DECORATORS

CS 3030: Python

Instructor: Damià Fuentes Escoté

University of Colorado
Colorado Springs

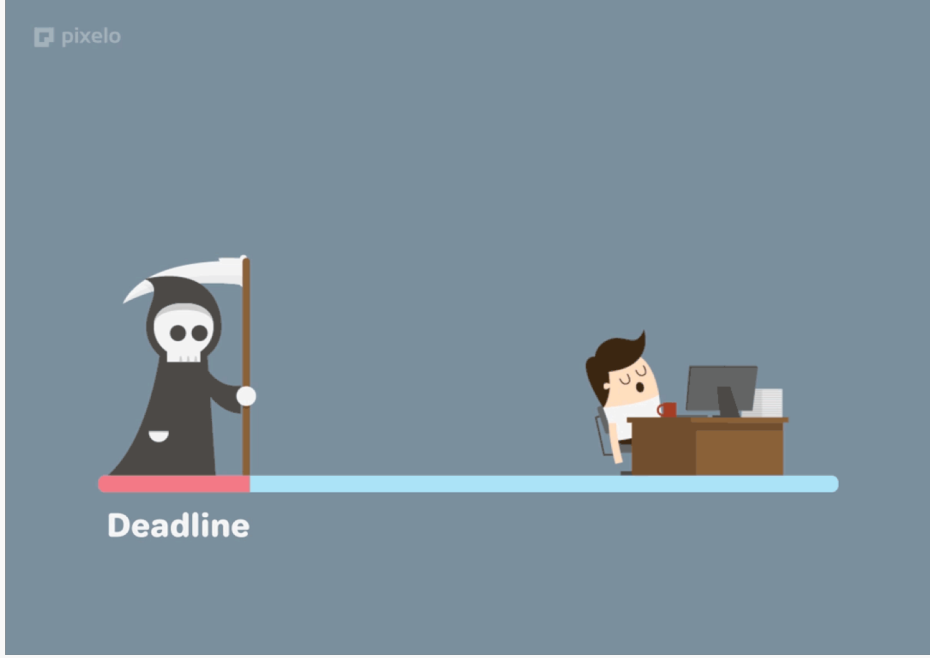
Previous lesson

- Iterators
- Iterator protocol
- Creating an iter object
- Generators
- Generators expressions

(Off topic) HackCU V

- This weekend (February 23rd and 24th) hackathon in CU Boulder
- You have 24h to develop a project
- A lot of prizes
  - *Last year there was $32k in prizes*
- Free to attend.
- They provide meals, snacks, and refreshments!

Final project



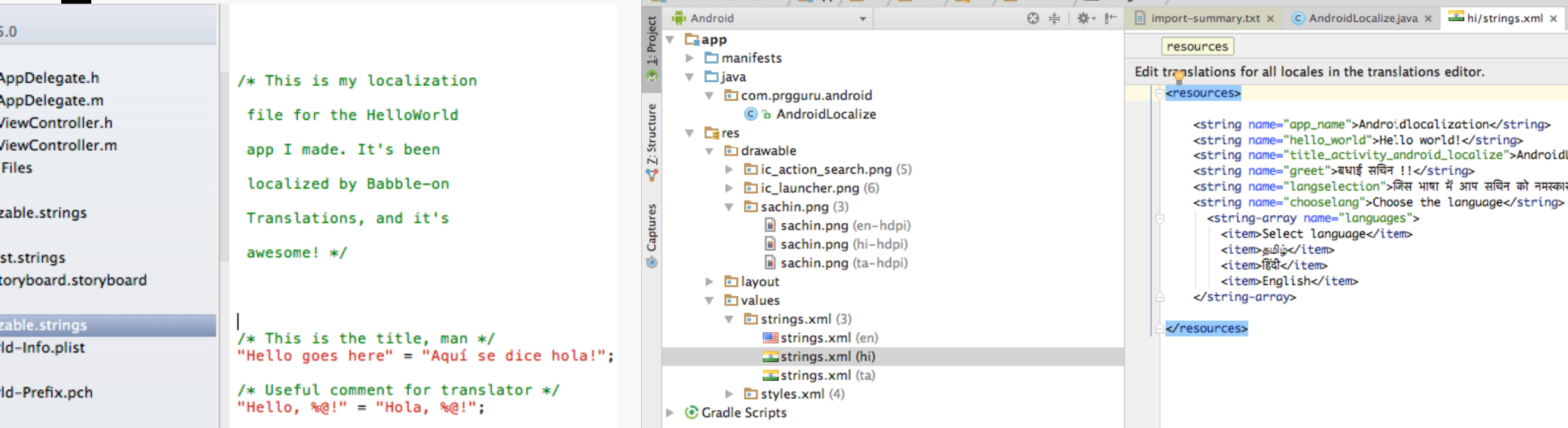
- 40% of the mark
- Develop whatever you want! But in python.
- Groups of 2 or 3. Work with Github.
- One-page project proposal by March 21st . One month left to do the proposal!
  - *You can send me the proposal before and start working once I accept it. If you do so, upload it to Canvas and send it to my email: dfuentes@uccs.edu with the subject CS 3030: Python – Project proposal or come to the office hours after class.*
  - *Maybe you could start it in the hackathon*
- Demo and presentation between May 2nd - 16th
- 5 – 10 page paper

Final Project examples

- **Make something you can use at work or school**
  - *You'll be more motivated to complete the project too, if it's going to actually be useful to you.*
  - *Anything that involves repetitive manual steps on a computer can be automated – moving files around, sending email, that sort of thing.*
- **Make something you have been dreaming to do or learn**
  - *Don't be afraid that someone can steal your ideas. If you have one, you could start it in this course!*
- **Make something to pass the course**

Final Project example 1

- Manage iOS and Android localization strings in a google drive excel sheet. Then export the google drive excel to files used for Android and iOS.
  - *Compatibility with different languages.*
  - *Some UX for those strings not translated yet.*
  - *Use Google Translator API for a first step to translating.*



Final Project example 2

- Web Scraping project to retrieve all the following information of tech companies in a specific country or state, etc.
  - *Company name*
  - *Location, address*
  - *Email*
  - *Person of contact*
- Save all this information in a database and be able to retrieve the companies for a specific city.
- Be able to make some sort of company categories?
  - *Android, iOS, Machine Learning, etc*
- Then, write an automated cool email to all the emails we want requesting for a position with attached resume.
  - *Triggered by an email, automated persistence if the company doesn't respond.*
  - *Have a google sheet tracking email sent, responded, answered, job interview time, offer done, etc*

Final Project example 3

- Get auto listings from Craigslist (for example if you are looking for a new car, or a new room) and send automated texts whenever a new post appears.
  - *Simple idea, needs more functionalities.*

	Object oriented programming	Homework 4
	Iterators and generators	
	Decorators, Lambda functions	Homework 5
	7. Pattern matching with regular expressions	
	8. Reading and writing files	Homework 6
	9. Organizing files	
	10. Debugging	Homework 7
	11. Web scraping	
	12. Working with excel, word and PDF documents	Homework 8
	14. Working with csv files and json data	
	Midterm review	
	Midterm	
	Spring break	
	Spring break	
	15. Keeping time, scheduling tasks, and launching programs	HW 9
	16. Sending email and text messages	
	Network fundamentals and socket programming (TCP/UDP)	
	Network fundamentals and socket programming (TCP/UDP)	
	Develop a simple server in Django	HW 10
	Scientific computing with *NumPy*	
	Data visualization with *matplotlib*	
	Interview exercises	
	Interview exercises	

Topics we are going to do through the rest of the course

Other topics in Python

- Implement a RESTful server with Django, Flask, etc
- Unit testing
- Graphical User Interface with tkinter
- Upload a python package to PyPi
- PyGame
- Machine Learning with KERAS, TensorFlow, etc
- etc.

Whatever you can do with Python

Groups of 2 – 3 students

- If you don’t find a partner and have an idea you could do a little presentation at the end of a lecture and try to find one.
  - *Send me an email before the lecture*

DECORATORS

Functions

- By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.
- Before you can understand decorators, you must first understand how functions work. For our purposes, a function returns a value based on the given arguments.

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

add_one(2)     # 3
```

First-Class objects

- In Python, **functions are first-class objects**. This means that functions can be passed around and used as arguments, just like any other object (string, int, float, list, and so on). Only a reference to the function is passed.

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

def sayBye(name):
    return f"Bye {name}!”

def tellBob(greeter_func):
    return greeter_func("Bob")

tellBob(sayHello)      # 'Hello Bob’
tellBob(sayBye)        # 'Bye Bob!'
```

Inner functions

- Functions inside other functions are called **inner functions**.

```
def parent():
    print("Printing from the parent() function")
    def firstChild():
        print("Printing from the firstChild() function")

    def secondChild():
        print("Printing from the secondChild() function")

    secondChild()
    firstChild()


parent()        # Printing from the parent() function
                # Printing from the secondChild() function
                # Printing from the firstChild() function
```

Returning functions from functions

- Python also allows you to use functions as return values.

```
def parent(num):
    def firstChild():
        return ”String from the firstChild() function"

    def secondChild():
        return ”String from the secondChild() function”

    if num == 1:
        return firstChild
    else:
        return secondChild

print(parent(1)())  # Printing from the firstChild() function
print(parent(2)())  # Printing from the secondChild() function
```

Returning functions from functions

- Python also allows you to use functions as return values.

Functions are just like any other object in Python!

```
def parent(num):
    def firstChild():
        return ”String from the firstChild() function"

    def secondChild():
        return ”String from the secondChild() function”

    if num == 1:
        return firstChild
    else:
        return secondChild

print(parent(1)())  # Printing from the firstChild() function
print(parent(2)())  # Printing from the secondChild() function
```

Simple decorator

```
def myDecorator(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 sayWhee():
    print("Whee!")

decoratedSayWhee = myDecorator(sayWhee)
```

Simple decorator

```python
def myDecorator(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 sayWhee():
    print("Whee!")

decoratedSayWhee = myDecorator(sayWhee)
```

```python
decoratedSayWhee()
# Something is happening before the function is called.
# Whee!
# Something is happening after the function is called.
```

Simple decorator

Decorators wrap a function, modifying its behavior.

```
def myDecorator(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 sayWhee():
    print("Whee!")

decoratedSayWhee = myDecorator(sayWhee)
```

```
decoratedSayWhee()
# Something is happening before the function is called.
# Whee!
# Something is happening after the function is called.
```

<<< Decoration happens in this line! decoratedSayWhee is a reference to the wrapper function

Second example

- Write a decorator that will only make run the decorated function between 10 pm and 7 am.

Syntactic sugar

- Python allows you to use decorators in a simpler way with the @ symbol

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

@myDecorator                    # Same as: sayWhee = myDecorator(sayWhee)
def sayWhee():
    print("Whee!")
```

Reusing decorators

- Create a new file
- Create decorator called doTwice(), which runs twice any function that it decorates.
- Try it with the sayWhee() function

Decorating Functions With Arguments

```
@doTwice
def sayHello(name):
    print(f"Hello {name}")

sayHello('Bob’)   # This will break! wrapperDoTwice() takes 0
                  # positional arguments but 1 was given
```

Decorating Functions With Arguments

Where do we need to put the function arguments?

```
def myDecorator(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 sayHello(name):
    print("Hello {}".format(name))

decoratedSayHello = myDecorator(sayHello)
decoratedSayHello(”Bob”)                    # This will break!
```

Decorating Functions With Arguments

```
def myDecorator(func):
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        func(*args, **kwargs)
        print("Something is happening after the function is called.")
    return wrapper

def sayHello(name):
    print("Hello {}".format(name))

decoratedSayHello = myDecorator(sayHello)
decoratedSayHello(”Bob”)
```

Decorating Functions With Arguments

```
def doTwice(func):
    def wrapperDoTwice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapperDoTwice
```

Returning Values From Decorated Functions

```
@doTwice
def sayHello(name):
    return f"Hello {name}”

print(sayHello('Damia'))  # None
```

Returning Values From Decorated Functions

- To fix this, you need to make sure the wrapper function returns the return value of the decorated function.

```
def doTwice(func):
    def wrapperDoTwice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapperDoTwice
```

Introspection

- **Introspection** is the ability of an object to know about its own attributes at runtime.

```
print(sayHello.__name__) # wrapper_do_twice
```

- To fix this, decorators should use the **@functools.wraps** decorator, which will preserve information about the original function

Introspection

```
import functools

def doTwice(func):
    @functools.wraps(func)
    def wrapperDoTwice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapperDoTwice


print(sayHello.__name__) # sayHello
```

Decorator boilerplate template

```
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapperDecorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapperDecorator
```

Real world examples

- A @timer decorator that will measure the time a function takes to execute and print the duration to the console.
- A @debug decorator that will print the arguments a function is called with as well as its return value every time the function is called.
- A @slowDown decorator that will sleep one second before it calls the decorated function

Nesting decorators

```
@debug
@doTwice
def sayHello(name):                    # debug(doTwice(sayHello()))
    print(f"Hello {name}")

sayHello('Damia’)          # Calling say_hello('Damia’)
                           # Hello Damia
                           # Hello Damia
                           # 'say_hello' returned None
```

Nesting decorators

```
@doTwice
@debug
def sayHello(name):                    # doTwice(debug(sayHello()))
    print(f"Hello {name}")

sayHello('Damia')   # ????
```

Nesting decorators

```
@doTwice
@debug
def sayHello(name):                    # doTwice(debug(sayHello()))
    print(f"Hello {name}")

sayHello('Damia’)   # Calling sayHello('Damia’)
                    # Hello Damia
                    # 'sayHello' returned None
                    # Calling sayHello('Damia’)
                    # Hello Damia
                    # 'sayHello' returned None
```

Decorators with arguments

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

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat
    return decoratorRepeat

@repeat(numTimes=4)
def sayHello(name):
    print(f"Hello {name}")

sayHello('Damia’)      # Hello Damia
                       # Hello Damia
                       # Hello Damia
                       # Hello Damia
```

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat
    return decoratorRepeat

@repeat(numTimes=4)
def sayHello(name):
    print(f"Hello {name}")

sayHello('Damia’)      # Hello Damia
                       # Hello Damia
                       # Hello Damia
                       # Hello Damia
```

This is no different from the earlier wrapper functions you have seen, except that it is using the **numTimes** parameter that must be supplied from the outside.

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat
    return decoratorRepeat

@repeat(numTimes=4)
def sayHello(name):
    print(f"Hello {name}")

sayHello('Damia’)      # Hello Damia
                       # Hello Damia
                       # Hello Damia
                       # Hello Damia
```

Again, decoratorRepeat() looks exactly like the decorator functions you have seen earlier

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat
    return decoratorRepeat

@repeat(numTimes=4)
def sayHello(name):
    print(f"Hello {name}")

sayHello('Damia')      # Hello Damia
                       # Hello Damia
                       # Hello Damia
                       # Hello Damia
```

The outermost function, in this case repeat(numTimes=4) returns a reference to the decorator function, in this case decoratorRepeat(func).

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat
    return decoratorRepeat
```

```
@repeat
def sayHello(name):
    print(f"Hello {name}")

sayHello('Damia')
# wrapperRepeat is never executed
```

What if we now use the decorator without arguments?

Decorators with arguments

```
def repeat(numTimes): # This is another def that handles the arguments of the decorator
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat
    return decoratorRepeat
```

What if we now use the decorator without arguments?

```
@repeat
def sayHello(name):
    print(f"Hello {name}")

sayHello('Damia')
# wrapperRepeat is never executed
```

```
# Same as:
def sayHello(name):
    print(f"Hello {name}")

sayHello = repeat(sayHello)
sayHello('Damia') # Now this is the
        # reference of wrapperRepeat
```

Both please – With and without arguments

```
def name(_func=None, *, kw1=val1, kw2=val2, ...):
    def decoratorName(func):
        ...  # Create and return a wrapper function.

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

Both please

```
def repeat(_func=None, *, numTimes=2):
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat

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

Both please

```
def repeat(_func=None, *, numTimes=2):
    def decoratorRepeat(func):
        @functools.wraps(func)
        def wrapperRepeat(*args, **kwargs):
            for _ in range(numTimes):
                value = func(*args, **kwargs)
            return value
        return wrapperRepeat

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

Both please

```
@repeat                 # Same as: sayWhee = repeat(sayWhee)
def sayWhee():
    print("Whee!")

@repeat(numTimes=3)     # Same as: sayWhee2 = repeat(num_times=3)(sayWhee2)
def sayWhee2():
    print("Whee2!")

sayWhee()                       # Whee!
sayWhee2()                      # Whee!
                                # Whee2!
                                # Whee2!
                                # Whee2!
```

.

Function attributes

- Everything in Python is an object, and almost everything has attributes and methods.
- In python, functions too are objects. So they have attributes like other objects.

```
def foo():
    pass

foo.gender ='male’
foo.name ='Bob’
print(foo.gender)          # male
print(foo.name)            # Bob
```

Stateful Decorators

- You can save the state of a function by using **function attributes**.

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

@countCalls
def passFunc():
    pass

passFunc() # Call 1 of 'passFunc’
passFunc() # Call 2 of 'passFunc’
print(passFunc.numCalls) # 2
```

Time to code – Fibonacci sequence – HW5 ex 2

The Fibonacci Sequence is the series of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The next number is found by adding up the two numbers before it. The 2 is (1+1), the 3 is (1+2), the 5 is (2+3), and so on!. An implementation of the Fibonacci function could be as follows:

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

But the runtime performance is terrible. This is because the code keeps recalculating Fibonacci numbers that are already known.

1. Create a @cache decorator that will save the calculations in a function attribute dictionary. Make the decorator work for functions with more than one argument.
2. Compare with the @countCalls decorators the difference between using @cache and not using it.