Decorators

دکوراتورها می‌توانند به عنوان توابعی در نظر گرفته شوند که قابلیت تغییر عملکرد تابع دیگری را دارند. آنها به کوتاه‌تر و بیشتر "پایتونی" شدن کد شما کمک می‌کنند.

برای توضیح مناسب دکوراتورها، ما از توابع شروع کرده و به آرامی آنها را گسترش خواهیم داد. اطمینان حاصل کنید که تمام سلول‌های این نوت‌بوک را اجرا کنید تا این درس در رایانه شما به همان شکل نشان داده شود.

توابع

قبل از اینکه بتوانید دکوراتورها را درک کنید، باید ابتدا نحوه عملکرد توابع را درک کنید. برای مقاصد ما، یک تابع بر اساس آرگومان‌های داده شده، یک مقدار را برمی‌گرداند. در ادامه، یک مثال بسیار ساده وجود دارد:

In [1]:

```
def add_one(number):
    return number + 1

add_one(5)
```

Out[1]:

```
6
```

در کل، توابع در پایتون ممکن است تأثیرات جانبی داشته باشند به جای اینکه فقط یک ورودی را به خروجی تبدیل کنند. تابع `print()` مثالی ابتدایی در این مورد است: آن مقداری را برگردانده و در عین حال تأثیر جانبی خروجی دادن چیزی به کنسول دارد. با این حال، برای درک دکوراتورها، کافی است به توابع به عنوان چیزی که آرگومان‌های داده شده را به یک مقدار تبدیل می‌کنند، فکر کنید.

First-Class Objects

در پایتون، توابع یک قسمت از اولین کلاس ها هستند. این به این معنی است که توابع می‌توانند به عنوان آرگومان به تابع دیگری منتقل شوند و مانند هر شیء دیگری (رشته، عدد صحیح، عدد اعشاری، لیست و غیره) استفاده شوند. به مثال زیر توجه کنید:

In [2]:

```
def say_hello(name):
    return f"Hello {name}"

def be_awesome(name):
    return f"Yo {name}, together we are the awesomest!"

def greet_bob(greeter_func):
    return greeter_func("Bob")
```

In [3]:

```
type(say_hello)
```

Out[3]:

```
function
```

In [5]:

```
ss = say_hello
```

In [7]:

```
say_hello('John')
```

Out[7]:

```
'Hello John'
```

In [8]:

```
ss('John')
```

Out[8]:

```
'Hello John'
```

In [ ]:

```
add = lambda x, y : x + y
```

در اینجا، توابع say_hello() و be_awesome() توابع معمولی هستند که یک نام به عنوان ورودی دریافت می‌کنند. اما تابع greet_bob() توقع دارد یک تابع را به عنوان آرگومان خود دریافت کند. مثلاً، می‌توانیم آن را به عنوان آرگومان تابع say_hello() یا be_awesome() منتقل کنیم:

In [9]:

```
greet_bob(say_hello)
```

Out[9]:

```
'Hello Bob'
```

In [10]:

```
1 greet_bob(be_awesome)
```

Out[10]:

```
'Yo Bob, together we are the awesomest!'
```

توجه کنید که greet_bob(say_hello) به دو تابع ارجاع دارد، اما به روش‌های متفاوتی: تابع greet_bob() و تابع say_hello بدون پرانتز نامگذاری شده است. این بدان معنی است که فقط یک ارجاع به تابع say_hello. منتقل می‌شود و تابع اجرا نمی‌شود. از طرف دیگر، تابع greet_bob() با پرانتز نوشته شده است، بنابراین به طور معمول فراخوانی خواهد شد.

توابع داخلی

ممکن است توابع را در داخل توابع دیگر تعریف کنیم. اینگونه توابع را توابع داخلی می‌نامیم. در ادامه مثالی از یک تابع با دو تابع داخلی آورده شده است.

In [11]:

```
1  def parent():
2      print("Printing from the parent() function")
3
4      def first_child():
5          print("Printing from the first_child() function")
6
7      def second_child():
8          print("Printing from the second_child() function")
9
10     second_child()
11     first_child()
```

چه اتفاقی می‌افتد وقتی تابع parent() را فراخوانی می‌کنید؟ برای یک دقیقه به این موضوع فکر کنید. خروجی به شرح زیر خواهد بود:

In [12]:

```
1 parent()
```

```
Printing from the parent() function
Printing from the second_child() function
Printing from the first_child() function
```

توجه کنید که ترتیب تعریف توابع داخلی اهمیتی ندارد. مانند هر تابع دیگری، چاپ فقط در زمانی انجام می‌شود که توابع داخلی اجرا می‌شوند.

علاوه بر این، توابع داخلی تا زمانی که تابع والد (parent()) فراخوانی نشده باشد، فرزندان تعریف نمی‌شوند. آن‌ها به عنوان متغیرهای محلی در parent() وجود دارند. تلاش کنید تابع first_child() را فراخوانی کنید. باید خطایی دریافت کنید:

In [13]:

```
1 first_child()
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[13], line 1
----> 1 first_child()

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

هربار که تابع parent() را فراخوانی می‌کنید، توابع داخلی first_child() و second_child() نیز فراخوانی می‌شوند. اما به دلیل اینکه دارای محدوده محلی هستند، بیرون از تابع parent() در دسترس نیستند.

برگرداندن توابع از داخل توابع

پایتون به شما اجازه می‌دهد تا از توابع به عنوان مقادیر برگشتی استفاده کنید. مثال زیر یکی از توابع داخلی را از تابع parent() برمی‌گرداند.

In [14]:

```
1 def parent(num):
2     def first_child():
3         return "Hi, I am Emma"
4 
5     def second_child():
6         return "Call me Liam"
7 
8     if num == 1:
9         return first_child
10    else:
11        return second_child
```

توجه کنید که تابع first_child را بدون پرانتز برمی‌گردانید. به یاد داشته باشید که این بدان معناست که شما یک ارجاع به تابع first_child را برمی‌گردانید. در مقابل، first_child() با استفاده از پرانتز به نتیجه ارزیابی تابع اشاره می‌کند. این می‌تواند در مثال زیر مشاهده شود:

In [15]:

```
1 first = parent(1)
```

In [16]:

```
1 second = parent(2)
```

In [18]:

```
1 parent
```

Out[18]:

```
<function __main__.parent(num)>
```

In [17]:

```
1 first
```

Out[17]:

```
<function __main__.parent.<locals>.first_child()>
```

In [19]:

```
1 second
```

Out[19]:

```
<function __main__.parent.<locals>.second_child()>
```

خروجی نسبتاً رمزآمیز به این معناست که متغیر اول به تابع محلی `first_child` داخل `parent` اشاره دارد، در حالی که متغیر دوم به تابع `second_child` اشاره دارد.

اکنون می‌توانید از `first` و `second` به عنوان توابع عادی استفاده کنید، اگرچه توابعی که به آنها اشاره دارند نمی‌توانند به صورت مستقیم دسترسی داشته باشند:

In [20]:

```
1 first()
```

Out[20]:

```
'Hi, I am Emma'
```

In [21]:

```
1 second()
```

Out[21]:

```
'Call me Liam'
```

در نهایت، توجه کنید که در مثال قبل توابع داخلی را در داخل تابع والد (`parent`) اجرا کرده بودید، به عنوان مثال `first_child()` . با این حال، در این مثال آخر، شما پرانتز را به توابع داخلی (`first_child`) در هنگام برگشت برنامه اضافه نکرده‌اید. به این ترتیب، شما یک ارجاع به هر تابع بدست آورده‌اید که در آینده می‌توانید آن را فراخوانی کنید. منطقی است؟

دکوراتورهای ساده

اکنون که دیدید توابع همانند هر شیء دیگری در پایتون هستند، آماده‌اید به سمت جادوگری بروید که دکوراتور پایتون است. همین الان با یک مثال شروع کنیم:

In [29]:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")
def be_awesome():
    print(f"Yo , together we are the awesomest!")


be_awesome = my_decorator(be_awesome)
say_whee = my_decorator(say_whee)

```

می‌توانید حدس بزنید که چه اتفاقی می‌افتد وقتی تابع say_whee را صدا می‌زنید؟ امتحان کنید:

In [30]:

```
say_whee()
```

```
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

برای درک آنچه در اینجا اتفاق می‌افتد، به مثال‌های قبلی نگاهی بیندازید. به طور کامل همان چیزی که تاکنون یاد گرفته‌اید را به کار می‌گیریم.

دکوراسیون مورد نظر در خط زیر رخ می‌دهد:

In [ ]:

```
say_whee = my_decorator(say_whee)
```

در واقع، نام say_whee اکنون به تابع داخلی wrapper اشاره می‌کند. به یاد داشته باشید که وقتی my_decorator(say_whee) را صدا می‌زنید، تابع wrapper را به عنوان یک تابع برمی‌گردانید.

In [31]:

```
say_whee
```

Out[31]:

```
<function __main__.my_decorator.<locals>.wrapper()>
```

با این حال، تابع wrapper یک ارجاع به تابع اصلی say_whee را به عنوان func دارد و این تابع را بین دو فراخوانی print فرا می‌خواند.

به طور ساده می‌توان گفت: دکوراتورها تابعی را دربر می گیرند و رفتار آن را تغییر می‌دهند.

قبل از ادامه، نگاهی به مثال دوم بیندازیم. به این دلیل که تابع wrapper یک تابع معمولی در پایتون است، شیوه‌ای که دکوراتور رفتار یک تابع را تغییر می‌دهد، به طور پویا می‌تواند تغییر کند. به منظور از همسایگانتان خیلی آزرده نشوید، در مثال زیر تنها در طول روز کد دکوره شده اجرا می‌شود:

In [32]:

```
from datetime import datetime

def not_during_the_night(func):
    def wrapper():
        if 7 <= datetime.now().hour < 22:
            func()
        else:
            pass  # Hush, the neighbors are asleep
    return wrapper

def say_whee():
    print("Whee!")

say_whee = not_during_the_night(say_whee)
```

اگر بعد از خوابیدن همسایه‌ها، تابع say_whee را فراخوانی کنید، هیچ اتفاقی نخواهد افتاد:

In [33]:

```
say_whee()
```

```
Whee!
```

Syntactic Sugar!

توابع را می‌توان با استفاده از سینتکس شکرآمیزتری دکوره کرد. ابتدا باید به یاد داشته باشید که در این حالت، نام تابع را سه بار تایپ می‌کنید. به علاوه، دکوراسیون یکم در زیر تعریف تابع پنهان می‌شود.

در عوض، پایتون به شما اجازه می‌دهد تا از دکوراتورها با استفاده از نماد @ که گاهی به آن نماد "پای" می‌گویند، به ساده‌ترین شکل ممکن استفاده کنید. مثال زیر دقیقاً همان کاری را انجام می‌دهد که در مثال اول دکوراتور صورت گرفت:

In [ ]:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper


# say_whee = my_decorator(say_whee)
@my_decorator
def say_whee():
    print("Whee!")
```

بنابراین، `@my_decorator` تنها یک راه آسان برای گفتن `say_whee = my_decorator(say_whee)` است. این روشی است که یک دکوراتور را روی یک تابع اعمال می‌کنید.

استفاده مجدد از دکوراتورها

بیاد آورید که یک دکوراتور فقط یک تابع معمولی در پایتون است. تمام ابزارهای معمول برای قابلیت استفاده مجدد آسان در دسترس هستند. بیایید دکوراتور را به یک ماژول جداگانه منتقل کنیم که در بسیاری از توابع دیگر قابل استفاده باشد.

یک فایل با نام decorators.py ایجاد کنید و محتوای زیر را در آن قرار دهید:

In [ ]:

```
#  decorators.py


def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

توجه: می‌توانید نام تابع داخلی را هر چیزی که می‌خواهید بگذارید و نام عمومی مانند `wrapper()` به طور معمول قابل قبول است. شما در این مقاله بسیاری از دکوراتورها را خواهید دید. برای تمیز نگه داشتن آن‌ها، ما نام تابع داخلی را با همان نام دکوراتور اما با پیشوند `_wrapper` نامگذاری می‌کنیم.

حالا می‌توانید از این دکوراتور جدید در فایل‌های دیگر با استفاده از import عادی استفاده کنید:

In [34]:

```
from decorators import do_twice

@do_twice
def say_whee():
    print("Whee!")
```

هنگام اجرای این مثال، باید ببینید که تابع اصلی `say_whee()` دو بار اجرا می‌شود.

In [35]:

```
1 say_whee()
```

```
Whee!
Whee!
```

Decorating Functions With Arguments

فرض کنید یک تابع دارید که چندین آرگومان را می‌پذیرد. آیا هنوز می‌توانید آن را دکور کنید؟ بیایید امتحان کنیم:

In [36]:

```
1 from decorators import do_twice
2
3 @do_twice
4 def greet(name):
5     print(f"Hello {name}")
```

متأسفانه، اجرای این کد خطا را برمی‌گرداند:

In [37]:

```
1 greet("World")
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[37], line 1
----> 1 greet("World")

TypeError: do_twice.<locals>.wrapper_do_twice() takes 0 positional argumen
ts but 1 was given
```

مشکل این است که تابع داخلی `wrapper_do_twice()` هیچ آرگومانی را نمی‌پذیرد، اما `"name="World` به آن ارسال شده است. می‌توانید این مشکل را با اینکه به `wrapper_do_twice()` اجازه دهید یک آرگومان را بپذیرد، رفع کنید، اما در این صورت برای تابع `say_whee()` که قبلاً ایجاد کرده بودید کار نخواهد کرد.

راه حل این استفاده از `*args` و `**kwargs` در تابع داخلی wrapper است. در این صورت، این تابع تعداد خاصی از آرگومان‌های موقعیتی و کلمه‌ای را پذیرفته و آن‌ها را به تابعی که دکور می‌کند منتقل می‌کند. به عنوان مثال decorators.py را به صورت زیر بازنویسی کنید:

In [ ]:

```
1 def do_twice(func):
2     def wrapper_do_twice(*args, **kwargs):
3         func(*args, **kwargs)
4         func(*args, **kwargs)
5     return wrapper_do_twice
```

تابع داخلی `wrapper_do_twice()` در حال حاضر تعداد دلخواهی از آرگومان‌ها را پذیرفته و آن‌ها را به تابعی که دکور می‌کند منتقل می‌کند. حالا هر دو مثال `say_whee()` و `greet()` کار می‌کنند.

In [1]:

```
from decorators import do_twice

@do_twice
def say_whee():
    print("Whee!")

@do_twice
def greet(name):

    print(f"Hello {name}")
```

In [2]:

```
say_whee()
```

```
Whee!
Whee!
```

In [3]:

```
greet("World")
```

```
Hello World
Hello World
```

In [4]:

```
greet
```

Out[4]:

```
<function decorators.do_twice.<locals>.wrapper_do_twice(*args, **kwargs)>
```

بازگرداندن مقادیر از توابع دکوراتور شده

مقدار بازگشتی از توابع تدکوراتور شده چه اتفاقی می‌افتد؟ خب، این بستگی به دکوراتور دارد. فرض کنید یک تابع ساده را به شکل زیر دکوراتور کنید:

In [1]:

```
from decorators import do_twice

@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"
```

سعی کنید از آن استفاده کنید:

In [2]:

```
hi_adam = return_greeting("Adam")
```

```
Creating greeting
Creating greeting
```

In [3]:

```
print(hi_adam)
```

```
None
```

اوه، دکوراتور شما مقدار بازگشتی تابع را خورده است.

زیرا تابع `do_twice_wrapper()` به صورت صریح یک مقدار را بازنمی‌گرداند، فراخوانی `return_greeting("Adam")` بازگشتی از `None` داشت.

برای رفع این مشکل، باید مطمئن شوید که تابع `wrapper` مقدار بازگشتی تابع دکوراتور شده را برمی‌گرداند. فایل `decorators.py` را تغییر دهید:

In [ ]:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

In [1]:

```
from decorators import do_twice

@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"
```

مقدار بازگشتی از آخرین اجرای تابع برگردانده می‌شود.

In [2]:

```
return_greeting("Adam")
```

```
Creating greeting
Creating greeting
```

Out[2]:

```
'Hi Adam'
```

مشخص کردن منشا یک تابع

یکی از مزایای بزرگ کار با زبان پایتون، به ویژه در محیط تعاملی آن، قابلیت تفحص قدرتمند آن است. تفحص، قابلیت یک شیء است که به شیء مربوطه اجازه می‌دهد تا در زمان اجرا درباره ویژگی‌های خود اطلاعاتی داشته باشد. به عنوان مثال، یک تابع نام و مستندات خود را می‌شناسد:

In [3]:

```
1 print
```

Out[3]:

```
<function print>
```

In [4]:

```
1 print.__name__
```

Out[4]:

```
'print'
```

In [5]:

```
1 help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdou
t.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

قابلیت تفحص برای توابعی که خودتان تعریف می‌کنید نیز کار می‌کند:

In [6]:

```
1 @do_twice
2 def say_whee():
3     print("Whee!")
4
5 say_whee
```

Out[6]:

```
<function decorators.do_twice.<locals>.wrapper_do_twice(*args, **kwargs)>
```

In [7]:

```
say_whee.__name__
```

Out[7]:

```
'wrapper_do_twice'
```

In [8]:

```
help(say_whee)
```

```
Help on function wrapper_do_twice in module decorators:

wrapper_do_twice(*args, **kwargs)
```

با این حال، پس از دکوراتور، تابع `say_whee()` بسیار گیج شده است درباره هویت خود. اکنون به عنوان تابع داخلی `wrapper_do_twice()` درون دکوراتور `do_twice()` اعلام می‌شود. اگرچه این اصولاً درست است، اما اطلاعات خیلی مفیدی نیست.

برای رفع این مشکل، دکوراتورکننده‌ها باید از دکوراتورکننده `@functools.wraps` استفاده کنند که اطلاعات درباره تابع اصلی را حفظ می‌کند. دوباره فایل decorators.py را به روز کنید:

In [9]:

```
import functools

def do_twice(func):
    @functools.wraps(func)
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

شما نیازی به تغییر هیچ چیز در تابع دکوراتور شده `say_whee()` ندارید:

In [1]:

```
from decorators import do_twice

@do_twice
def say_whee():
    print("Whee!")

say_whee
```

Out[1]:

```
<function __main__.say_whee()>
```

خیلی بهتر! حالا `say_whee()` پس از تزئین همچنان خودش است.

In [2]:

```
1 say_whee.__name__
```

Out[2]:

```
'say_whee'
```

In [3]:

```
1 help(say_whee)
```

```
Help on function say_whee in module __main__:

say_whee()
```

چند مثال واقعی

بیایید به چند مثال مفید دیگر از دکوراتورها نگاهی بیندازیم. خواهید دید که آنها اغلب الگویی مشابه با آنچه تاکنون یاد گرفته‌اید را دنبال می‌کنند:

این الگوی قالب بسیار خوبی برای ساخت دکوراتورهای پیچیده‌تر است.

توجه: در مثال‌های بعدی، فرض می‌کنیم این دکوراتورها را در فایل decorators.py خود ذخیره کرده‌اید. به یاد داشته باشید که می‌توانید تمام مثال‌ها در این آموزش را دانلود کنید.

تایمینگ توابع

بیایید با ایجاد یک دکوراتور با نام `@timer` شروع کنیم. این دکوراتور زمانی که یک تابع را اجرا می‌کند، زمان اجرای آن را اندازه‌گیری کرده و مدت زمان را در کنسول چاپ می‌کند. کد زیر را ببینید:

In [5]:

```
import functools, time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*arg, **kwargs):
        start_time = time.perf_counter()
        value = func(*arg, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer



@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i ** 2 for i in range(10000)])
```

این دکوراتور با ذخیره زمان در لحظه شروع اجرای تابع (در خط مشخص شده با # 1) و زمان پایان آن (در خط مشخص شده با # 2) کار می‌کند. مدت زمانی که تابع طول می‌کشد، تفاوت بین دو زمان است (در خط مشخص شده با # 3). ما از تابع `time.perf_counter()` استفاده می‌کنیم که عملکرد خوبی در اندازه‌گیری فواصل زمانی دارد. در زیر چند نمونه از زمان‌ها را مشاهده می‌کنید:

In [6]:

```
waste_some_time(1)
```

```
Finished 'waste_some_time' in 0.0028 secs
```

In [7]:

```
waste_some_time(999)
```

```
Finished 'waste_some_time' in 2.7858 secs
```

In [8]:

```
waste_some_time
```

Out[8]:

```
<function __main__.waste_some_time(num_times)>
```

با اجرای آن به‌صورت خودکار، کد را خط به خط اجرا کنید. اطمینان حاصل کنید که نحوه عملکرد آن را متوجه شده‌اید. اگر نتوانستید آن را درک کنید، نگران نباشید. دکوراتورها موجودات پیشرفته‌ای هستند. سعی کنید روی نحوه عملکرد آنها بیشتر تمرکز کنید یا یک flowchart برنامه را بکشید.

توجه: دکوراتور `@timer` عالی است اگر شما فقط می‌خواهید یک ایده از زمان اجرای توابع خود داشته باشید. اگر می‌خواهید اندازه‌گیری‌های دقیق‌تری از کد خود انجام دهید، به‌جای آن باید ماژول `timeit` را در کتابخانه استاندارد در نظر بگیرید. این ماژول گردآوری زباله را به‌طور موقت غیرفعال می‌کند و چندین آزمایش اجرا می‌کند تا نویز از callهای سریع تابع را حذف کند.

اشکال‌زدایی کد

دکوراتور @debug زیر هر بار که تابعی فراخوانی می‌شود، آرگومان‌هایی که با آن فراخوانی شده است و همچنین مقدار بازگشتی آن را چاپ می‌کند:

In [9]:

```
x = [1,2,3]
repr(x)
```

Out[9]:

```
'[1, 2, 3]'
```

In [11]:

```
y  = {'age' : 100, 'name' : 'John'}
repr(y)
```

Out[11]:

```
"{'age': 100, 'name': 'John'}"
```

In [12]:

```
# Calling make_greeting('Richard', age=112)
# 'make_greeting' returned 'Whoa Richard! 112 already, you are growing up!'
import functools

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        args_repr = [repr(a) for a in args] # ['Richard', 'handsome']
        kwargs_repr = [f"{k}={v!r}" for k,v in kwargs.items()] # ["age=112", "scors=
        arguments = args_repr + kwargs_repr # ['Richard', 'handsome', "age=112", "sc
        signature = ", ".join(arguments) # "Richard, handsome, age=112, scors=[12,29

        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")

        return value
    return wrapper_debug


```

امضا توسط اتصال نمایش رشته‌های مربوط به همه آرگومان‌ها ایجاد می‌شود. اعداد در لیست زیر با نظرات شماره‌گذاری شده در کد مطابقت دارند:

- یک لیست از آرگومان‌های موقعیتی ایجاد می‌شود. از تابع repr() برای به دست آوردن یک رشته مناسب برای نمایش هر آرگومان استفاده می‌شود.
- یک لیست از آرگومان‌های کلیدواژه‌ای ایجاد می‌شود. عبارت f-string هر آرگومان را به صورت کلید=مقدار فرمت می‌کند، جایی که مشخص‌کننده !r به این معنی است که از تابع repr() برای نمایش مقدار استفاده شده است.
- لیست‌های آرگومان‌های موقعیتی و کلیدواژه‌ای به صورت یک رشته امضا با هر آرگومان که توسط یک کاما جدا شده‌اند، ترکیب می‌شوند.

- مقدار بازگشتی پس از اجرای تابع چاپ می‌شود.

بیایید با استفاده از این دکوراتور به یک تابع ساده با یک آرگومان موقعیتی و یک آرگومان کلیدواژه‌ای اعمال کنیم:

In [13]:

```
@debug
def make_greeting(name, age=None):
    if age is None:
        return f"Howdy {name}!"
    else:
        return f"Whoa {name}! {age} already, you are growing up!"
```

توجه کنید که دکوراتور @debug امضا و مقدار بازگشتی تابع make_greeting() را چاپ می‌کند.

In [14]:

```
make_greeting("Benjamin")
```

```
Calling make_greeting('Benjamin')
'make_greeting' returned 'Howdy Benjamin!'
```

Out[14]:

```
'Howdy Benjamin!'
```

In [15]:

```
make_greeting("Richard", age=112)
```

```
Calling make_greeting('Richard', age=112)
'make_greeting' returned 'Whoa Richard! 112 already, you are growing up!'
```

Out[15]:

```
'Whoa Richard! 112 already, you are growing up!'
```

In [16]:

```
make_greeting(name="Dorrisile", age=116)
```

```
Calling make_greeting(name='Dorrisile', age=116)
'make_greeting' returned 'Whoa Dorrisile! 116 already, you are growing u
p!'
```

Out[16]:

```
'Whoa Dorrisile! 116 already, you are growing up!'
```

این مثال در ابتدا به ظاهر مفید نمی‌آید زیرا دکوراتور @debug فقط تکرار می‌کند آنچه که به تازگی نوشته‌اید. اما وقتی به توابع کمکی کوچکی اعمال شود که به طور مستقیم خودتان فراخوانی نمی‌کنید، قدرتمندتر است.

مثال زیر یک تقریب از ثابت ریاضی e را محاسبه می‌کند:

In [17]:

```python
import math
#from decorators import debug

# Apply a decorator to a standard library function
math.factorial = debug(math.factorial)

def approximate_e(terms=18):
    return sum(1 / math.factorial(n) for n in range(terms))
```

این مثال نشان می‌دهد چگونه می‌توانید یک دکوراتور را به یک تابعی که قبلاً تعریف شده است، اعمال کنید. تقریب e بر اساس گسترش سری زیر است:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots = \frac{1}{1} + \frac{1}{1} + \frac{1}{1 \cdot 2} + \cdots$$

هنگام فراخوانی تابع approximate_e() ، می‌توانید دکوراتور @debug در حال کار را ببینید.

In [19]:

```python
approximate_e(10)
```

```
Calling factorial(0)
'factorial' returned 1
Calling factorial(1)
'factorial' returned 1
Calling factorial(2)
'factorial' returned 2
Calling factorial(3)
'factorial' returned 6
Calling factorial(4)
'factorial' returned 24
Calling factorial(5)
'factorial' returned 120
Calling factorial(6)
'factorial' returned 720
Calling factorial(7)
'factorial' returned 5040
Calling factorial(8)
'factorial' returned 40320
Calling factorial(9)
'factorial' returned 362880
```

Out[19]:

```
2.7182815255731922
```

در این مثال، تقریب مناسبی از مقدار واقعی e = 2.718281828 حاصل می‌شود، با اضافه کردن تنها ۵ عبارت.

In [22]:

```
import functools, time

def slow_down(func):
    """Sleep 1 second before calling the function"""
    @functools.wraps(func)
    def wrapper_slow_down(*args, **kwargs):
        time.sleep(1)
        return func(*args, **kwargs)
    return wrapper_slow_down

@slow_down
def countdown(from_number):
    if from_number < 1:
        print('Liftoff')
    else:
        print(from_number)
        countdown(from_number - 1)
```

برای دیدن تأثیر دکوراتور `@slow_down` واقعاً باید مثال را خودتان اجرا کنید:

In [24]:

```
countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
Liftoff
```

دکوراتور `@slow_down` همیشه به مدت یک ثانیه تاخیر می‌کند. بعدها خواهید دید که چگونه با ارسال یک آرگومان به دکوراتور می‌توانید نرخ را کنترل کنید.

Fancy Decorators

تا اینجا شما چگونگی ایجاد دکوراتورهای ساده را دیده‌اید. شما در حال حاضر درک خوبی از آنچه دکوراتورها هستند و چگونه کار می‌کنند دارید. به آرامی از این مقاله استفاده کرده و تمرین کردن همه چیزی که یاد گرفته‌اید را آزادانه انجام دهید.

در بخش دوم این آموزش، ویژگی‌های پیشرفته‌تری را بررسی خواهیم کرد، از جمله استفاده از موارد زیر:

- دکوراتورها در کلاس‌ها
- چندین دکوراتور بر روی یک تابع یا دکوراتورهای تودرتو
- دکوراتورها با آرگومان‌ها
- دکوراتورهایی با وضعیت
- کلاس‌ها به عنوان دکوراتورها
- دکوراتور کلاس‌ها

دو روش متفاوت برای استفاده از دکوراتورها در کلاس‌ها وجود دارد. اولین روش بسیار شبیه به آنچه که قبلاً با توابع انجام داده‌اید است: شما می‌توانید متدهای یک کلاس را تزئین کنید. این یکی از دلایل معرفی دکوراتورها در گذشته بود.

تعدادی از دکوراتورهای معمول استفاده شده که در واقع در پایتون داخلی هستند عبارتند از @classmethod یا @staticmethod و @property . دکوراتورهای @classmethod و @staticmethod برای تعریف متدها در فضای نام یک کلاس استفاده می‌شوند که به هیچ نمونه خاصی از آن کلاس متصل نیستند. دکوراتور @property برای سفارشی‌سازی getter و setter برای ویژگی‌های کلاس استفاده می‌شود. به مثال زیر برای استفاده از این دکوراتورها مراجعه کنید.

بیایید یک کلاس را تعریف کنیم که برخی از متدهای آن را با استفاده از دکوراتورهای @debug و @timer از قبل تعریف شده دکور کنیم:

In [1]:

```
from decorators import debug, timer

class TimeWaster:

    @debug
    def __init__(self, max_num):
        self.max_num = max_num

    @timer
    def waste_time(self, num_times):
        for _ in range(num_times):
            sum([i ** 2 for i in range(num_times)])
```

با استفاده از این کلاس، تأثیر دکوراتورها را مشاهده می‌کنید:

In [2]:

```
tw = TimeWaster(1000)
```

```
Calling __init__(<__main__.TimeWaster object at 0x0000025C1972FFD0>, 1000)
'__init__' returned None
```

In [3]:

```
tw.waste_time(999)
```

```
Finished 'waste_time' in 0.2529 secs
```

روش دیگری برای استفاده از دکوراتورها در کلاس‌ها، دکوراتور کل کلاس است. این کار به عنوان مثال در ماژول dataclasses جدید در پایتون 3.7 انجام می‌شود:

In [ ]:

```
from dataclasses import dataclass

# PlayingCard = dataclass(PlayingCard)
@dataclass
class PlayingCard:
    rank: str
    suit: str
```

معنای دستور نحوی مشابه دکوراتورهای تابع است. در مثال بالا، شما می‌توانستید با نوشتن `PlayingCard = dataclass(PlayingCard)` ، تزئین کنید.

استفاده رایجی از دکوراتورهای کلاس، جایگزینی ساده‌تری برای برخی موارد استفاده از متاکلاس‌ها است. در هر دو مورد، شما در حال تغییر تعریف یک کلاس به صورت پویا هستید.

نوشتن یک دکوراتور کلاس بسیار شبیه به نوشتن یک دکوراتور تابع است. تنها تفاوت این است که دکوراتور یک کلاس را به عنوان آرگومان دریافت می‌کند و نه یک تابع. در واقع، همه دکوراتورهایی که در بالا دیدید، می‌توانند به عنوان دکوراتورهای کلاس عمل کنند. هنگام استفاده از آنها بر روی یک کلاس به جای یک تابع، تأثیر آنها ممکن است همانطور که می‌خواهید نباشد. در مثال زیر، دکوراتور `@timer` بر روی یک کلاس اعمال می‌شود:

In [4]:

```
from decorators import timer

@timer
class TimeWaster:

    def __init__(self, max_num):
        self.max_num = max_num

    def waste_time(self, num_times):
        for _ in range(num_times):
            sum([i ** 2 for i in range(num_times)])
```

تزئین یک کلاس متدهای آن را تزئین نمی‌کند. به یاد داشته باشید که `@timer` فقط اختصاری برای `TimeWaster = timer(TimeWaster)` است.

در اینجا، `@timer` فقط زمانی را که برای نمونه‌سازی کلاس لازم است، اندازه‌گیری می‌کند.

In [5]:

```
tw = TimeWaster(1000)
```

```
Finished 'TimeWaster' in 0.0000 secs
```

In [6]:

```
tw.waste_time(999)
```

بعدها، یک مثال از تعریف یک دکوراتور کلاس مناسب، به نام `@singleton` ، را خواهید دید که اطمینان می‌دهد تنها یک نمونه از یک کلاس وجود داشته باشد.

دکوراتورهای تودرتو

In [11]:

```
from decorators import debug, do_twice

@debug
@do_twice
def greet(name):
    print(f"Hello {name}")

# greet = do_twice(greet)
# greet = debug(do_twice(greet))
```

لطفاً در نظر بگیرید که دکوراتورها به ترتیبی که در لیست آنها قرار دارند، اجرا می‌شوند. به عبارت دیگر، @debug تماس می‌گیرد با @do_twice که تماس می‌گیرد با greet() ، به عبارت دیگر debug(do_twice(greet())) :

In [12]:

```
greet("Eva")
```

```
Calling greet('Eva')
Hello Eva
Hello Eva
'greet' returned None
```

تفاوت را مشاهده کنید اگر ترتیب @debug و @do_twice را تغییر دهیم:

In [13]:

```
from decorators import debug, do_twice

@do_twice
@debug
def greet(name):
    print(f"Hello {name}")
```

در این حالت، @do_twice همچنین بر روی @debug اعمال خواهد شد.

In [14]:

```
greet("Eva")
```

```
Calling greet('Eva')
Hello Eva
'greet' returned None
Calling greet('Eva')
Hello Eva
'greet' returned None
```

ارسال آرگومان به دکوراتورها

گاهی اوقات، ارسال آرگومان‌ها به دکوراتورهای شما مفید است. به عنوان مثال، `@do_twice` می‌تواند به یک دکوراتور `@repeat(num_times)` گسترش یابد. تعداد بار اجرای تابع تزئین شده را می‌توان به عنوان آرگومان ارائه کرد.

این به شما امکان می‌دهد که مانند زیر عمل کنید:

In [ ]:

```
@repeat(num_times=4)
def greet(name):
    print(f"Hello {name}")
```

In [ ]:

```
greet("World")
```

In [15]:

```
import functools

def repeat(num_times):

    @functools.wraps(func)
    def wrapper_repeat(*arg, **kwargs):
        for _ in range(num_times):
            value = func(*args, **kwargs)
        return value
    return wrapper_repeat

```

In [16]:

```
@repeat(num_times=4)
def greet(name):
    print(f"Hello {name}")
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[16], line 1
----> 1 @repeat(num_times=4)
      2 def greet(name):
      3     print(f"Hello {name}")

Cell In[15], line 5, in repeat(num_times)
      3 def repeat(num_times):
----> 5     @functools.wraps(func)
      6     def wrapper_repeat(*arg, **kwargs):
      7         for _ in range(num_times):
      8             value = func(*args, **kwargs)

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

درباره اینکه چگونه این کار را انجام دهید، فکر کنید.

تا به حال، نام نوشته شده پس از @ به یک شیء تابع اشاره کرده است که می‌تواند با یک تابع دیگر فراخوانی شود. برای انطباق، نیاز دارید repeat(num_times=4) یک شیء تابع را برگرداند که به عنوان یک دکوراتور عمل کند. خوشبختانه، شما در حال حاضر می‌دانید چگونه توابع را برگردانید! به طور کلی، شما چیزی شبیه به زیر را می‌خواهید:

In [ ]:

```
def repeat(num_times):
    def decorator_repeat(func):
        ...  # Create and return a wrapper function
    return decorator_repeat
```

معمولاً، دکوراتور یک تابع پوششی را ایجاد و برمی‌گرداند، بنابراین نوشتن مثال را به طور کامل درون یک def اضافی قرار می‌دهیم که آرگومان‌های دکوراتور را رسیدگی می‌کند. با یک تابع درونی بیشتر شروع کنیم:

In [17]:

```
def repeat(num_times):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat
    return decorator_repeat
```

این کمی پریشان‌کننده به نظر می‌رسد، اما ما تنها الگوی دکوراتور مشابهی که تاکنون چندین بار دیده‌اید را درون یک def اضافی قرار داده‌ایم که آرگومان‌های دکوراتور را رسیدگی می‌کند. بیایید با تابع درونی شروع کنیم:

In [ ]:

```
def wrapper_repeat(*args, **kwargs):
    for _ in range(num_times):
        value = func(*args, **kwargs)
    return value
```

این تابع wrapper_repeat() آرگومان‌های دلخواهی را می‌پذیرد و مقدار تابع تزئین شده یعنی func() را برمی‌گرداند. این تابع پوششی همچنین حلقه‌ای را شامل می‌شود که تابع تزئین شده را num_times بار فراخوانی می‌کند. این با تفاوتی نسبت به تابع‌های پوششی قبلی که دیده‌اید، نیست، به جز استفاده از پارامتر num_times که باید از بیرون ارسال شود.

یک قدم بیرون‌تر، تابع دکوراتور است:

In [ ]:

```python
def decorator_repeat(func):
    @functools.wraps(func)
    def wrapper_repeat(*args, **kwargs):
        ...
    return wrapper_repeat
```

دوباره، `decorator_repeat()` دقیقاً شبیه به توابع دکوراتوری است که قبلاً نوشته‌اید، به جز اینکه با یک نام متفاوت است. این بدلیل این است که نام پایه را (repeat) برای تابع بیرونی که کاربر فراخوانی خواهد کرد، رزرو کرده‌ایم.

همانطور که قبلاً دیده‌اید، تابع بیرونی یک ارجاع به تابع دکوراتور را برمی‌گرداند:

In [ ]:

```python
def repeat(num_times):
    def decorator_repeat(func):
        ...
    return decorator_repeat
```

در تابع `repeat()` چند نکته‌ی پنهانی وجود دارد:

- تعریف `decorator_repeat()` به عنوان یک تابع درونی به این معناست که `repeat()` به یک شیء تابع ارجاع خواهد داد - decorator_repeat. قبلاً از repeat بدون پرانتز برای ارجاع به شیء تابع استفاده کردیم. پرانتزهای اضافی هنگام تعریف دکوراتورهایی که آرگومان می‌پذیرند، ضروری است.
- آرگومان `num_times` به ظاهر در `repeat()` استفاده نمی‌شود. اما با ارسال `num_times` ، یک بسته‌بندی ایجاد می‌شود که مقدار `num_times` تا زمان استفاده بعدی توسط `wrapper_repeat()` ذخیره می‌شود.

با همه چیز آماده شده، بیایید ببینیم آیا نتایج مورد انتظار هستند:

In [18]:

```python
@repeat(num_times=4) # 1) => decorator_repeat(func)
                     # 2) @decorator_reapte
                     # 3) wrapper_repeat(*arg, **kwargs)
def greet(name):
    print(f"Hello {name}")
```

In [19]:

```python
greet("World")
```

```
Hello World
Hello World
Hello World
Hello World
```

دکوراتورهای دارای وضعیت (Stateful Decorators)

گاهی اوقات، داشتن یک دکوراتور که قادر به پیگیری وضعیت باشد مفید است. به عنوان مثال ساده، ما یک دکوراتور ایجاد می‌کنیم که تعداد بارهایی که یک تابع فراخوانی می‌شود را شمارش می‌کند.

توجه: در ابتدای این راهنما، درباره توابع خالص که بر اساس آرگومان‌های داده شده مقداری را برمی‌گردانند، صحبت کردیم. دکوراتورهای دارای وضعیت کاملاً متضاد هستند که در آن مقدار بازگشتی به وضعیت کنونی و همچنین آرگومان‌های داده شده بستگی دارد.

In [20]:

```
import functools

def count_calls(func):
    @functools.wraps(func)
    def wrapper_count_calls(*args, **kwargs):
        wrapper_count_calls.num_calls += 1
        print(f"Call {wrapper_count_calls.num_calls} of {func.__name__!r}")
        return func(*args, **kwargs)
    wrapper_count_calls.num_calls = 0
    return wrapper_count_calls

@count_calls
def say_whee():
    print("Whee!")
```

وضعیت - تعداد فراخوانی‌های تابع - در ویژگی `.num_calls` روی تابع پوشش‌داده می‌شود. اینجا تأثیر استفاده از آن را می‌بینید:

In [21]:

```
say_whee()
```

```
Call 1 of 'say_whee'
Whee!
```

In [22]:

```
say_whee()
```

```
Call 2 of 'say_whee'
Whee!
```

In [23]:

```
say_whee.num_calls
```

Out[23]:

```
2
```

استفاده از کلاس‌ها به عنوان دکوراتورها

راه معمول برای نگهداری وضعیت استفاده از کلاس‌ها است. در این بخش، خواهید دید که چگونه مثال `@count_calls` را از بخش قبل با استفاده از یک کلاس به عنوان تدکوراتور، بازنویسی کنید.

به یاد داشته باشید که سینتکس دکوراتور `@my_decorator` تنها یک روش آسان‌تر برای گفتن `func = my_decorator(func)` است. بنابراین، اگر `my_decorator` یک کلاس باشد، باید `func` را به عنوان آرگومان در متد `.__init__()` آن دریافت کند. علاوه بر این، نمونه کلاس باید قابل فراخوانی باشد تا بتواند به جای تابع تزئین‌شده قرار بگیرد.

برای اینکه یک نمونه کلاس قابل فراخوانی باشد، شما باید متد ویژه `__call__()` را پیاده‌سازی کنید:

In [24]:

```
class Counter:
    def __init__(self, start = 0):
        self.count = start

    def __call__(self):
        self.count += 1
        print(f"Current count is {self.count}")
```

متد `__call__()` هر بار که تلاش می‌کنید یک نمونه از کلاس را صدا بزنید اجرا می‌شود:

In [25]:

```
counter = Counter()
```

In [26]:

```
counter()
```

```
Current count is 1
```

In [27]:

```
counter()
```

```
Current count is 2
```

In [28]:

```
counter.count
```

Out[28]:

```
2
```

بنابراین، پیاده‌سازی معمول برای یک کلاس دکوراتور نیازمند پیاده‌سازی متد `__init__()` و `__call__()` است:

In [31]:

```
import functools

class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.num_calls = 0
        self.func = func

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__!r}")
        return self.func(*args, **kwargs)

@CountCalls
def say_whee():
    print("Whee!")
```

متد `__init__()` باید یک مرجع به تابع را ذخیره کند و می‌تواند هرگونه مقداردهی اولیه مورد نیاز را انجام دهد. متد `__call__()` به جای تابع تزئین‌شده فراخوانی می‌شود. این تقریباً همان کاری را انجام می‌دهد که تابع `wrapper()` در مثال‌های قبلی ما انجام می‌دهد. توجه کنید که باید از تابع `functools.update_wrapper()` به جای `@functools.wraps` استفاده کنید.

دکوراتور `@CountCalls` همانطور که در بخش قبل استفاده شده است، کار می‌کند.

In [32]:

```
say_whee()
```

```
Call 1 of 'say_whee'
Whee!
```

In [33]:

```
say_whee()
```

```
Call 2 of 'say_whee'
Whee!
```

In [34]:

```
say_whee.num_calls
```

Out[34]:

```
2
```

مثال‌های دیگر در کاربرد واقعی

تا اینجا راه زیادی را طی کرده‌ایم و توانسته‌ایم ببینیم که چگونه می‌توانیم انواع دکوراتورها را ایجاد کنیم. بیایید آن را به پایان برسانیم و با استفاده از دانش جدید خود، چندین مثال دیگری را ایجاد کنیم که ممکن است در دنیای واقعی مفید باشند.

کند کردن کد، دوباره بررسی شود

همانطور که قبلاً توضیح داده شد، نسخه قبلی ما از `@slow_down` همیشه به مدت یک ثانیه منتظر می‌ماند. اکنون می‌دانید چگونه پارامترها را به دکوراتورها اضافه کنید، بنابراین بیایید `@slow_down` را با استفاده از یک آرگومان اختیاری بنویسیم که مدت زمان خواب را کنترل می‌کند:

In [35]:

```
import functools, time

def slow_down(_func = None, *, rate = 1):
    """Sleep given amount of seconds before calling the function"""
    def decorator_slow_down(func):
        @functools.wraps(func)
        def wrapper_slow_down(*args, **kwargs):
            time.sleep(rate)
            return func(*args, **kwargs)
        return wrapper_slow_down
    if _func is None:
        return decorator_slow_down
    else:
        return decorator_slow_down(_func)
```

در اینجا از کد پایه که در بخش قبل معرفی شد، استفاده می‌کنیم تا `@slow_down` هم با آرگومان‌ها و هم بدون آرگومان قابل فراخوانی باشد. تابع بازگشتی `countdown()` مشابه نسخه قبلی است، اما حالا دو ثانیه در هر شمارش خوابیده می‌شود:

In [38]:

```
@slow_down(rate=2)
def countdown(from_number):
    if from_number < 1:
        print("Liftoff!")
    else:
        print(from_number)
        countdown(from_number - 1)
```

همانند قبل، باید نمونه را خودتان اجرا کنید تا تأثیر دکوراتور را ببینید.

In [39]:

```
countdown(3)
```

```
3
2
1
Liftoff!
```

ایجاد Singletons

یک singleton یک کلاس با تنها یک نمونه است. در پایتون چندین singleton وجود دارد که به طور متداول از آن‌ها استفاده می‌کنید، از جمله True، None و False. این واقعیت که None یک singleton است، به شما اجازه می‌دهد تا با استفاده از عبارت is برای بررسی مساوی بودن با None استفاده کنید.

In [ ]:

```
if _func is None:
    return decorator_name
else:
    return decorator_name(_func)
```

استفاده از is فقط برای اشیاءی که دقیقاً همان نمونه هستند، True را برمی‌گرداند. دکوراتور @singleton زیر را با ذخیره کردن نمونه اول کلاس به عنوان یک ویژگی، یک کلاس را به یک singleton تبدیل می‌کند. تلاش‌های بعدی برای ایجاد نمونه به سادگی نمونه ذخیره شده را برمی‌گردانند:

In [42]:

```
import functools

def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)

    def wrapper_singleton(*args, **kwargs):
        if not wrapper_singleton.instance:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton

@singleton
class TheOne:
    pass
```

همانطور که می‌بینید، این دکوراتور کلاس، الگویی مشابه دکوراتورهای تابع ما دنبال می‌کند. تنها تفاوت این است که از cls به جای func به عنوان نام پارامتر استفاده می‌کنیم تا نشان دهیم که به عنوان یک دکوراتور کلاس قرار داده شده است.

بیایید ببینیم کار می‌کند یا نه:

In [43]:

```
first_one = TheOne()
```

In [44]:

```
another_one = TheOne()
```

In [45]:

```
id(first_one)
```

Out[45]:

```
2594632824960
```

In [46]:

```
id(another_one)
```

Out[46]:

```
2594632824960
```

In [47]:

```
first_one is another_one
```

Out[47]:

```
True
```

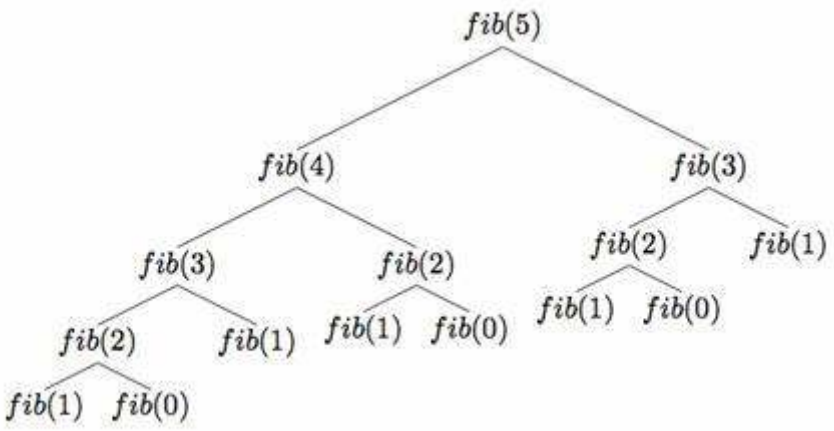
به نظر می‌رسد که first_one واقعاً همان نمونه دیگر است.

توجه: کلاس‌های Singleton در پایتون به طور معمول به اندازه سایر زبان‌ها استفاده نمی‌شوند. تأثیر یک singleton معمولاً بهتر در یک متغیر عمومی در یک ماژول پیاده‌سازی می‌شود.

کش کردن مقادیر بازگشتی

دکوراتورها می‌توانند یک مکانیزم خوب برای کش کردن (Caching) و ذخیره‌سازی حافظه باشند. به عنوان مثال، بیایید به تعریف بازگشتی دنباله فیبوناچی نگاهی بیندازیم:

1,1,2,3,5,8,13,21, ...



In [51]:

```
@count_calls
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

اگرچه پیاده‌سازی ساده است، اما عملکرد زمان اجرای آن وحشتناک است:

In [52]:

```
1 fibonacci(10)
```

```
Call 148 of 'fibonacci'
Call 149 of 'fibonacci'
Call 150 of 'fibonacci'
Call 151 of 'fibonacci'
Call 152 of 'fibonacci'
Call 153 of 'fibonacci'
Call 154 of 'fibonacci'
Call 155 of 'fibonacci'
Call 156 of 'fibonacci'
Call 157 of 'fibonacci'
Call 158 of 'fibonacci'
Call 159 of 'fibonacci'
Call 160 of 'fibonacci'
Call 161 of 'fibonacci'
Call 162 of 'fibonacci'
Call 163 of 'fibonacci'
Call 164 of 'fibonacci'
Call 165 of 'fibonacci'
Call 166 of 'fibonacci'
Call 167 of 'fibonacci'
```

In [53]:

```
1 fibonacci.num_calls
```

Out[53]:

177

برای محاسبه عدد دهم فیبوناچی، در واقع فقط باید عددهای فیبوناچی قبلی را محاسبه کنید، اما این پیاده‌سازی به یک راه‌حل عدد ۱۷۷ نیاز دارد. وضعیت بسیار بدتر می‌شود: ۲۱۸۹۱ محاسبه برای `fibonacci(20)` و تقریباً ۲.۷ میلیون محاسبه برای عدد سی‌ام لازم است. این به دلیل این است که کد عددهای فیبوناچی را که قبلاً محاسبه شده‌اند، دوباره محاسبه می‌کند.

راه حل معمول این است که از حلقه `for` و یک جدول جستجو برای پیاده‌سازی عددهای فیبوناچی استفاده کنید. با این حال، کش ساده کردن محاسبات نیز می‌تواند کار را انجام دهد:

In [54]:

```
import functools

def cache(func):
    """Keep a cache of previous function calls"""
    @functools.wraps(func)
    def wrapper_cache(*args, **kwargs):
        cache_key = args + tuple(kwargs.items())
        if cache_key not in wrapper_cache.cache:
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = dict()
    return wrapper_cache

@cache
@count_calls
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

کش به عنوان یک جدول جستجو عمل می‌کند، بنابراین اکنون `fibonacci()` فقط یکبار محاسبات لازم را انجام می‌دهد:

In [55]:

```
fibonacci(8)
```

```
Call 1 of 'fibonacci'
Call 2 of 'fibonacci'
Call 3 of 'fibonacci'
Call 4 of 'fibonacci'
Call 5 of 'fibonacci'
Call 6 of 'fibonacci'
Call 7 of 'fibonacci'
Call 8 of 'fibonacci'
Call 9 of 'fibonacci'
```

Out[55]:

```
21
```

In [56]:

```
fibonacci(10)
```

```
Call 10 of 'fibonacci'
Call 11 of 'fibonacci'
```

Out[56]:

```
55
```

In [58]:

```
fibonacci(30)
```

```
Call 22 of 'fibonacci'
Call 23 of 'fibonacci'
Call 24 of 'fibonacci'
Call 25 of 'fibonacci'
Call 26 of 'fibonacci'
Call 27 of 'fibonacci'
Call 28 of 'fibonacci'
Call 29 of 'fibonacci'
Call 30 of 'fibonacci'
Call 31 of 'fibonacci'
```

Out[58]:

```
832040
```

توجه کنید که در call نهایی با `fibonacci(8)` ، محاسبات جدیدی لازم نبودند، زیرا عدد هشتم فیبوناچی قبلاً برای `fibonacci(10)` محاسبه شده بود.

در کتابخانه استاندارد، یک کش Least Recently Used (LRU) با نام `@functools.lru_cache` در دسترس است.

این دکوراتور دارای ویژگی‌های بیشتری نسبت به دکوراتور بالا است. بجای نوشتن دکوراتور کش خودتان، باید از `@functools.lru_cache` استفاده کنید:

In [59]:

```
import functools

@functools.lru_cache(maxsize=4)
def fibonacci(num):
    print(f"Calculating fibonacci({num})")
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

پارامتر `maxsize` تعداد callهای اخیری که کش می‌شود را مشخص می‌کند. مقدار پیش‌فرض ۱۲۸ است، اما می‌توانید `maxsize=None` را مشخص کنید تا تمام call های تابع را کش کنید. با این حال، توجه کنید که اگر تعداد زیادی اشیاء بزرگ را کش کنید، این ممکن است باعث مشکلات حافظه شود.

می‌توانید از روش `.cache_info()` برای مشاهده عملکرد کش استفاده کنید و در صورت نیاز آن را تنظیم کنید. در مثال ما، از یک `maxsize` مصنوعی کوچک استفاده کردیم تا تأثیر حذف عناصر از کش را مشاهده کنیم.

In [60]:

```
1 fibonacci(10)
```

```
Calculating fibonacci(10)
Calculating fibonacci(9)
Calculating fibonacci(8)
Calculating fibonacci(7)
Calculating fibonacci(6)
Calculating fibonacci(5)
Calculating fibonacci(4)
Calculating fibonacci(3)
Calculating fibonacci(2)
Calculating fibonacci(1)
Calculating fibonacci(0)
```

Out[60]:

```
55
```

In [61]:

```
1 fibonacci(8)
```

Out[61]:

```
21
```

In [62]:

```
1 fibonacci(5)
```

```
Calculating fibonacci(5)
Calculating fibonacci(4)
Calculating fibonacci(3)
Calculating fibonacci(2)
Calculating fibonacci(1)
Calculating fibonacci(0)
```

Out[62]:

```
5
```

In [63]:

```
1 fibonacci(8)
```

```
Calculating fibonacci(8)
Calculating fibonacci(7)
Calculating fibonacci(6)
```

Out[63]:

```
21
```

In [64]:

```
1 fibonacci(5)
```

Out[64]:

```
5
```

خیلی عالی!! الان شما تقریبا بر تمام مباحث دکوراتورهای پایتون مسلط شدین و به راحتی می توانین از کتابخانه ها و فریمورک های پایتون از جمله Flask یا Django و بسیاری دیگر استفاده نمایید.