SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 6

Fabian Wermelinger

Harvard University

CS107 / AC207

Tuesday, September 20th 2022

LAST TIME

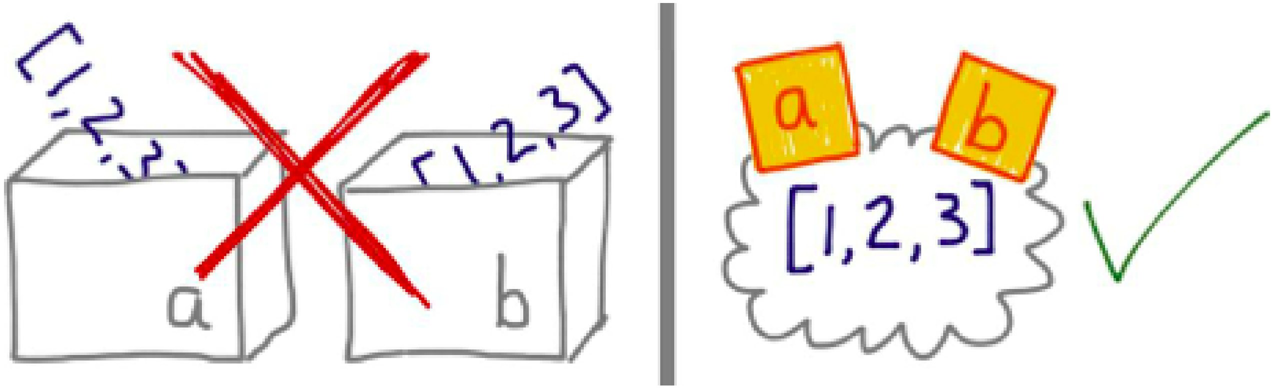
- More basic Git commands
- Repository maintenance
- Remote repositories
- Branching

TODAY

Main topics: ***Python basics (reviewed), Objects in memory and variables to reference objects, Functions, Environments, Nested environments and Closures***

Details:

- Python basics (a review)
- Nested environments
- Closures (functional programming)
- Decorators



PYTHON BASICS

It is assumed that you are familiar with the very basics of Python and its syntax. If you need a refreshment on these basics, a good reference to work through is: https://learnxinyminutes.com/docs/python/. You can further find two notebooks in the supplementary codes distributed with this lecture (see the main Git repository).

Topics include:

- Python types
- Basic data structures including lists, dictionaries and tuples.
- How to write user defined functions including variable numbers of arguments using `*args` and `**kwargs` for positional and keyword arguments.
- Writing `for`-loops and know how to use `enumerate` and `zip` in the loop header.
- Proper syntax for opening files using the `with` syntax.
- Some basic exception handling
- Know a little of `numpy` and `matplotlib`

PYTHON BASICS

`python 2.7` is no longer supported since January 1st, 2020

- If you are still using `python 2.7` please upgrade
- We are using `python 3` in this class
- A few useful notes about porting code from `python 2` to `python 3` should you ever need to do that: https://docs.python.org/3/howto/pyporting.html

PYTHON BASICS

Python Tutor:

Python tutor is tool to visualize the Python data model. We will make use of it during the following lectures to look a little bit further and understand what is going on *under the hood*. You can find it at https://pythontutor.com/. To visualize your code interactively, you can start here: https://pythontutor.com/visualize.html#mode=edit.

PYTHON BASICS: REFERENCE VARIABLES

- A "*variable*" in Python is usually called a "*name*".
- ***Example:*** the assignment `a = 1` declares the *name* `a` to hold an integer value of `1`.

```
>>> a = 1
>>> type(a)
<class 'int'>
```

- The term "*variable*" is synonymous to "*name*" and may be used interchangeably.

PYTHON BASICS: REFERENCE VARIABLES

Important take-away for today:
Variables in Python are ***references*** to objects in memory.

- If you heard this the first time now, you should make sure you remember it.
- *It is perfectly valid in Python that **multiple** names (variables) point to the same object in memory.*
- From the `python 3.9.7` Language Reference, Section 3.1:

> *Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory. The '`is`' operator compares the identity of two objects; the `id()` function returns an integer representing its identity.*

Reference: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types

PYTHON BASICS: REFERENCE VARIABLES

Let us look at the following code:

```
a = [1, 3, 5]  # list definition
b = a
print(f"a = {a} and has id {id(a)}")
print(f"b = {b} and has id {id(b)}")
print(f"Is b a? {b is a}")

# append data at the end of the list
a.append(7)
print(f"a = {a}")
print(f"b = {b}")
```

See this link for more info on the `f`-strings used here:
https://realpython.com/python-f-strings/

PYTHON BASICS: REFERENCE VARIABLES

We can investigate this further in pythontutor:

- Note that b points to the same object in memory
- The memory address of what a and b points to is *identical*

Python 3.6

```
a = [1, 3, 5]  # list definition
b = a
print(f"a = {a} and has id {id(a)}")
print(f"b = {b} and has id {id(b)}")
print(f"Is b a? {b is a}")

# append data at the end of the list
a.append(7)
print(f"a = {a}")
print(f"b = {b}")
```

[Edit this code]

line that just executed

next line to execute

< Prev Next >

Step 1 of 8

Visualized using [Python Tutor]

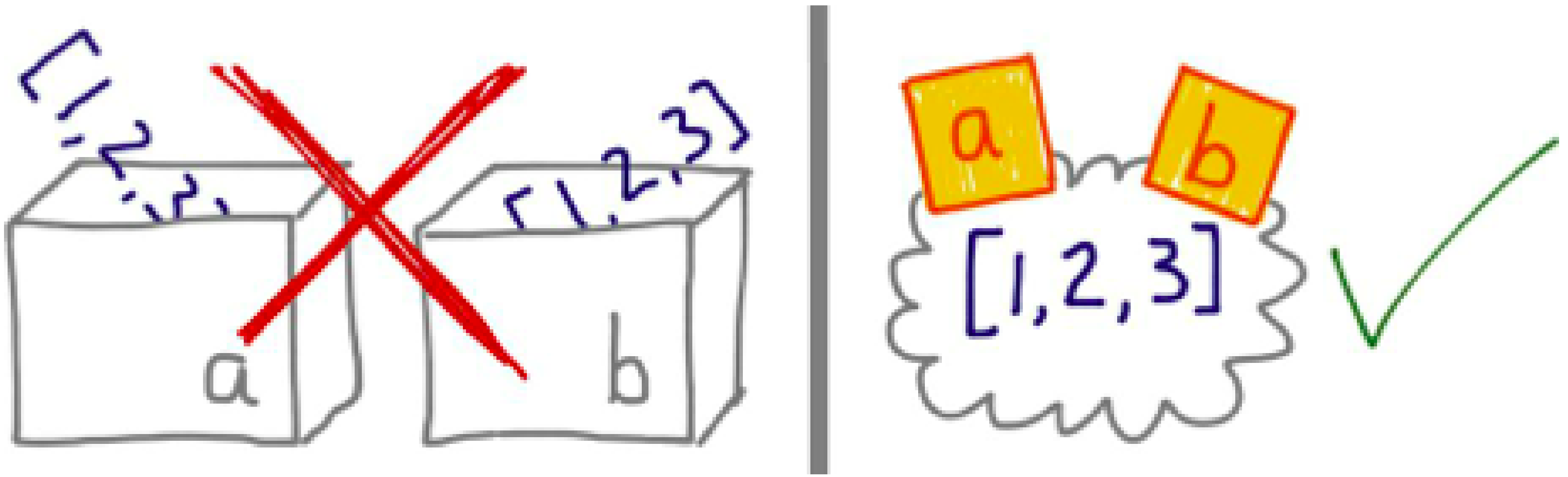
[Customize visualization]

Print output (drag lower right corner to resize)

Frames   Objects

PYTHON BASICS: REFERENCE VARIABLES

A corresponding illustration from *Fluent Python: Clear, Concise, and Effective Programming* by Luciano Ramalho (O'Reilly Media, 2015):



Can you relate what you saw on the previous slide to the right image?

PYTHON BASICS: REFERENCE VARIABLES

But be careful when working with functions. Variables inside functions may become new objects depending on the operators you use:

```python
def f(x):
    x.append(7)  # member function of object x
    return x

def g(x):
    x += [7]  # translates to an operation on object x
    return x

def h(x):
    x = x + [7]  # assign something new to x (it is now local to the function)
    return x

a = [1, 3, 5]
b = f(a)
c = g(a)
d = h(a)
```

PYTHON BASICS: REFERENCE VARIABLES

Check what is going on in pythontutor:

Python 3.6

```
def f(x):
    x.append(7)  # member function of ob
    return x

def g(x):
    x += [7]  # translates to an operati
    return x

def h(x):
    x = x + [7]  # assign something new
    return x

a = [1, 3, 5]
b = f(a)
c = g(a)
d = h(a)
```

Edit this code

line that just executed

next line to execute

< Prev Next >

Step 1 of 19

Frames Objects

PYTHON BASICS: TYPES

- Every variable in Python has a *type* (e.g. `float`, `string`, `int`, etc.)
- Python is a *dynamically typed* language:
  - Types are assigned at *runtime* rather than compile time (for example as in `C`)
  - This implies a performance penalty (slower execution) as type interpretation is not for free
  - But it makes your life (a bit) easier when writing code since you do not need to worry (too much) about it.
  - When the program starts, it is *undefined* what that variable will point to (same for *uninitialized* variables in `C`)

Python:

```python
a = 1     # interpreted as an integer (int)
a = 1.1   # a is now a float of value 1.1
          # (a different type!)
```

C:

```c
int a = 1;  // defined as an integer (int)
a = 1.1;    // a is still an integer,
            // its value will be 1
```

A name (variable) in Python can *dynamically* change its type, depending what you *assign* to it.

PYTHON BASICS: TYPES

From *Fluent Python: Clear, Concise, and Effective Programming* by Luciano Ramalho (O'Reilly Media, 2015), Chapter 11:

Static versus dynamic typing:

If type-checking is performed at *compile time*, the language is *statically typed*; if it happens at *runtime*, it is *dynamically typed*. Static typing requires type declarations (some modern languages use type inference to avoid some of that). `Fortran` and `Lisp` are the two oldest programming languages still alive. They use static and dynamic typing, respectively.

PYTHON BASICS: FRAMES

You may have noticed the two columns in the pythontutor examples we were discussing before. So far we have been talking about *objects* which are instances in memory that can have *one or more references* to it.

The evaluation of any expression requires *knowledge of the context in which the expression is being evaluated*. This context is called a ***frame***. Recall the pythontutor example from before where we *were entering a new frame every time we executed a new function*.

An ***environment*** is a *sequence of frames*, with each frame or context having a bunch of labels, or bindings, associating variables with values. The first frame in an environment is called *global* frame, which contains the bindings for imports, built-in functions, etc.

PYTHON BASICS: FRAMES

Example to study frames in Python:

```python
def f(x):
    return [i**2 for i in x]

a = [2, 3, 4]
c1 = [i**2.0 for i in a] # list comprehension is pythonic
c2 = f(a)
print(c1)
print(c2)
```

PYTHON BASICS: FRAMES

Example to study frames in Python: analyze in pythontutor

Python 3.6

```
def f(x):
    return [i**2 for i in x]

a = [2, 3, 4]
c1 = [i**2.0 for i in a] # list comprehen
c2 = f(a)
print(c1)
print(c2)
```

Edit this code

line that just executed

next line to execute

< Prev Next >

Step 1 of 21

Visualized using **Python Tutor**

Customize visualization

Print output (drag lower right corner to resize)

Frames Objects

PYTHON BASICS: FUNCTIONS AND ENVIRONMENTS

Functions are ***first class objects*** in Python. *They are treated the same as any other name.* If this term is fuzzy to you please review the supplementary notebooks in the distributed lecture codes.

Let's look at another code example:

```
s = 'The lost world...' # a string
len_of_s = len(s)       # len() is a built-in function for the length of an
                        # iterable
my_len = len            # What are we doing here? (assign a function to a name)
my_len_of_s = my_len(s)
```

- The name `my_len` is a reference to a *function object*.
- Once it is assigned a function object, the name is *bound* to that object and behaves exactly like a function → functions are *first class objects*.
- This is not true in C for example.

PYTHON BASICS: FUNCTIONS AND ENVIRONMENTS

Execution frames:

By now we know that whenever we execute a function a *new frame* is pushed onto a frame stack. The statements in the function body are executed in that new frame.

Pass by assignment:

We have also figured out that function arguments are *passed by reference* initially (the argument is a *copy of the reference* but they point to the same object in memory). After *rebinding* the name in the function, this relationship is lost. The behavior is specific to Python and the terminology used is "*pass by assignment*".

PYTHON BASICS: FUNCTIONS AND ENVIRONMENTS

There are two types of objects in Python:

1. ***mutable:*** You can mutate the state of the object. *If you **rebind** a mutable object in a function, the outer scope (outside the function) will **not** be aware of it (pass by assignment).*
2. ***immutable:*** You can not mutate such an object (they are constant) ***nor can you rebind*** it inside a function body.

PYTHON BASICS: FUNCTIONS AND ENVIRONMENTS

Example: `list` is *mutable*, `tuple` is *immutable*

```python
def mutate(x):
    x[0] = 1  # mutate first element
    return x

def rebind(x):
    x = x[:]  # rebind by assignment (overwrite old reference)
    return x

l = [4, 3, 2]  # list:  mutable object
t = (4, 3, 2)  # tuple: immutable object

# list (mutable)
l0 = mutate(l)
l1 = rebind(l)  # rebind a mutable object creates a new object

# tuple (immutable)
# t0 = mutate(t) # error: can not mutate immutable!
t1 = rebind(t)  # rebind an immutable object maintains the reference
```

PYTHON BASICS: FUNCTIONS AND ENVIRONMENTS

Example: `list` is *mutable*, `tuple` is *immutable* (pythontutor)

Python 3.6

```
def mutate(x):
    x[0] = 1  # mutate first element
    return x

def rebind(x):
    x = x[:]  # rebind by assignment (ov
    return x

l = [4, 3, 2]  # list:  mutable object
t = (4, 3, 2)  # tuple: immutable object

# list (mutable)
l0 = mutate(l)
l1 = rebind(l)  # rebind a mutable objec

# tuple (immutable)
# t0 = mutate(t) # error: can not mutate
t1 = rebind(t)  # rebind an immutable ob
```

Edit this code

line that just executed

next line to execute

< Prev   Next >

Frames   Objects

PYTHON BASICS: EXECUTION MODEL

A *code block* is executed in an *execution frame*. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block's execution has completed. *A Python program is constructed from code blocks. A block is a piece of Python code that is executed as a unit*. Code blocks are among the following:

- modules
- function bodies
- class definitions
- commands typed interactively
- a script file (what you pass to Python as an argument)
- See Python execution model for the remaining

PYTHON BASICS: NAME (VARIABLE) BINDING

Names (or variables) refer to objects. Names are introduced by *name binding operations*. The following constructs bind names:

- Formal parameters to functions
- `import` statements
- `class` and `function` definitions (these bind the class or function name in the defining block)
- *Targets that are identifiers if occurring in an assignment (what we did in the function body of `rebind(x)` before)*
- `for`-loop headers
- after the `as` keyword in a `with` statement or the `expect` clause

The `import` statement of the form "`from ... import *`" binds *all* names defined in the imported module, *except those beginning with an underscore.* This form may only be used at the module level.

PYTHON BASICS: NAME (VARIABLE) LOOKUP

A ***scope*** defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. *If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name (what we did in the `rebind(x)` function before)*.

When a name is used in a code block, it is resolved using the *nearest enclosing scope*. The set of all such scopes visible to a code block is called the block's ***environment***.

PYTHON BASICS: NAME (VARIABLE) LOOKUP

Example: nested function blocks

```python
def block0():
    a = 1  # define a variable in the block0 scope (function body)
    b = 2

    def block1():
        b = 3  # another variable not visible in block0 scope (but shadows b)

        def block2():
            print(a)  # nearest scope is block0
            print(b)  # nearest scope is block1

        block2()  # calling inside block1()

    block1()  # calling inside block0()


block0()  # calling inside global scope
```

PYTHON BASICS: NAME (VARIABLE) LOOKUP

Example: nested function blocks pythontutor

Python 3.6

```
def block0():
    a = 1  # define a variable in the bl
    b = 2

    def block1():
        b = 3  # another variable not vi

        def block2():
            print(a)  # nearest scope is
            print(b)  # nearest scope is

        block2()  # calling inside block

    block1()  # calling inside block0()


block0()  # calling inside global scope
```

Edit this code

line that just executed

next line to execute

< Prev Next >

Print output (drag lower right corner to resize)

Frames Objects

NESTED ENVIRONMENTS

- The previous example showed that you can *nest the definitions of functions* (something *not possible* in C for example).
- When you do this, the *inner* function definitions are not even evaluated until the outer function is called.
- We also found that these inner functions have automatic access to the name bindings (or variables) in the *scope* of the outer function(s). They are bound to the nearest enclosing scope (frame).

NESTED ENVIRONMENTS

Example: nested function that is partially completed

```python
def set_partial_value(partial):
    """Return a function to be called later with captured `partial` argument"""

    def set_final_value(final):
        """Combine the `partial` name from the outer scope with the `final` name"""

        return ' '.join((partial, final))

    return set_final_value  # we return a function here!

# recall: functions in python are first class objects
i_am = set_partial_value('Hi, my name is')
print(type(i_am))

print(i_am('Alice')) # you can call i_am like any other function
print(i_am('Bob'))   # you can call i_am like any other function
```

Output:

```
<class 'function'>
Hi, my name is Alice
Hi, my name is Bob
```

NESTED ENVIRONMENTS

Example: nested function that is partially completed

An explanation in words: in `line 12` we call the function `set_partial_value('Hi, my name is')` and bind its return value to the name `i_am`. The returned value is the inner function defined inside `set_partial_value` which has access to the `'Hi, my name is'` argument that we have passed to the outer function call. Because functions are first class objects in Python we can now use the name `i_am` just like any other function (note that for this "name" is more intuitive than "variable"). The call to `i_am` will finalize the `tuple` that we could only define partially when we defined the outer function `set_partial_value`.

NESTED ENVIRONMENTS

The reason this works is that in addition to the environment in which a *user-defined* function is running, that function has access to a second environment: the environment in which the function was *defined*. Here the inner function `set_final_value` has access to the environment spanned by `set_partial_value`, which is its *parent environment*.

This enables two properties:

1. Names inside the inner functions (or the outer ones for that matter) *do not interfere* with names in the global scope. Inside the inner and outer functions, *the names that are nearest to them are the ones that matter*.
2. An inner function can access the environment of its enclosing (non-local) outer function.

CLOSURES

Since the inner functions can *capture* names from an outer function's environment, the inner function is sometimes called a ***closure***. As the outer function returns another function (the closure) this programming style is called functional programming.

```python
def set_partial_value(partial):

    def set_final_value(final):
        return ' '.join((partial, final))

    return set_final_value
```

- Once `partial` is captured by the inner function, *it cannot be changed anymore.*
- This *inability to access data directly* is called ***data encapsulation*** and is one of the foundations in Object Oriented Programming (OOP), next to inheritance and polymorphism.

CLOSURES

The concept of closures in Python is useful to *augment* functions. Because functions are first class objects, passing them around as arguments to other functions and capturing them in a closure turns out to be useful. For example, you can augment a function with call information or ***wrap*** a timer around them.

CLOSURES

Example: wrap a timer around a function `f`

```python
import time

def timer(f):
    """Augment arbitrary function f with profiling information"""

    def inner(*args, **kwargs): # the closure
        t0 = time.time()
        retval = f(*args, **kwargs) # here we call the captured function
        elapsed = time.time() - t0
        print(f"{f.__name__}: elapsed time {elapsed:e} seconds")
        return retval

    return inner
```

- The function `inner` accepts a *variable* list of ***positional*** and ***keyword*** arguments.
- It *wraps* the arbitrary function `f` in between an execution timer.
- We assume an *unknown* argument list for `f`, abstracted by `*args` (positional arguments) and `**kwargs` (keyword arguments). See Section 4 in https://learnxinyminutes.com/docs/python/ to refresh these.

CLOSURES

Example: wrap a timer around a function f

```python
import time  # needed for the sleep function below

def sleep(x):  # one positional argument x
    """Sleep for x seconds"""
    time.sleep(x)

timed_sleep = timer(sleep) # augment sleep with the timer from previous slide
timed_sleep(2) # sleep for 2 seconds (timer is now wrapped around timed_sleep)
```

Output:

```
sleep: elapsed time 2.002061e+00 seconds
```

DECORATORS

Decorator = ***outer function*** + ***closure*** (that wraps code around the captured function f)

```python
def timer(f):
    def inner(*args, **kwargs):
        t0 = time.time()
        retval = f(*args, **kwargs) # here we call the captured function
        elapsed = time.time() - t0
        print(f"{f.__name__}: elapsed time {elapsed:e} seconds")
        return retval
    return inner
```

Example: (what we just did before)

```python
# function that performs useful work (with required and optional arguments)
def target(a, b, method='linear', log=True):
    pass

decorated_target = timer(target)        # decorate it
return_value = decorated_target(a, b)  # use in code (assume a and b are defined)
```

Optional reading: *Decorators* in Chapter 4 of *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison Wesley Professional, 1995.

DECORATORS

Because the decorator pattern is so useful, Python provides a special syntax for it to reduce code bloat and make code more readable.

- What we did so far (*not pythonic*):

```python
def target():
    pass

decorated_target = decorator(target)
```

- The correct way (*pythonic*):

```python
@decorator
def target():
    pass
```

DECORATORS

The `sleep` function decorated with the `timer` function correctly done:

```python
import time

@timer  # the timer decorator was defined previously (code not repeated here)
def sleep(x):
    """Sleep for x seconds"""
    time.sleep(x)

sleep(2) # sleep for 2 seconds
```

Output:

```
sleep: elapsed time 2.002102e+00 seconds
```

- Be sure you understand the decorator pattern
- It can be useful at many places in your code
- The `@timer` syntax is often called "syntactic sugar". It hides all of what we have done in the previous discussion in one line of code.
- *No need to define an intermediate function name, you can just use the decorated `sleep` function as is.*

DECORATORS

Be aware that a decorator ***is run right after*** you define the decorated function and *not at the time you call the decorated function*. Therefore, if you have decorated code in a *module*, the code of the decorating function will be executed at the time you `import` the module.

```
def decorator(f):
    print(f'{f.__name__}: start decoration')       # decorating function
    def closure(*args, **kwargs):
        print(f'running closure for {f.__name__}') # inside closure
        f(*args, **kwargs)                         # decorated function
    print(f'{f.__name__}: end decoration')         # decorating function
    return closure

@decorator
def my_func():
    print('inside function body of my_func')

print('RUNNING my_func NOW:')
my_func()
```

DECORATORS

```python
def decorator(f):
    print(f'{f.__name__}: start decoration')        # decorating function
    def closure(*args, **kwargs):
        print(f'running closure for {f.__name__}') # inside closure
        f(*args, **kwargs)                          # decorated function
    print(f'{f.__name__}: end decoration')          # decorating function
    return closure

@decorator
def my_func():
    print('inside function body of my_func')

print('RUNNING my_func NOW:')
my_func()
```

Output:

```
my_func: start decoration      # decorating function
my_func: end decoration        # decorating function
RUNNING my_func NOW:           # time when you call decorated function
running closure for my_func    # inside closure
inside function body of my_func   # actual function body of decorated function
```

DECORATORS

Step-by-step with pythontutor:

Python 3.6

```
def decorator(f):
    print(f'{f.__name__}: start decorati
    def closure(*args, **kwargs):
        print(f'running closure for {f._
        return f(*args, **kwargs)
    print(f'{f.__name__}: end decoration
    return closure

@decorator
def my_func():
    print('inside function body of my_fu

print('RUNNING my_func NOW:')
my_func()
```

Edit this code

line that just executed

next line to execute

< Prev   Next >

Print output (drag lower right corner to resize)

Frames   Objects

RECAP

- Python basics (references to objects, frames, environments, functions, https://pythontutor.com/)
- Nested environments
- Closures
- Decorators

Further reading:

- ***Recommended to read:*** Python programming FAQ
- ***Nice and compact Python refresher:*** Learn Python in X minutes
- Chapters 5, 7 and 8 in Luciano Ramalho, *"Fluent Python: Clear, Concise, and Effective Programming"*, O'Reilly Media, 2015
- Chapter 4 in E. Gamma, R. Helm, R. Johnson and J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison Wesley Professional, 1995