Functions as objects

WRITING FUNCTIONS IN PYTHON

Shayne Miel
Director of Software Engineering @ American Efficient

datacamp

Functions are just another type of object

Python objects:

```
def x():
  pass
x = [1, 2, 3]
x = {'foo': 42}
x = pandas.DataFrame()
x = 'This is a sentence.'
x = 3
x = 71.2
import x
```

2. Functions are just another type of object
The main thing you should take away from this lesson is that functions are just like any other object in Python. They are not fundamentally different from lists, dictionaries, DataFrames, strings, integers, floats, modules, or anything else in Python.

Functions as variables

```
def my_function():
  print('Hello')
x = my_function
type(x)
```

3. Functions as variables

And because functions are just another type of object, you can do anything to or with them that you would do with any other kind of object. You can take a function and assign it to a variable, like "x". Then, if you wanted to, you could call x() instead of my_function(). It doesn't have to be a function you defined, either. If you felt so inclined, you could assign the print() function to PrintyMcPrintface, and use it as your print() function.

```
<type 'function'>
```

```
x()
```

```
Hello
```

```
PrintyMcPrintface = print
PrintyMcPrintface('Python is awesome!')
```

```
Python is awesome!
```

Lists and dictionaries of functions

```
list_of_functions = [my_function, open, print]
list_of_functions[2]('I am printing with an element of a list!')
```

```
I am printing with an element of a list!
```

```
dict_of_functions = {
  'func1': my_function,
  'func2': open,
  'func3': print
}
dict_of_functions['func3']('I am printing with a value of a dict!')
```

4. Lists and dictionaries of functions

You can also add functions to a list or dictionary. Here, we've added the functions my_function(), open(), and print() to the list "list_of_functions". We can call an element of the list, and pass it arguments. Since the third element of the list is the print() function, it prints the string argument to the console. Below that, we've added the same three functions to a dictionary, under the keys "func1", "func2", and "func3". Since the print() function is stored under the key "func3", we can reference it and use it as if we were calling the function directly.

```
I am printing with a value of a dict!
```

Referencing a function

```
def my_function():
  return 42

x = my_function
my_function()
```

5. Referencing a function
Notice that when you assign a function to a variable, you do not include the parentheses after the function name. This is a subtle but very important distinction. When you type my_function() with the parentheses, you are calling that function. It evaluates to the value that the function returns. However, when you type "my_function" without the parentheses, you are referencing the function itself. It evaluates to a function object.

```
42
```

```
my_function
```

```
<function my_function at 0x7f475332a730>
```

Functions as arguments

```
def has_docstring(func):
  """Check to see if the function
  `func` has a docstring.

  Args:
    func (callable): A function.

  Returns:
    bool
  """
  return func.__doc__ is not None
```

6. Functions as arguments
The has_docstring() function checks to see whether the function that is passed to it has a docstring or not. We could define these two functions, no() and yes(), and pass them as arguments to the has_docstring() function. Since the no() function doesn't have a docstring, the has_docstring() function returns False. Likewise, has_docstring() returns True for the yes() function.

```
def no():
  return 42

def yes():
  """Return the value 42
  """
  return 42
```

```
has_docstring(no)
```

```
False
```

```
has_docstring(yes)
```

```
True
```

Defining a function inside another function

```
def foo():
  x = [3, 6, 9]

  def bar(y):
    print(y)

  for value in x:
    bar(x)
```

https://realpython.com/inner-functions-what-are-they-good-for/

8. Defining a function inside another function
A nested function can make your code easier to read. In this example, if x and y are within some bounds, foo() prints x times y. We can make that if statement easier to read by defining an in_range() function.

Defining a function inside another function

```
def foo(x, y):
  if x > 4 and x < 10 and y > 4 and y < 10:
    print(x * y)
```

```
def foo(x, y):
  def in_range(v):
    return v > 4 and v < 10

  if in_range(x) and in_range(y):
    print(x * y)
```

8. Defining a function inside another function
A nested function can make your code easier to read. In this example, if x and y are within some bounds, foo() prints x times y. We can make that if statement easier to read by defining an in_range() function.

Functions as return values

```
def get_function():
  def print_me(s):
    print(s)


  return print_me
```

9. Functions as return values
There's also nothing stopping us from returning a function. For instance, the function get_function() creates a new function, print_me(), and then returns it. If we assign the result of calling get_function() to the variable "new_func", we are assigning the return value, "print_me()" to "new_func". We can then call new_func() as if it were the print_me() function.

```
new_func = get_function()
new_func('This is a sentence.')
```

```
This is a sentence.
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Scope

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

datacamp

Names

Scope determines which variables can be accessed at different points in your code.

Names

Scope

Scope

Scope

```
x = 7
y = 200
print(x)
```

```
7
```

```
def foo():
    x = 42
    print(x)
    print(y)
```

```
foo()
```

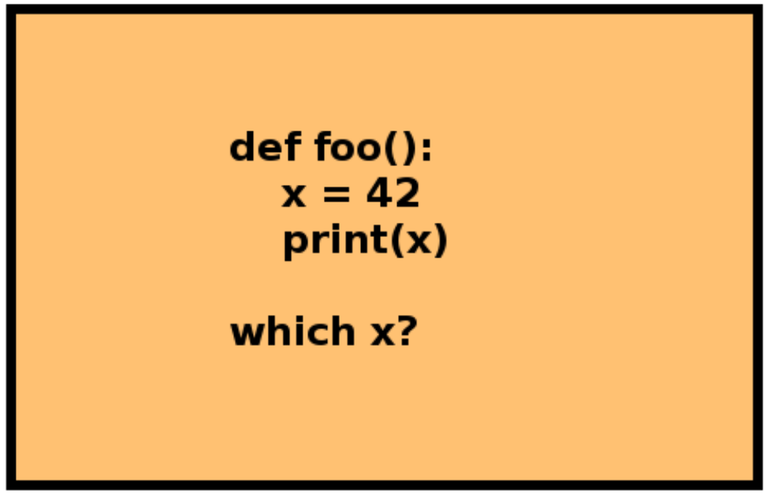
```
42
200
```

```
print(x)
```

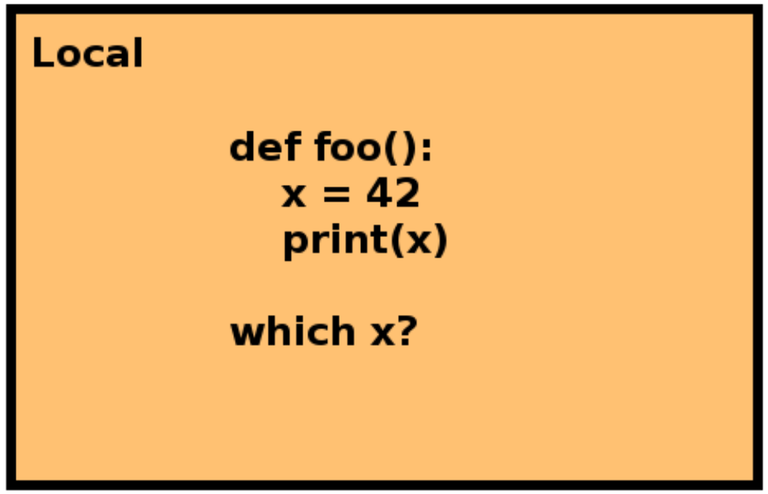
```
7
```

In foo()'s print() statement, do we mean the x that equals 42 or the x that equals 7? Python applies the same logic we applied with Tom and Janelle and assumes we mean the x that was defined right there in the function. However, there is no y defined in the function foo(), so it looks outside the function for a definition when asked to print y. Note that setting x equal to 42 inside the function foo() doesn't change the value of x that we set earlier outside of the function.

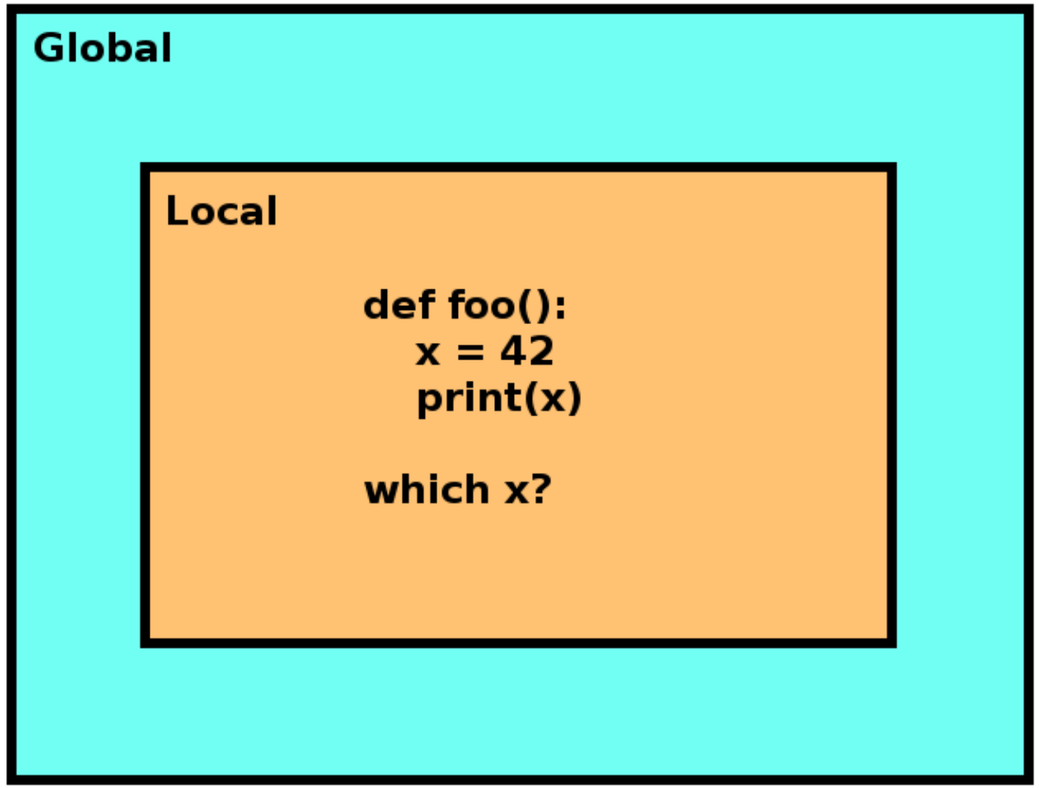
Scope



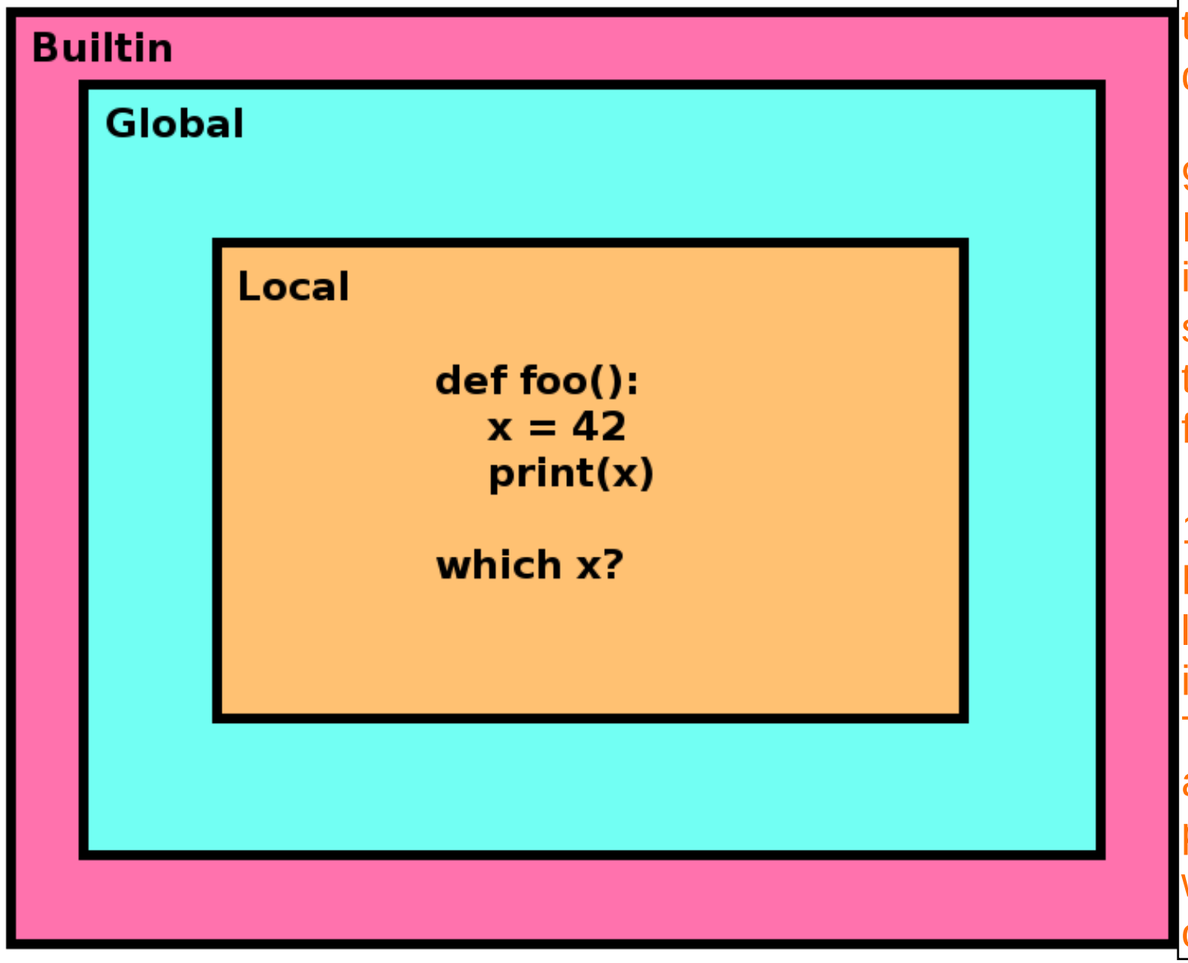
Scope



Scope



Scope

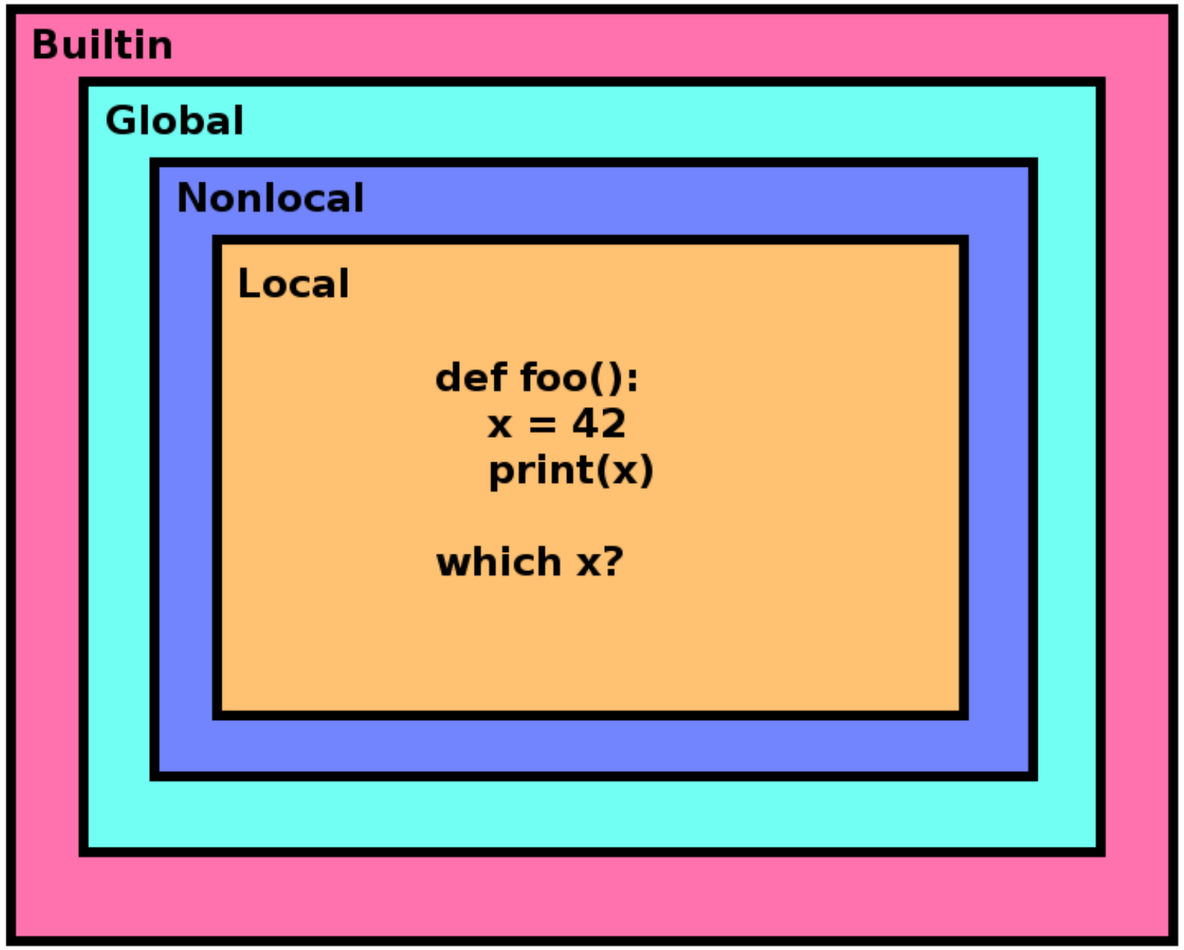


8. Scope
First, the interpreter looks in the local scope. When you are inside a function, the local scope is made up of the arguments and any variables defined inside the function.

9. Scope
If the interpreter can't find the variable in the local scope, it expands its search to the global scope. These are the things defined outside the function.

10. Scope
Finally, if it can't find the thing it is looking for in the global scope, the interpreter checks the builtin scope. These are things that are always available in Python. For instance, the print() function is in the builtin scope, which is why we are able to use it in our foo() function.

Scope



11. Scope
I actually skipped a level in that diagram. In the case of nested functions, where one function is defined inside another function, Python will check the scope of the parent function before checking the global scope. This is called the nonlocal scope to show that it is not the local scope of the child function and not the global scope.

The global keyword

```
x = 7

def foo():
  x = 42
  print(x)

foo()
```

```
42
```

```
print(x)
```

```
7
```

```
x = 7

def foo():
  global x
  x = 42
  print(x)

foo()
```

```
42
```

```
print(x)
```

```
42
```

12. The global keyword
Note that Python only gives you read access to variables defined outside of your current scope. In foo() when we set x equal to 42, Python assumed we wanted a new variable in the local scope, not the x in the global scope. If what we had really wanted was to change the value of x in the global scope, then we have to declare that we mean the global x by using the global keyword. Notice that when we print x after calling foo() now, it prints 42 instead of 7 like it used to. However, you should try to avoid using global variables like this if possible, because it can make testing and debugging harder.

The nonlocal keyword

13. The nonlocal keyword
And if we ever want to modify a variable that is defined in the nonlocal scope, we have to use the "nonlocal" keyword. It works exactly the same as the "global" keyword, but it is used when you are inside a nested function, and you want to update a variable that is defined inside your parent function.

```
def foo():
  x = 10

  def bar():
    x = 200
    print(x)

  bar()
  print(x)

foo()
```

```
200
10
```

```
def foo():
  x = 10

  def bar():
    nonlocal x
    x = 200
    print(x)

  bar()
  print(x)

foo()
```

```
200
200
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Closures

WRITING FUNCTIONS IN PYTHON

Shayne Miel

Director of Software Engineering @ American Efficient

datacamp

Attaching nonlocal variables to nested functions

```
def foo():
  a = 5
  def bar():
    print(a)
  return bar

func = foo()

func()
```

```
5
```

2. Attaching nonlocal variables to nested functions
But wait a minute, how does function "func()" know anything about variable "a"? "a" is defined in foo()'s scope, not bar()'s. You would think that "a" would not be observable outside of the scope of foo(). That's where closures come in. When foo() returned the new bar() function, Python helpfully attached any nonlocal variable that bar() was going to need to the function object. Those variables get stored in a tuple in the "__closure__" attribute of the function. The closure for "func" has one variable, and you can view the value of that variable by accessing the "cell_contents" of the item.

Closures!

```
type(func.__closure__)
```

```
<class 'tuple'>
```

```
len(func.__closure__)
```

```
1
```

```
func.__closure__[0].cell_contents
```

```
5
```

Closures and deletion

```
x = 25

def foo(value):
  def bar():
    print(value)
  return bar

my_func = foo(x)
my_func()
```

```
25
```

```
del(x)
my_func()
```

```
25
```

```
len(my_func.__closure__)
```

```
1
```

```
my_func.__closure__[0].cell_contents
```

```
25
```

3. Closures and deletion
Let's examine this bit of code. Here, x is defined in the global scope. foo() creates a function bar() that prints whatever argument was passed to foo(). When we call foo() and assign the result to "my_func", we pass in "x". So, as expected, calling my_func() prints the value of x. Now let's delete x and call my_func() again. What do you think will happen this time? If you guessed that we would still print 25, then you are correct. That's because foo()'s "value" argument gets added to the closure attached to the new "my_func" function. So even though x doesn't exist anymore, the value persists in its closure.

Closures and overwriting

```
x = 25

def foo(value):
  def bar():
    print(value)
  return bar

x = foo(x)
x()
```

```
25
```

```
len(x.__closure__)
```

```
1
```

```
x.__closure__[0].cell_contents
```

```
25
```

4. Closures and overwriting
Notice that nothing changes if we overwrite "x" instead of deleting it. Here we've passed x into foo() and then assigned the new function to the variable x. The old value of "x", 25, is still stored in the new function's closure, even though the new function is now stored in the "x" variable. This is going to be important to remember when we talk about decorators in the next lesson.

Definitions - nested function

Nested function: A function defined inside another function.

```
# outer function
def parent():
  # nested function
  def child():
    pass
  return child
```

5. Definitions - nested function
Let's go over some of the key concepts again to be sure you understand. A nested function is a function defined inside another function. We'll sometimes refer to the outer function as the parent and the nested function as the child.

Definitions - nonlocal variables

Nonlocal variables: Variables defined in the parent function that are used by the child function.

```
def parent(arg_1, arg_2):
  # From child()'s point of view,
  # `value` and `my_dict` are nonlocal variables,
  # as are `arg_1` and `arg_2`.
  value = 22
  my_dict = {'chocolate': 'yummy'}

  def child():
    print(2 * value)
    print(my_dict['chocolate'])
    print(arg_1 + arg_2)

  return child
```

6. Definitions - nonlocal variables
A nonlocal variable is any variable that gets defined in the parent function's scope, and that gets used by the child function.

7. Definitions - closure
And finally, a closure is Python's way of attaching nonlocal variables to a returned function so that the function can operate even when it is called outside of its parent's scope.

Closure: Nonlocal variables attached to a returned function.

```
def parent(arg_1, arg_2):
  value = 22
  my_dict = {'chocolate': 'yummy'}

  def child():
    print(2 * value)
    print(my_dict['chocolate'])
    print(arg_1 + arg_2)

  return child

new_function = parent(3, 4)

print([cell.cell_contents for cell in new_function.__closure__])
```

```
[3, 4, 22, {'chocolate': 'yummy'}]
```

Why does all of this matter?

Decorators use:

- Functions as objects
- Nested functions
- Nonlocal scope
- Closures

8. Why does all of this matter?
We've gone pretty deep into the internals of how Python works, and you must be wondering, "Why does all of this matter?" Well, in the next lesson we'll finally get to talk about decorators. In order to work, decorators have to make use of all of these concepts: functions as objects, nested functions, nonlocal scope, and closures. Now that you have a firm foundation to build on, understanding how decorators work should be easy.

Let's practice!

WRITING FUNCTIONS IN PYTHON

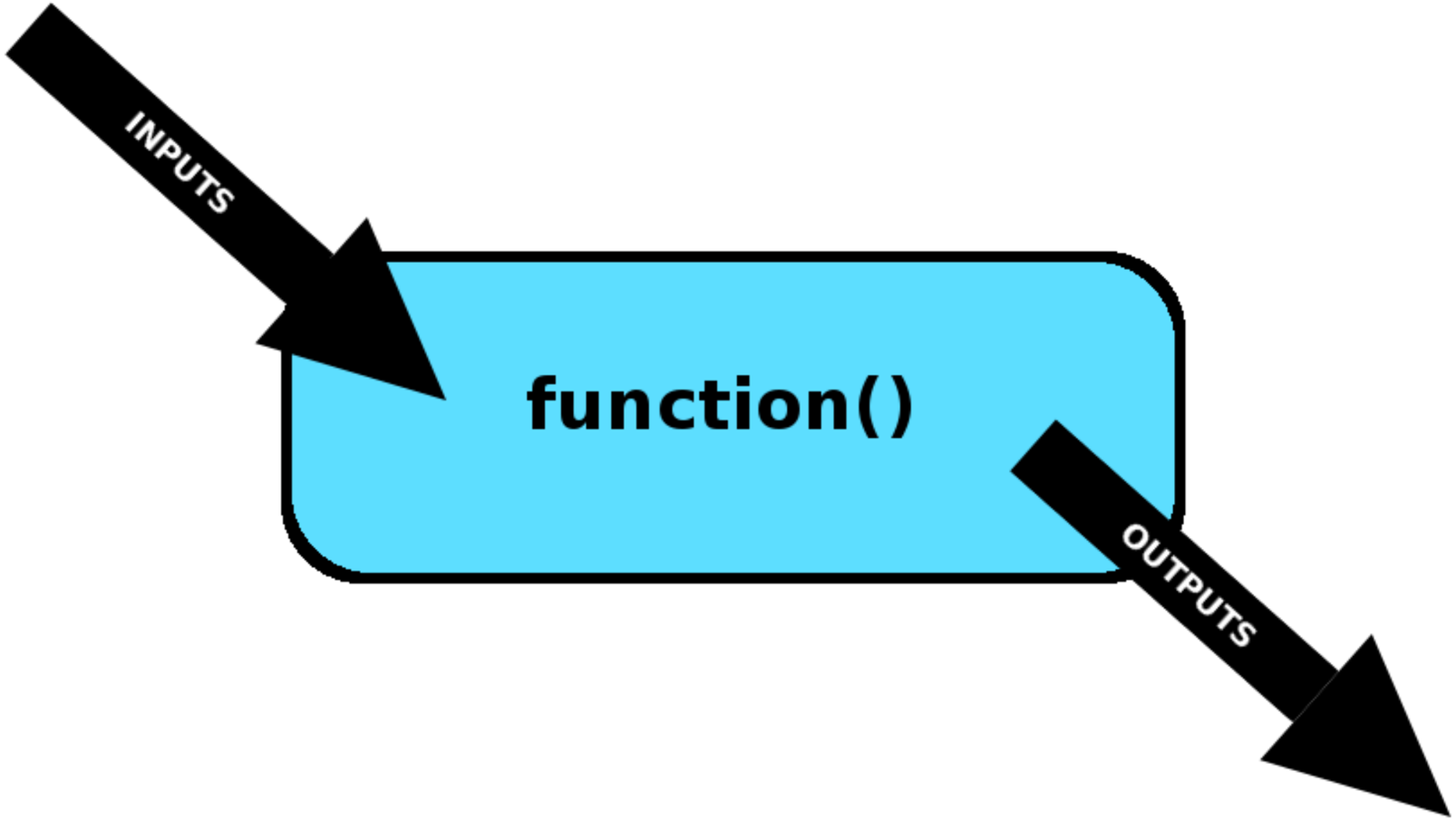
Decorators

WRITING FUNCTIONS IN PYTHON

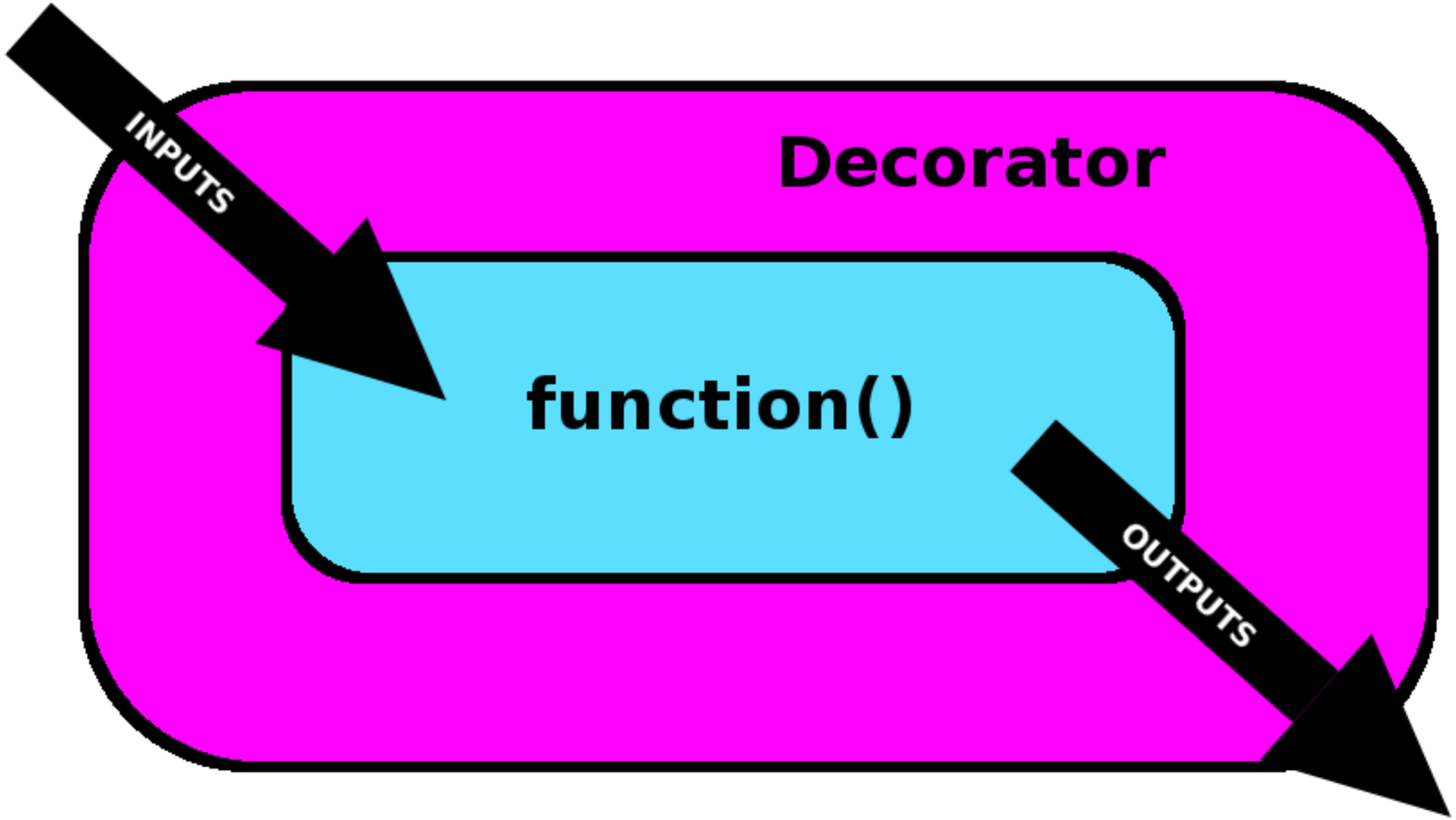
Shayne Miel
Director of Software Engineering @ American Efficient

datacamp

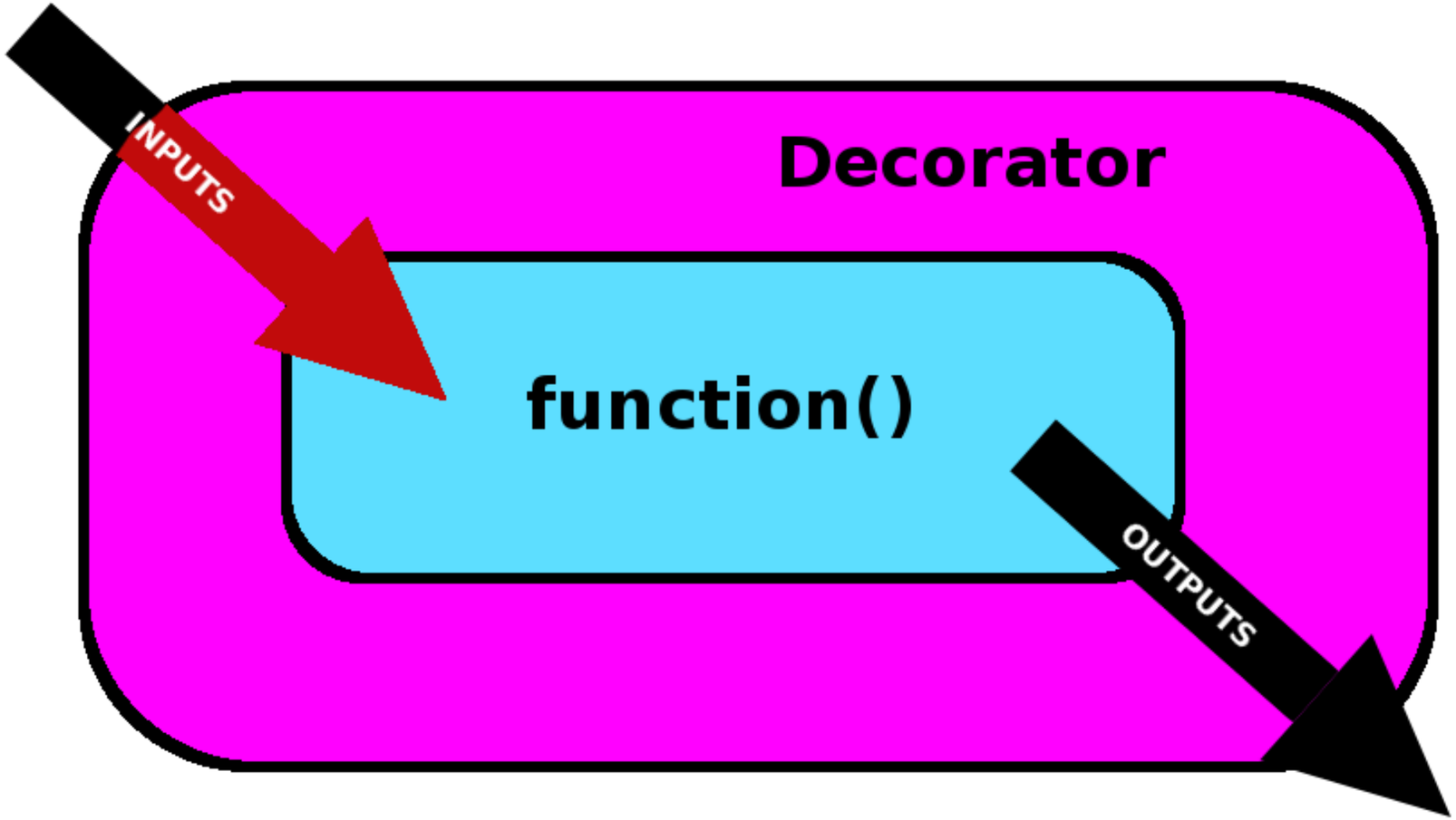
Functions



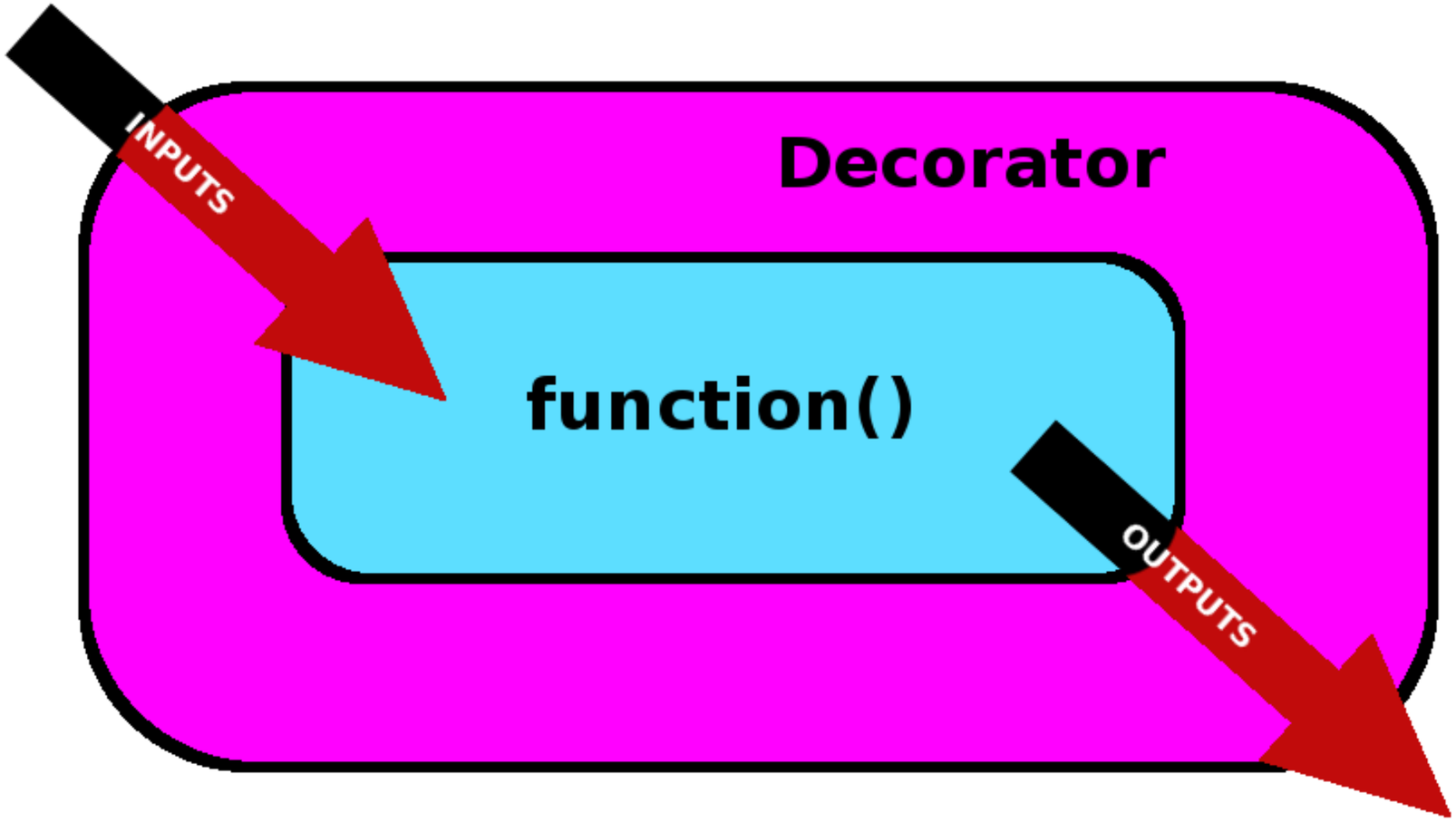
Decorators



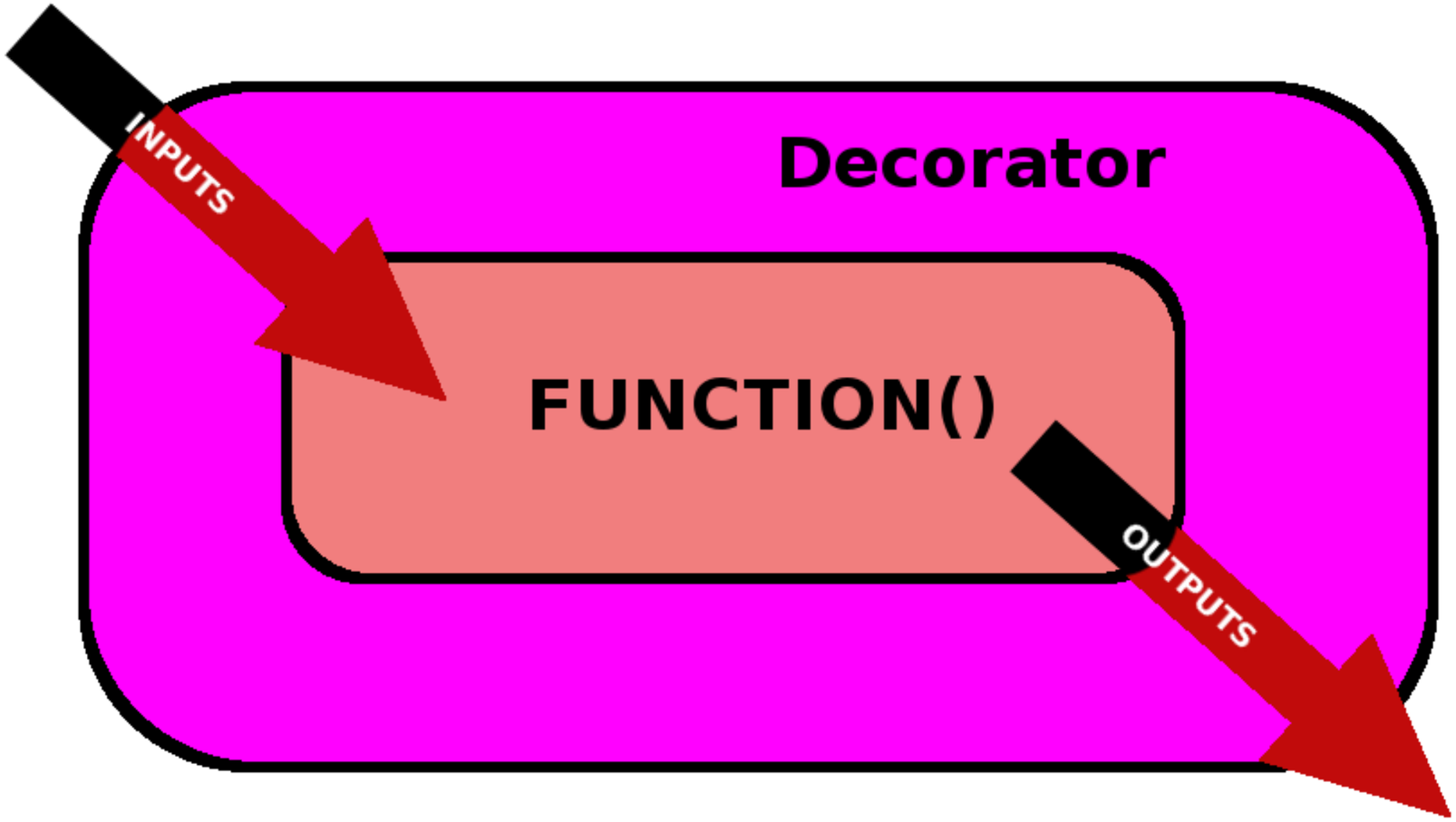
Modify inputs



Modify outputs



Modify function



What does a decorator look like?

```
@double_args
def multiply(a, b):
  return a * b
multiply(1, 5)
```

```
20
```

The double_args decorator

```
def multiply(a, b):
  return a * b
def double_args(func):
  return func
new_multiply = double_args(multiply)
new_multiply(1, 5)
```

```
5
```

```
multiply(1, 5)
```

```
5
```

The double_args decorator

```
def multiply(a, b):
  return a * b
def double_args(func):
  # Define a new function that we can modify
  def wrapper(a, b):
    # For now, just call the unmodified function
    return func(a, b)
  # Return the new function
  return wrapper
new_multiply = double_args(multiply)
new_multiply(1, 5)
```

```
5
```

The double_args decorator

```
def multiply(a, b):
  return a * b
def double_args(func):
  def wrapper(a, b):
    # Call the passed in function, but double each argument
    return func(a * 2, b * 2)
  return wrapper
new_multiply = double_args(multiply)
new_multiply(1, 5)
```

```
20
```

The double_args decorator

```
def multiply(a, b):
  return a * b
def double_args(func):
  def wrapper(a, b):
    return func(a * 2, b * 2)
  return wrapper
multiply = double_args(multiply)
multiply(1, 5)
```

```
20
```

```
multiply.__closure__[0].cell_contents
```

```
<function multiply at 0x7f0060c9e620>
```

Decorator syntax

```
def double_args(func):
  def wrapper(a, b):
    return func(a * 2, b * 2)
  return wrapper


def multiply(a, b):
  return a * b

multiply = double_args(multiply)

multiply(1, 5)
```

```
20
```

```
def double_args(func):
  def wrapper(a, b):
    return func(a * 2, b * 2)
  return wrapper


@double_args
def multiply(a, b):
  return a * b



multiply(1, 5)
```

```
20
```

Let's practice!

WRITING FUNCTIONS IN PYTHON