `abc` would treat it as a sequence, which would call `htmlize` character by character. But each character is itself just a string of length 1, so it will `htmlize` for that single character, which would treat it as a sequence, which would call `htmlize` for that single character again, and so on, in an infinite loop.

```
[57]: htmlize('abc')
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-57-d6479a8af936> in <module>()
----> 1 htmlize('abc')

D:\Users\fbapt\Anaconda3\envs\deepdive\lib\functools.py in wrapper(*args, **kw)
    801
    802     def wrapper(*args, **kw):
--> 803         return dispatch(args[0].__class__)(*args, **kw)
    804
    805     registry[object] = func

<ipython-input-53-50c13b0d81b3> in html_sequence(l)
      1 @htmlize.register(Sequence)
      2 def html_sequence(l):
----> 3     items = ['<li>{0}</li>'.format(htmlize(item)) for item in l]
      4     return '<ul>\n' + '\n'.join(items) + '\n</ul>'

<ipython-input-53-50c13b0d81b3> in <listcomp>(.0)
      1 @htmlize.register(Sequence)
      2 def html_sequence(l):
----> 3     items = ['<li>{0}</li>'.format(htmlize(item)) for item in l]
      4     return '<ul>\n' + '\n'.join(items) + '\n</ul>'

… last 3 frames repeated, from the frame below …

D:\Users\fbapt\Anaconda3\envs\deepdive\lib\functools.py in wrapper(*args, **kw)
    801
    802     def wrapper(*args, **kw):
--> 803         return dispatch(args[0].__class__)(*args, **kw)
    804 
    805     registry[object] = func

RecursionError: maximum recursion depth exceeded
```

Instead, we are going to register a string handler specifically - that way we will avoid that problem entirely:

[58]:
```
@htmlize.register(str)
def html_str(s):
    return escape(s).replace('\n', '<br/>\n')
```

[59]:
```
htmlize.dispatch(str)
```

[59]:
```
<function __main__.html_str>
```

So, even though a string is both an `str` instance and in general a sequence type, the "closest" type will be picked by the dispatcher (again something our own implementation did not do).

This means, we have something for generic sequences, but something specific for more specialized strings.

[60]:
```
htmlize('abc')
```

[60]:
```
'abc'
```

We can do the same thing with sequences - right now `html_sequence` will be used for both lists and tuples.

But suppose we want slightly different handling of tuples:

[61]:
```
@htmlize.register(tuple)
def html_tuple(t):
    items = [escape(str(item)) for item in t]
    return '({0})'.format(', '.join(items))
```

[62]:
```
htmlize.dispatch(list)
```

[62]:
```
<function __main__.html_sequence>
```

[63]:
```
htmlize.dispatch(tuple)
```

[63]:
```
<function __main__.html_tuple>
```

[64]:
```
print(htmlize(['a', 100, 3.14]))
```

```
<ul>
<li>a</li>
<li>100(<i>0x64</i)</li>
<li>3.14</li>
</ul>
```

```
[65]: print(htmlize(('a', 100, 3.14)))
```

```
(a, 100, 3.14)
```

One thing of note is that we started our decoration with a `@singledispatch` decorator - you'll notice that no specific type was indicated here - and in fact this means the dispatcher will use the generic `object` type.

This means that any object type not specifically handled by our dispatcher will fall back on that `object` key - hence you can think of it as the default for the dispatcher.

```
[66]: type(None)
```

```
[66]: NoneType
```

```
[67]: htmlize.dispatch(type(None))
```

```
[67]: <function __main__.htmlize>
```

```
[68]: type(1+1j)
```

```
[68]: complex
```

```
[69]: htmlize.dispatch(complex)
```

```
[69]: <function __main__.htmlize>
```

```
[70]: type(3)
```

```
[70]: int
```

```
[71]: htmlize.dispatch(int)
```

```
[71]: <function __main__.htmlize_int>
```

Lastly, because the name of the individual specialized functions does not really matter to us (the dispatcher will pick the appropriate function), it is quite common for an underscore character (_) to be used for the function name - the memory address of each specialized function will be stored in the `registry` dictionary, and the function name does not matter - in fact we can even add lambdas to the registry.

```
[72]: @singledispatch
      def htmlize(a):
```

```
    return escape(str(a))
```

[73]:
```
@htmlize.register(int)
def _(a):
    return '{0}({1})'.format(a, str(hex(a)))
```

[74]:
```
@htmlize.register(str)
def _(s):
    return escape(s).replace('\n', '<br/>\n')
```

[75]:
```
htmlize.register(float)(lambda f: '{0:.2f}'.format(f))
```

[75]:
```
<function __main__.<lambda>>
```

[76]:
```
htmlize.registry
```

[76]:
```
mappingproxy({object: <function __main__.htmlize>,
              int: <function __main__._>,
              str: <function __main__._>,
              float: <function __main__.<lambda>>})
```

But note that the `__main__._` function for `int` and `str` are not the same functions (even tough they have the same name):

[77]:
```
id(htmlize.registry[str])
```

[77]:
```
3104966916432
```

[78]:
```
id(htmlize.registry[int])
```

[78]:
```
3104967451784
```

And everything works as expected:

[79]:
```
htmlize(100)
```

[79]:
```
'100(0x64)'
```

[80]:
```
htmlize(3.1415)
```

[80]:
```
'3.14'
```

[81]:
```
print(htmlize("""this
is a multi-line string
a < 10"""))
```

```
this<br/>
is a multi-line string<br/>
a &lt; 10
```

If this same name but different function thing has you confused, look at it this way:

```
[82]: def my_func():
          print('my_func initial')
```

```
[83]: id(my_func)
```

```
[83]: 3104966916296
```

```
[84]: f = my_func
```

```
[85]: id(f)
```

```
[85]: 3104966916296
```

So, `f` and `my_func` point to the same function in memory.

Let's go ahead and "redefine" the function `my_func`:

```
[86]: def my_func():
          print('second my_func')
```

In fact, we did not "redefine" the previous `my_func`, it still exists in memory (and `f` still points to it). Instead we have re-assigned the function that `my_func` points to:

```
[87]: id(my_func)
```

```
[87]: 3104966914800
```

But the original `my_func` is still around, and 'f' still has a reference to it:

```
[88]: id(f)
```

```
[88]: 3104966916296
```

So, we can call each one:

```
[89]: f()
```

```
my_func initial
```

```
[90]: my_func()
```

```
second my_func
```

But the function `__name__` have the same value:

```
[91]: f.__name__
```

```
[91]: 'my_func'
```

```
[92]: my_func.__name__
```

```
[92]: 'my_func'
```

Just always keep in mind that labels point to something in memory, it is not the object itself. So in this case we have two distinct objects (functions) which happen to have the same name, but are two very different objects - `f` points to the first one we created, and `my_func` points to the second.