Lesson 7 - Closures and decorators

June 23, 2020

1 Agenda

- Variable scope
- Closure
- Decorators

2 Resources

- PEP 318 – Decorators for Functions and Methods
- Introduction to the PuDB Python Debugging Tool

3 Variable scope

[1]:
```
# %load scope1.py
def read_variables(a):
    print(a)
    print(b)

read_variables(5)
```

```
5
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-1-e09d7ccb6a2c> in <module>
  4     print(b)
  5
----> 6 read_variables(5)
```

```
<ipython-input-1-e09d7ccb6a2c> in read_variables(a)
      2 def read_variables(a):
      3     print(a)
----> 4     print(b)
      5
      6 read_variables(5)


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

[ ]:
```
# %load scope2.py
# global variable
b = 15


def read_variables(a):
    print(a)
    print(b)


read_variables(5)
```

[ ]:
```
# %load scope3.py
# global variable
b = 15


def read_variables(a):
    print(a)
    print(b)

    # local variable b
    b = 25


read_variables(5)
```

Python considers local a variable declared inside the function body.

[ ]:
```
# %load scope4.py
# global variable
b = 15


def read_variables(a):
```

```
    global b

    print(a)
    print(b)

    # global variable b
    b = 25


read_variables(5)

print('b is {}'.format(b))
```

4 Closure

A closure is a function with an extended scope that includes nonglobal variables referenced in the body of the function, but not defined there. A closure is only relevant in the context of nested functions.

[ ]:
```
# %load running_counter.py
def get_running_sum():
    numbers = []

    def compute_sum(new_number):
        print('Numbers id: {}'.format(id(numbers)))
        print('Closure data before append: {}'.format(numbers))
        numbers.append(new_number)
        print('Closure data after append: {}'.format(numbers))
        total = sum(numbers)

        print("Total: {}. Items: {}".format(
            total, len(numbers)))

        return total

    return compute_sum


get_sum = get_running_sum()

get_sum(2)
get_sum(6)
get_sum(20)
```

A free variable is a variable that is not bound to the local scope.

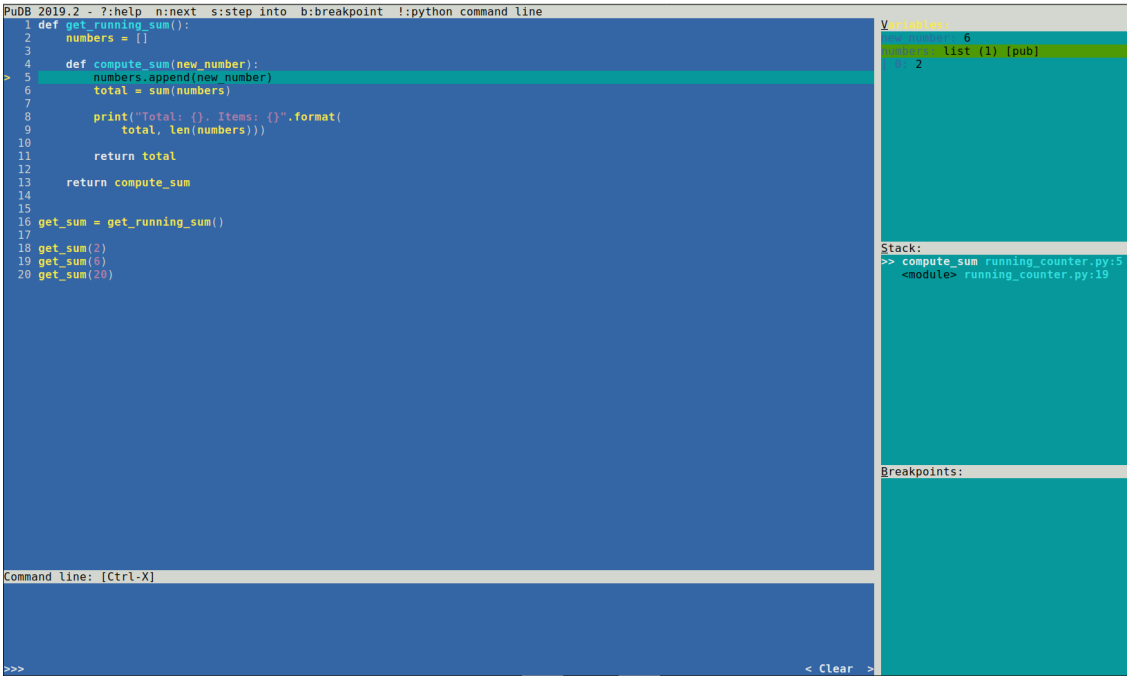
[ ]:
```
get_sum.__code__.co_varnames
```

[ ]:
```
get_sum.__code__.co_freevars
```

[ ]:
```
get_sum.__closure__
```

[ ]:
```
get_running_sum.__code__.co_cellvars
```

```
>>> pip install pudb
>>> python -m pudb running_counter.py
```



[ ]:
```
# %load running_counter2.py
def get_running_sum():
    total_sum = 0
    total_items = 0

    def compute_sum(new_number):
        total_items += 1
        total_sum += new_number

        print("Total: {}. Items: {}".format(
            total_sum, total_items))

        return total_sum

    return compute_sum
```

```
get_sum = get_running_sum()

get_sum(2)
get_sum(6)
get_sum(20)
```

[ ]:
```
# %load running_counter3.py
def get_running_sum():
    total_sum = 0
    total_items = 0

    def compute_sum(new_number):
        # we are referencing total_items before the assingment
        total_items = total_items + 1
        total_sum = total_sum + new_number

        print("Total: {}. Items: {}".format(
            total_sum, total_items))

        return total_sum

    return compute_sum


get_sum = get_running_sum()

get_sum(2)
get_sum(6)
get_sum(20)
```

[ ]:
```
# %load running_counter4.py
def get_running_sum():
    total_sum = 0
    total_items = 0

    def compute_sum(new_number):
        nonlocal total_items, total_sum

        total_items += 1
        total_sum += new_number

        print("Total: {}. Items: {}".format(
            total_sum, total_items))

        return total_sum
```

```
    return compute_sum


get_sum = get_running_sum()

get_sum(2)
get_sum(6)
get_sum(20)
```

5 Decorators

A decorator is a callable that takes a function as parameter (the decorated function) and returns it or replaces it with another function or callable object. The decorator can perform some processing on the decorated function, but it's not mandatory.

[ ]:
```
# %load bound.py
def bound_it(fnc):
    print("*" * 20)
    return fnc


def say_hello():
    print("Hello world!")


say_hello = bound_it(say_hello)

say_hello()
```

[ ]:
```
# %load bound_deco.py
def bound_it(fnc):
    print("*" * 20)
    return fnc

@bound_it
def say_hello():
    print("Hello world!")


say_hello()
```

Decorators are simply sintatic-sugar.

[ ]:
```
# %load boxit.py
def box_it(fnc):
```

```
    def inner_func():
        print("*" * 20)
        fnc()
        print("*" * 20)

    return inner_func


@box_it
def say_hello():
    print("Hello world!")


say_hello()
```

Decorators are loaded at import time

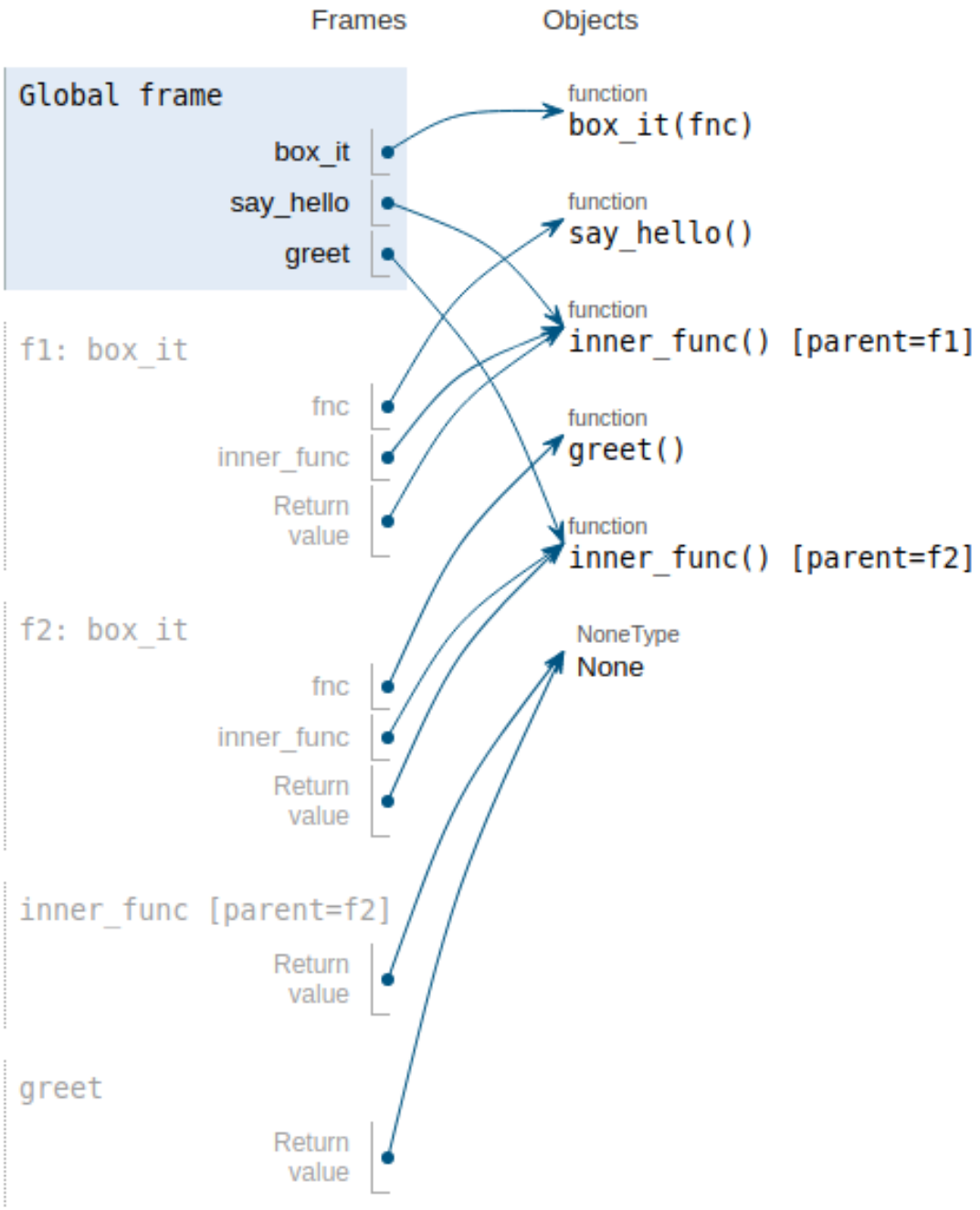
[ ]:
```
# %load box_it_loading.py
def box_it(fnc):
    print('running box_it for {}'.format(fnc))

    def inner_func():
        print("*" * 20)
        fnc()
        print("*" * 20)

    return inner_func


@box_it
def say_hello():
    print("Hello world!")


@box_it
def greet():
    print("Greetings my liege!")


say_hello()
greet()
```

[ ]:
```
# %load box_it_loading2.py
def box_it(fnc):
    print('running box_it for {}'.format(fnc))
```

```
    def inner_func():
        print("*" * 20)
        fnc()
        print("*" * 20)

    return inner_func


@box_it
def say_hello():
    print("Hello world!")


@box_it
def greet():
    print("Greetings my liege!")
```

[ ]:
```
import box_it_loading2
```

[ ]:
```
greet()
```



Python Tutor link

[ ]:
```
# %load timeme.py
import time


def timeme(fnc):
```

```
    print('Decorating {} ...'.format(fnc))

    def get_timing(*args):

        fnc_args = ', '.join(repr(arg) for arg in args)
        fnc_name = fnc.__name__

        start_timing = time.perf_counter()
        result = fnc(*args)
        fnc_run_time = time.perf_counter() - start_timing

        print("Running {} with args: {}, took {}".format(
            fnc_name, fnc_args, fnc_run_time))

        return result

    return get_timing
```

[ ]:
```
# %load using_timeme.py
import time

from timeme import timeme


@timeme
def sleep(seconds):
    print('sleep has been called with {}s: '.format(
        seconds))

    time.sleep(seconds)


@timeme
def factorial(n):
    print('factorial has been called with n: {}'.format(n))

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


sleep(3)
factorial(6)
```

[ ]:
```
# %load delay.py
```

```
def join_args(args):
    return ', '.join(repr(arg) for arg in args)


def join_kwargs(kwargs):
    kwargs_as_str = [
        "{}: {}".format(key, value)
        for key, value in kwargs.items()
    ]

    return ', '.join(kwargs_as_str)


def delayme(*args, **kwargs):
    print("Decorator got the following args: {}".format(
        join_args(args)
    ))
    print("Decorator got the following kwargs: {}".format(
        join_kwargs(kwargs)
    ))

    def wrapper(fnc):
        fnc_name = fnc.__name__

        print("Decorating function {}".format(fnc_name))

        return fnc

    return wrapper
```

[ ]:
```
# %load using_delay.py
from delay import delayme


@delayme(sec=2)
def greet(name):
    print("Greetings {}!".format(name))


def hello(name):
    print('Hello {}!'.format(name))


hello = (delayme('deco_arg', sec=3))(hello)

greet('my liege')
```

```
hello('world')
```

[ ]:
```
# %load delay2.py
import time


def join_args(args):
    return ', '.join(repr(arg) for arg in args)


def join_kwargs(kwargs):
    kwargs_as_str = [
        "{}: {}".format(key, value)
        for key, value in kwargs.items()
    ]

    return ', '.join(kwargs_as_str)


def delayme(*args, **kwargs):
    print("Decorator got the following args: {}".format(
        join_args(args)
    ))
    print("Decorator got the following kwargs: {}".format(
        join_kwargs(kwargs)
    ))
    time_to_sleep = kwargs.get('sec', 0)

    def wrapper(fnc):
        fnc_name = fnc.__name__
        print("Decorating function {}".format(fnc_name))

        def inner(*fargs, **fkwargs):
            print("Decorated function {} got the following args: {}".format(
                fnc_name, join_args(fargs)
            ))
            print("Decorated function {} got the following kwargs: {}".format(
                fnc_name, join_kwargs(fkwargs)
            ))

            if time_to_sleep:
                print("Sleeping for {}s".format(
                    time_to_sleep))
                time.sleep(time_to_sleep)
            else:
                print("Skipping sleep ...")
```

```
            result = fnc(*fargs, **fkwargs)

            return result

        return inner

    return wrapper
```

[ ]:
```
# %load using_delay2.py
from delay2 import delayme


@delayme(sec=2)
def greet(name):
    print("Greetings {}!".format(name))


def hello(name):
    print('Hello {}!'.format(name))


hello = delayme('deco_arg')(hello)

greet('my liege')
hello('world')
```

5.1 Let's practise

1. Given the following function:

```
def greet(name):
    return "Greetings {}!".format(name)
```

Create a decorator called **uppercase** that will uppercase the result

```
@uppercase
def greet(name):
    return "Greetings {}!".format(name)

print(greet("World"))
>>> "GREETINGS WORLD!"
```

2. Given the following function:

```
def divide(first_number, second_number):
    return first_number / second_number
```

Create a decorator called **safe_divide** that will output a message if the division cannot be performed, othervise it will return the result.

3. Given a set of print methods:

```
print_registry = []


def greet(name):
    return "Greetings {}!".format(name)


def say_hello(name):
    return "Hello {}!".format(name)


def say_goodbye(name):
    return "Goodbye {}!".format(name)
```

Create a decorator called **register** that will update a list called **print_registry** with all the decorated functions names.

```
print_registry = []

@register
def greet(name):
    return "Greetings {}!".format(name)


def say_hello(name):
    return "Hello {}!".format(name)

@register
def say_goodbye(name):
    return "Goodbye {}!".format(name)

print(print_registry)
>>> ['greet', 'say_goodbye']
```