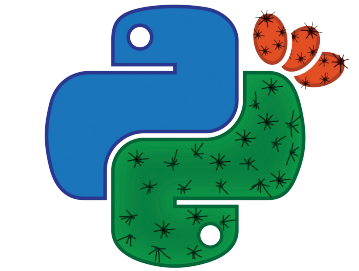
Decorators in Python

Powerful Patterns & Practical Uses



PyConIsrael

Haim Michael

blog.lifemichael.com

Few Words About Myself

My Background

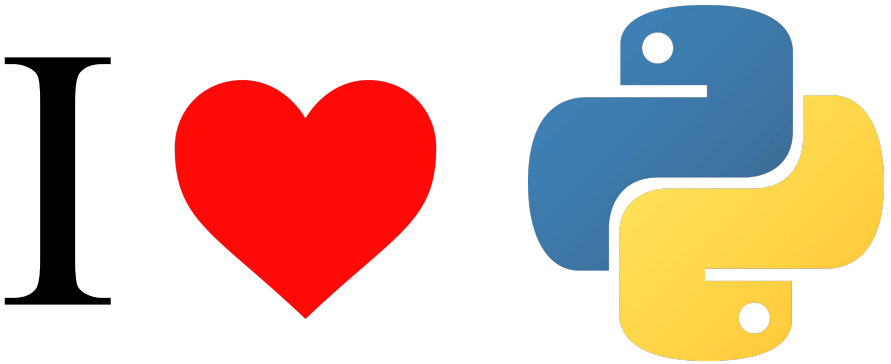
- My name is Haim Michael. I live in Tel-Aviv, and I am a father for two kids.
- I am into programming for more than 30 years. It all started with Basic (Sinclair ZX-81) and moved forward with Pascal, Prolog and Lisp.
- My passion lies with teaching and endless learning. I continuously learn and evolve.

I Love Teaching

- I focus on teaching advanced topics in software development, both in academic institutions and in high tech companies.

life michael	life michael pro	life michael udemy
www.lifemichael.com	academy.lifemichael.com/he	udemy.com/user/life-michael

I Love Programming



17+ Years

Swift (9+ Years)

PHP (15+ Years)

JS (25+ Years)

TypeScript (12+ Years)

C#

C# (24+ Years)

Scala (16+ Years)

Java (30+ Years)

Kotlin (10 Years)

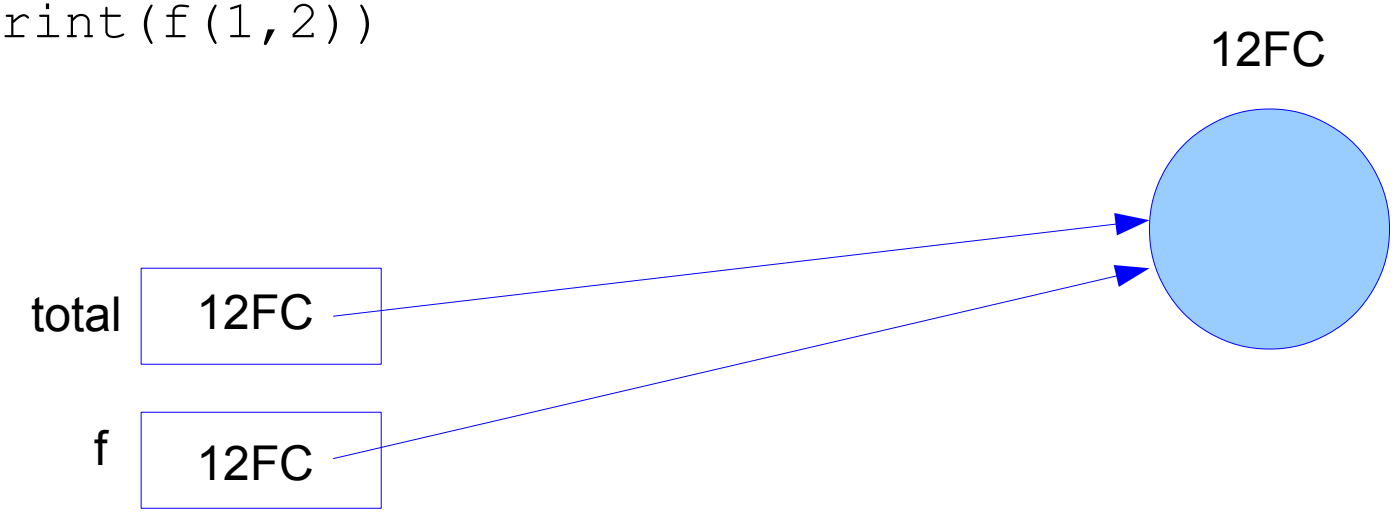
C++

C++ (30+ Years)

Functions as First Class Objects

Functions is an Object

```
def total(a:int, b:int) -> int:
    return a + b

f = total

print(f(1,2))
```



Introduction to Decorators

Introduction

- The decorators in Python allow us to add functionality to a function or a class that we define.
- Similarly to Attributes in C#, Annotations in Java/Scala/Kotlin, and Decorators in TypeScript.

Defining Data Class Sample

```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

p = Point(2,5)

print(p)
```

Developing Utilities with Typer

```
import typer

app = typer.Typer()

@app.command()
def hello(name:str, num:int) -> None:
    i = num
    while i > 0:
        print(f"Hello {name}")
        i -= 1

@app.command()
def goodbye(name:str, num:int) -> None:
    i = num
    while i > 0:
        print(f"Goodbye {name}")
        i -= 1

if __name__ == "__main__":
    app()
```

Check out Typer at
https://typer.tiangolo.com

Developing Decorators

Introduction

- When decorating a function (e.g. `foo()`), the decorator is invoked. The reference for `foo` is passed over.

```
@decorator
def foo(): pass
```

is indirectly changed into

```
foo = decorator(foo)
```

Jump Start

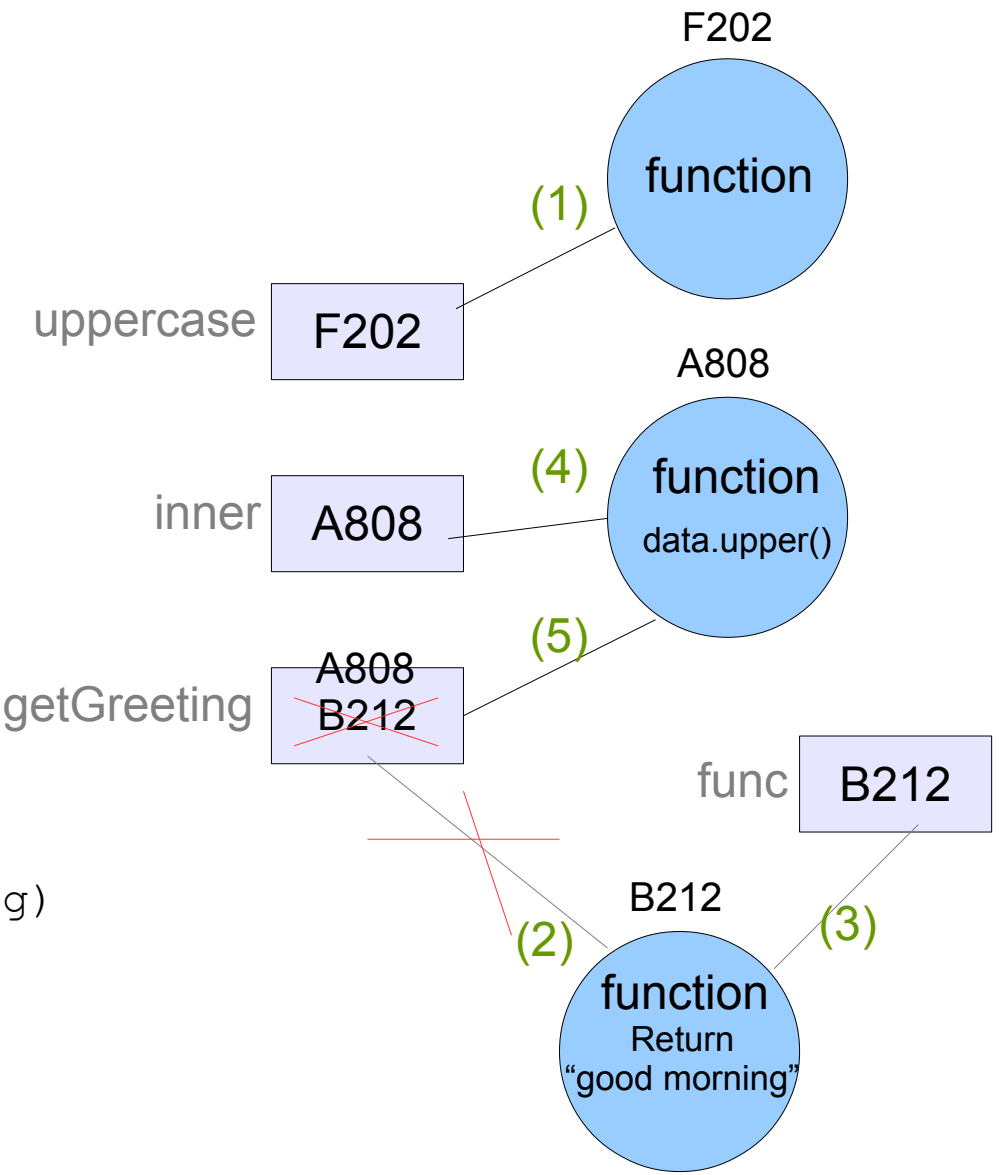
```
def uppercase(func):
    print("inside uppercase")
    def inner():
        print("inside inner")
        data = func()
        return data.upper()
    return inner

print("we love coding!")

@uppercase
def getGreeting():
    print("inside getGreeting")
    return "good morning"

#getGreeting = uppercase(getGreeting)

print("we love python!")

print(getGreeting())
```

```
we love coding!
inside uppercase
we love python!
inside inner
inside getGreeting
GOOD MORNING
```

The Output

Jump Start

```
def uppercase(func):
    print("inside uppercase")
    def inner():
        print("inside inner")
        data = func()
        return data.upper()
    return inner

print("we love coding!")

@uppercase
def getGreeting():
    print("inside getGreeting")
    return "good morning"

#getGreeting = uppercase(getGreeting)

print("we love python!")

print(getGreeting())
```



Decorating Function with Parameters

- In order to decorate the function `foo` that has parameter(s) we just need to make sure the signature of the function returned by the decorator allows delegating the call to `foo`.

Decorating Function with Parameters

using *args and **kwargs

```
def uppercase(func):
    print("inside uppercase")
    def inner(*args, **kwargs):
        print("inside inner")
        result = func(*args, **kwargs)
        return result.upper() if isinstance(result, str) else result
    return inner

print("we love coding!")

@uppercase
def get_greeting(name: str, punctuation: str = "!"):
    print("inside get_greeting")
    return f"good morning, {name}{punctuation}"

print("we love python!")

#get_greeting = uppercase(get_greeting)

print(get_greeting("Alice", punctuation="!!"))
```

```
we love coding!
inside uppercase
we love python!
inside inner
inside get_greeting
GOOD MORNING, ALICE!!
```

The Output

Using a Class as Decorator

- The decorator should be a callable object. It can be a function. I can also be a class we define together with the `__init__` and the `__call__` functions.

Using a Class as Decorator

```
class uppercase:
    def __init__(self, f):
        print("inside __init__")
        self.f = f
    def __call__(self, nickname):
        print("inside __call__")
        return self.f(nickname).upper()

print("we love coding!")

@uppercase
def greet(nickname):
    return "Good Morning " + nickname

#greet = uppercase(greet)

print("we love python!")

print(greet("Danny"))
```

```
we love coding!
inside __init__
we love python!
inside __call__
GOOD MORNING DANNY
```

The Output

Using a Class as Decorator

```
class uppercase:
    def __init__(self, f):
        print("inside __init__")
        self.f = f
    def __call__(self, *args, **kwargs):
        print("inside __call__")
        return self.f(*args, **kwargs).upper()

print("we love coding!")

@uppercase
def greet(nickname):
    return "Good Morning " + nickname

#printing with kwargs
print("we love python!")

print(greet(nickname="Danny"))
```

Might Be a Better Version

```
we love coding!
inside __init__
we love python!
inside __call__
GOOD MORNING DANNY
```

The Output

Decorators Nesting

- We can place together more than one decorator. Doing so, we should imagine the decorators execution one after the other in the order in which they are listed (top to bottom).
- The output of each decorator execution is the input for the decorator above.

Decorators Nesting

```
def stars(func):
    print("inside stars")
    def inner(text):
        print("inside inner of stars")
        input = func(text)
        return "*** " + input + " ***"
    return inner

def uppercase(func):
    print("inside uppercase")
    def inner(nickname):
        print("inside inner of uppercase")
        data = func(nickname)
        return data.upper()
    return inner

@stars
@uppercase
def getGreeting(nickname):
    return "good morning " + nickname
# getGreeting = stars(uppercase(getGreeting))

print(getGreeting("dave"))
```

```
inside uppercase
inside stars
inside inner of stars
inside inner of uppercase
*** GOOD MORNING DAVE ***
```

The Output

Decorators Nesting

```
def stars(func):
    print("inside stars")
    def inner(*args, **kwargs):
        print("inside inner of stars")
        output = func(*args, **kwargs)
        return f"*** {output} ***" if isinstance(output, str) else output
    return inner

def uppercase(func):
    print("inside uppercase")
    def inner(*args, **kwargs):
        print("inside inner of uppercase")
        result = func(*args, **kwargs)
        return result.upper() if isinstance(result, str) else result
    return inner

@stars
@uppercase
def greet(name, punctuation="!"):
    return f"Good Morning {name}{punctuation}"

print(greet("Danny", punctuation="!!"))
```

Might Be a Better Version

The Output

```
inside uppercase
inside stars
inside inner of stars
inside inner of uppercase
*** GOOD MORNING DAVE ***
```

Class Decorators

- We can develop decorators for classes. The object that represents a class is also a factory function we can decorate.

```
def bang(func):
    def inner():
        print("bang!!!")
        ob = func()
        return ob
    return inner

@bang
class Rectangle(): pass

ob1 = Rectangle()
ob2 = Rectangle()
```

```
bang!!!
bang!!!
```

The Output

Class Decorators

- We can develop decorators for classes. The object that represents a class is also a factory function we can decorate.

```
def bang(func):
    def inner(*args, **kwargs):
        print("bang!!!")
        ob = func(*args, **kwargs)
        return ob
    return inner

@bang
class Rectangle(): pass

ob1 = Rectangle()
ob2 = Rectangle()
```

Might Be a Better Version

```
bang!!!
bang!!!
```

The Output

Decorators with Arguments

- We can develop a decorator that takes arguments when called.

```
def bang(*args, **kwargs):
    def f(func):
        def inner():
            print(kwargs["text"])
            ob = func()
            return ob
        return inner
    return f

@bang(text="Picaso")
class Rectangle(): pass

#Rectangle = bang(text="Picaso")(Rectangle)

ob1 = Rectangle()
ob2 = Rectangle()
```

```
Picaso
Picaso
```

The Output

Real World Examples

Measuring Execution Time

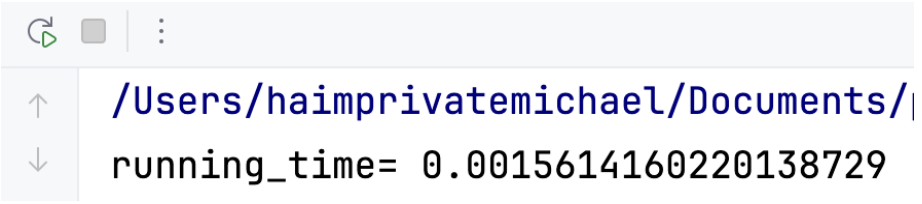
- We can easily develop a decorator for the purpose of measuring the execution time of specific functions.

```
import time

def stopper(func):
  def inner(*args, **kwargs):
    start = time.perf_counter()
    value = func(*args,**kwargs)
    end = time.perf_counter()
    run = end - start
    print("running_time=",run)
    return value
  return inner
```

Measuring Execution Time

```
@stopper
def doSomething(*args, **kwargs):
    total = 0
    for i in range(kwargs["times"]):
        total += i
    return total

doSomething(times=99999)
```

The Output



Logging

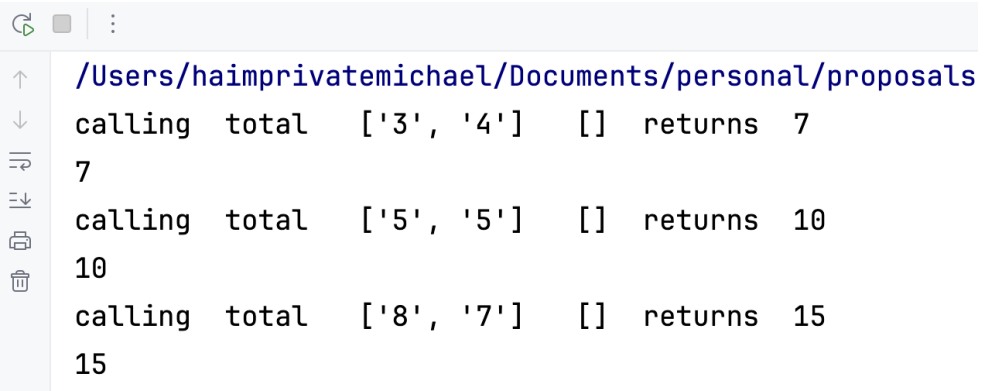
- We can easily develop a decorator that will track each and every execution of a specific function.

```
def tinylog(func):
  def f(*args, **kwargs):
    args1 = [repr(a) for a in args]
    args2 = [repr(k)+":"+repr(v) for k, v in kwargs.items()]
    value = func(*args, **kwargs)
    print("calling ", func.__name__, " ", repr(args1), " ",
            repr(args2), " returns ", value)
    return value
  return f
```

Logging

```
@tinylog
def total(a,b):
    return a+b

print(total(3,4))
print(total(5,5))
print(total(8,7))
```

The Output



RESTful Web Services

- That FastAPI framework is a great example for using decorators.

```
@app.get("/")
def read_root(message: str = Depends(get_message)):
    return {"message": message}

@app.post("/items/")
def create_item(item: dict):
    return {"item": item}
```

Caching Data

- Assuming that we have a function that performs expensive operation (e.g. fetches data from a server) we can use a decorator that will provide us with an caching mechanism.

```
from cachetools import cached, LRUCache

cache = LRUCache(maxsize=100)

@cached(cache)
def get_data(*args,**kwargs):
    # expensive computation
    return kwargs["x"] * 2

print(get_data(x=100))
```

CLI Applications Development

- Using `Typer`, a library that was developed by the same people that develop FastAPI, we can easily develop new CLI applications.

```
import typer

application = typer.Typer()

@application.command()
def shalom(name: str) -> None:
  print(f"shalom {name}")

@application.command()
def hola(name: str) -> None:
  print(f"hola {name}")

if __name__ == "__main__":
  application()
```

Best Practices

The `functools.wraps` Decorator

- Use the `functools.wraps` decorator for wrapping the inner function in order to preserve the meta of the original function.

  [Must]

```
from functools import wraps

def uppercase(func):
    @wraps(func)
    def inner():
        data = func()
        return data.upper()
    return inner

@uppercase
def getGreeting():
    return "good morning"

print(getGreeting())
print(getGreeting.__name__)
```

Arbitrary Arguments

- Consider using `*args` and `**kwargs` in your wrapper in order to ensure the decorator works with any function signature.

  [Must] in most cases

Decorator Documentation

- The value of proper documentation is well known. Don't skip the documentation when developing decorators.

  [Must]

The SRP Principle

- SRP stands for Single Responsibility Principle. It is the first of five important principles known as SOLID. Keep your decorator focused. Make sure it performs one specific task, and make sure it performs it well. Follow the spirit of the SRP principle.

  [Must]

Handle Exceptions Properly

- Make sure your decorator handles possible exceptions properly. Don’t let errors leak to the caller.

  [Must]

Don't Invent The Wheel

- Consider using other well established libraries instead of developing a new one that does the same.

  [Must]

Unit Testing

- Don't skip the testing. Maintain well written unit tests and update them when needed.

  [Must]

Type Hinting

- The explicit type annotations for function parameters and returned values, makes your code easier to understand, helps tools like linters and IDEs provide better assistance, and reduces the chance of subtle bugs. Clear type hints make long-term maintenance much simpler.

  [Must]

Tips

Learn from Others

- Explore decorators that were already developed by others. Start with the ones the standard library includes. These decorators might inspire

Step by Step

- Start with the development of relatively simple decorators and make an effort to understand how they work.

Other Programming Languages

- You can learn and get some excellent ideas by overviewing the usability of decorators in other programming languages.

Questions & Answers

Thanks for your time!

Haim.Michael@lifemichael.com
+972.54.6655837 (WhatsApp)

Python Developers

Public group · 83.7K members

https://www.facebook.com/groups/lifemichaelpython

life michael

4.6 ★★★★½ 293 ratings

Tel Aviv-Yafo, Israel

4,690 members · Public group

Organized by **Haim Michael**

https://meetup.com/lifemichael

XtremePython

https://xtremepython.dev

Use `PYCONIL2025` to Enroll for Free

Valid Till September 30th, 2025

Decorators in Python

https://tinyurl.com/pythondecorators

Online Course on Udemy

Use `PYCONIL2025` to Enroll for Free

Valid Till September 30th, 2025

Scan me!

NEWSLETTER

Python Monthly Review

The Python Monthly Review focuses on the Python programming language.

https://tinyurl.com/pythonmonthly