Functions are just another type of object

Python objects:

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

Objects in PYTHON

Everything in the python is an Object.

Functions as variables

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

```
<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!
```

using the print function indirectly to print something

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

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

Referencing a function

```
def my_function():
  return 42

x = my_function
my_function()
```

```
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
```

```
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)
```

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)
```

Functions as return values

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

  return print_me
```

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

```
This is a sentence.
```

Let's practice!

WRITING FUNCTIONS IN PYTHON

Names

Names

Scope

Scope

Scope

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

```
7
```

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

```
foo()
```

```
42
200
```

```
print(x)
```

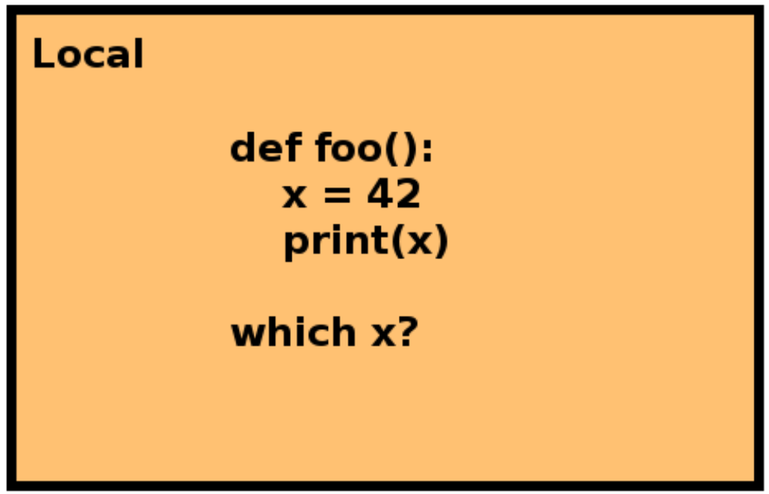
```
7
```

Scope

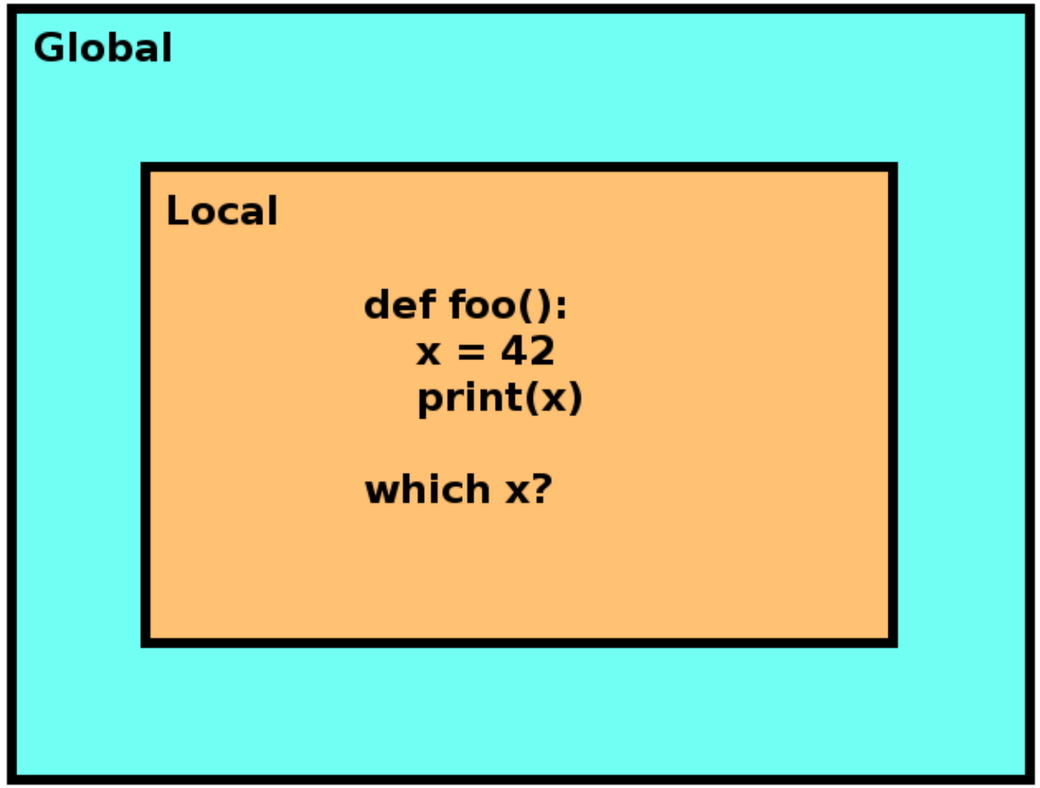
```
def foo():
    x = 42
    print(x)

which x?
```

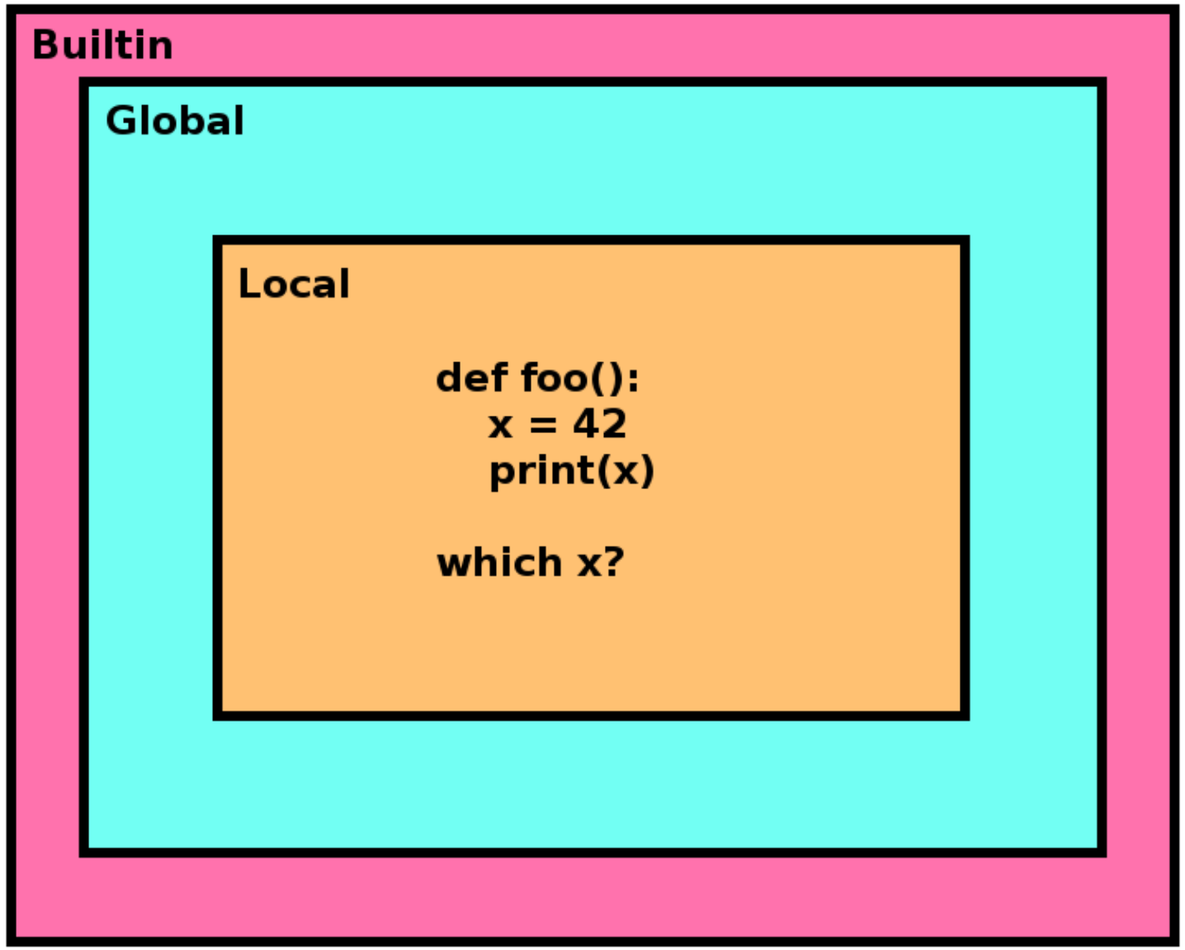
Scope



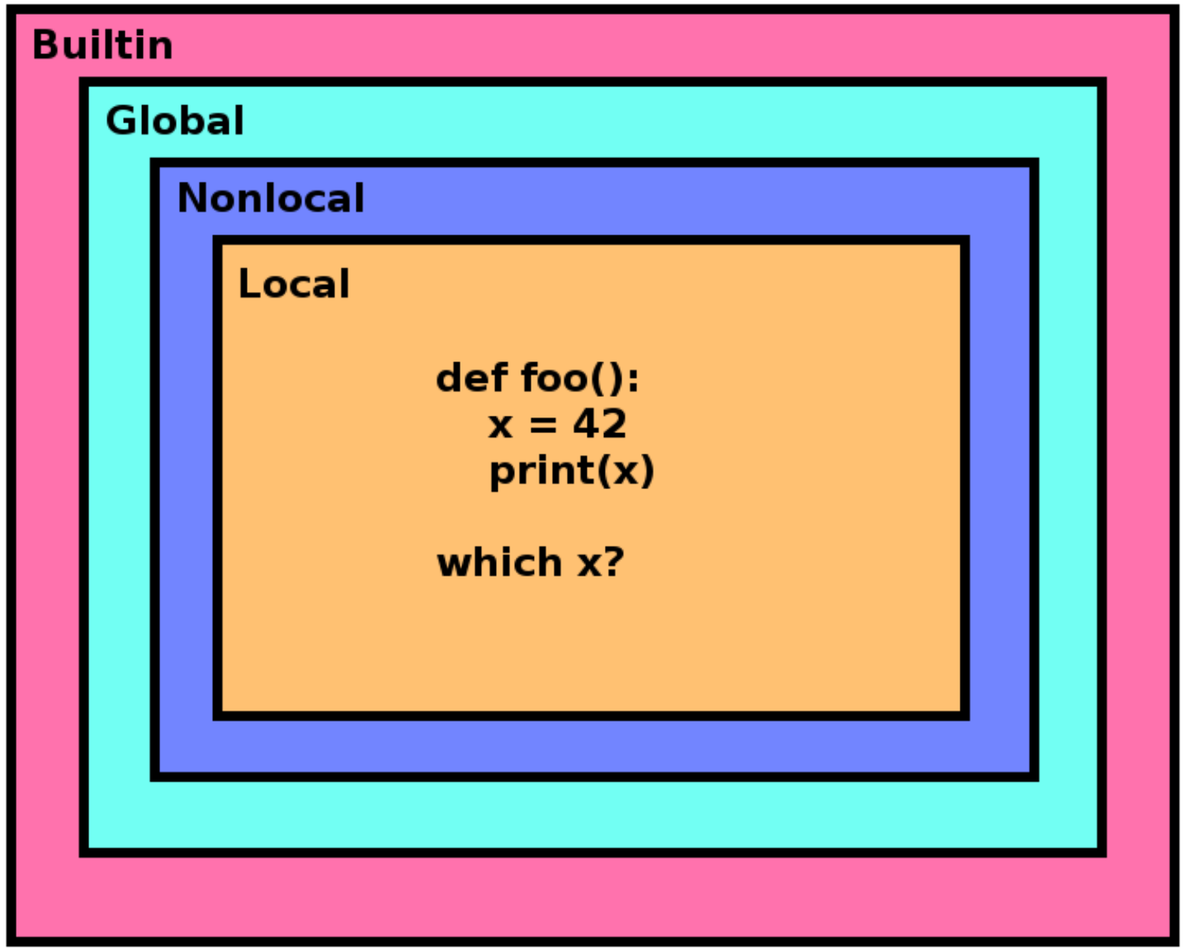
Scope



Scope



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
```

The nonlocal keyword

```
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

Attaching nonlocal variables to nested functions

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

func = foo()

func()
```

```
5
```

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
```

Closure means that an inner function always has access to the vars and parameters of its outer function, even after the outer function has returned.

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
```

Definitions - nested function

Nested function: A function defined inside another function.

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

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:

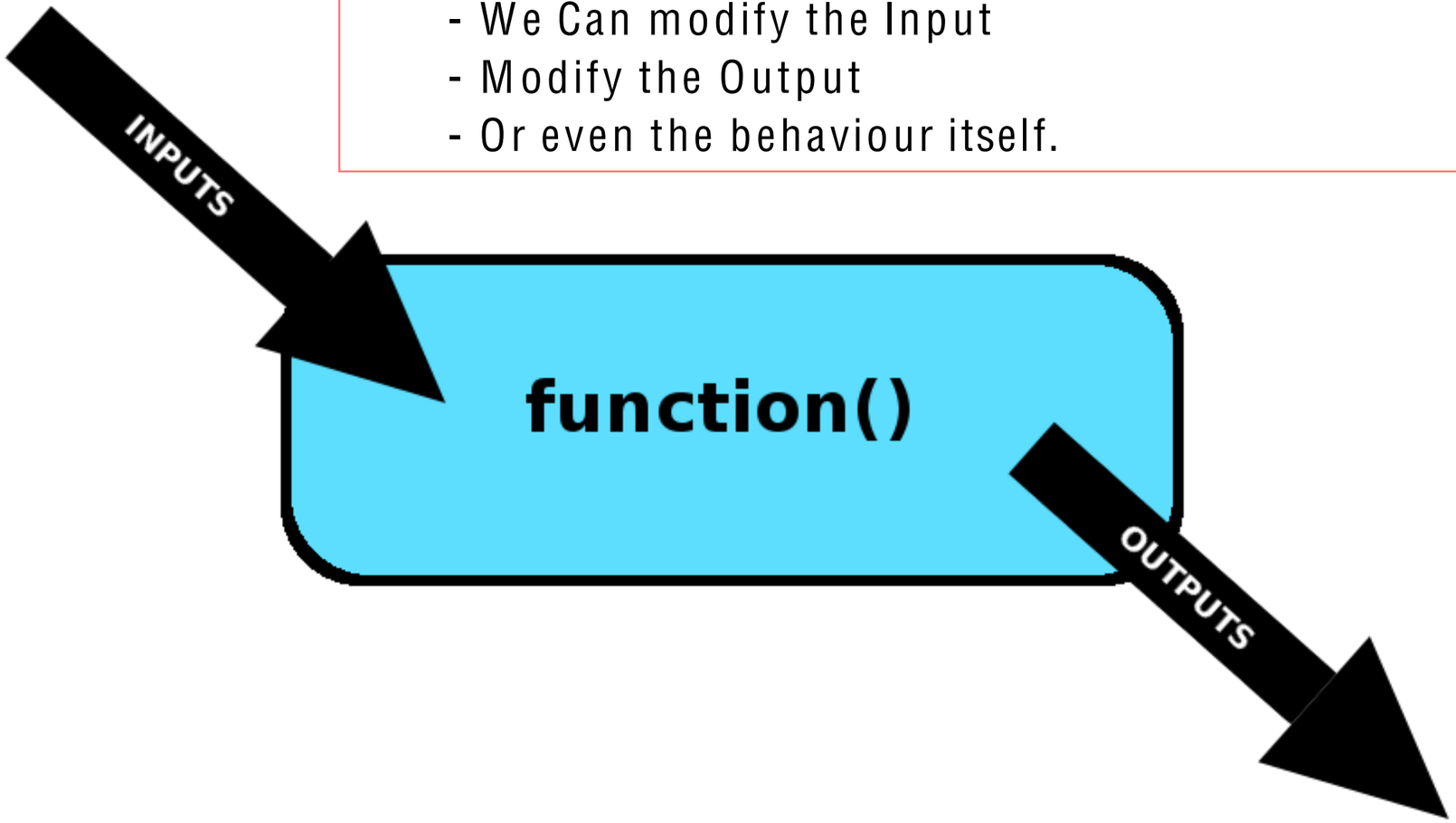
- Functions as objects
- Nested functions
- Nonlocal scope
- Closures

Let's practice!

WRITING FUNCTIONS IN PYTHON

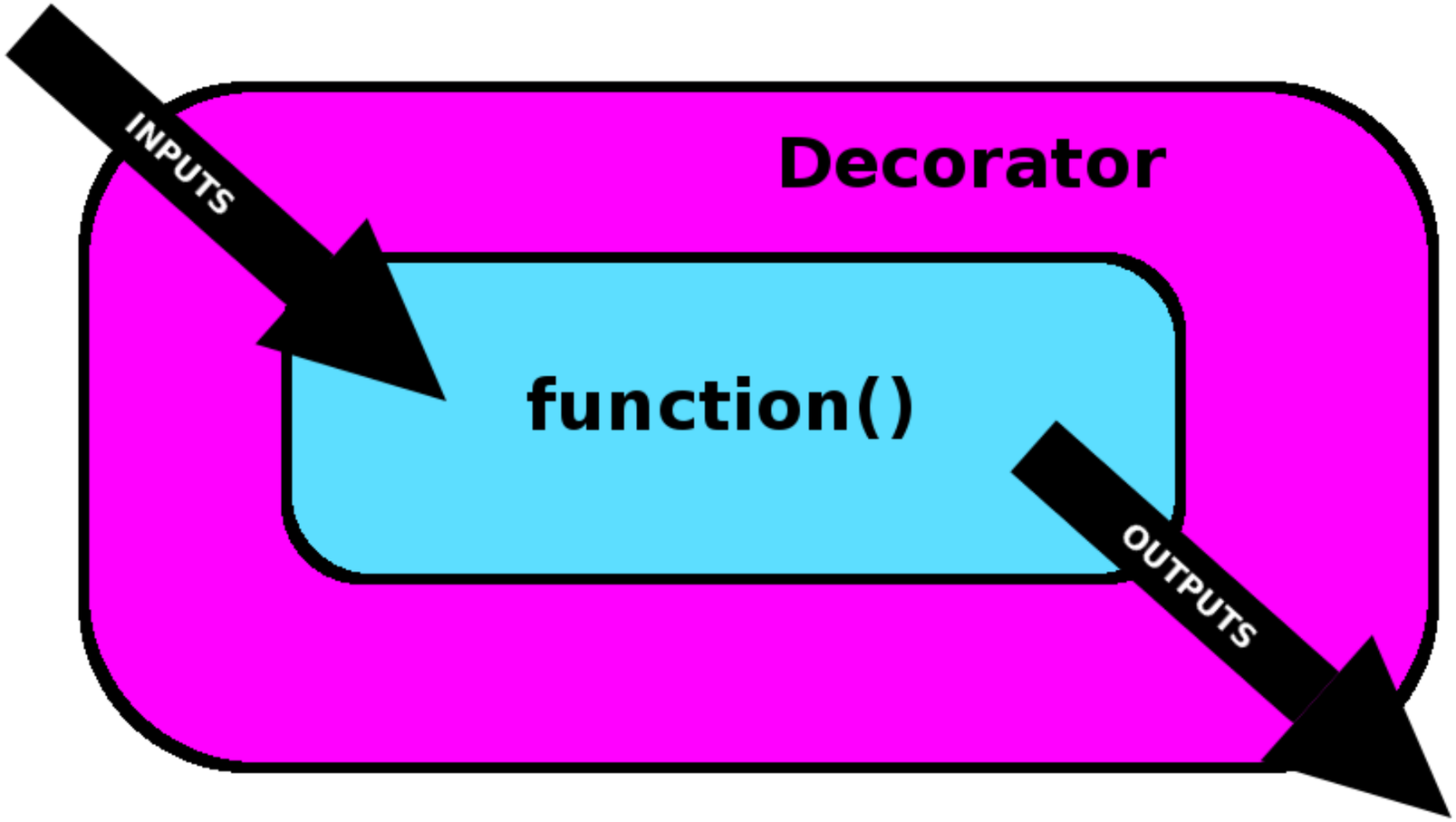
Functions

- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure
- Decorator changes the function behaviour
    - We Can modify the Input
    - Modify the Output
    - Or even the behaviour itself.

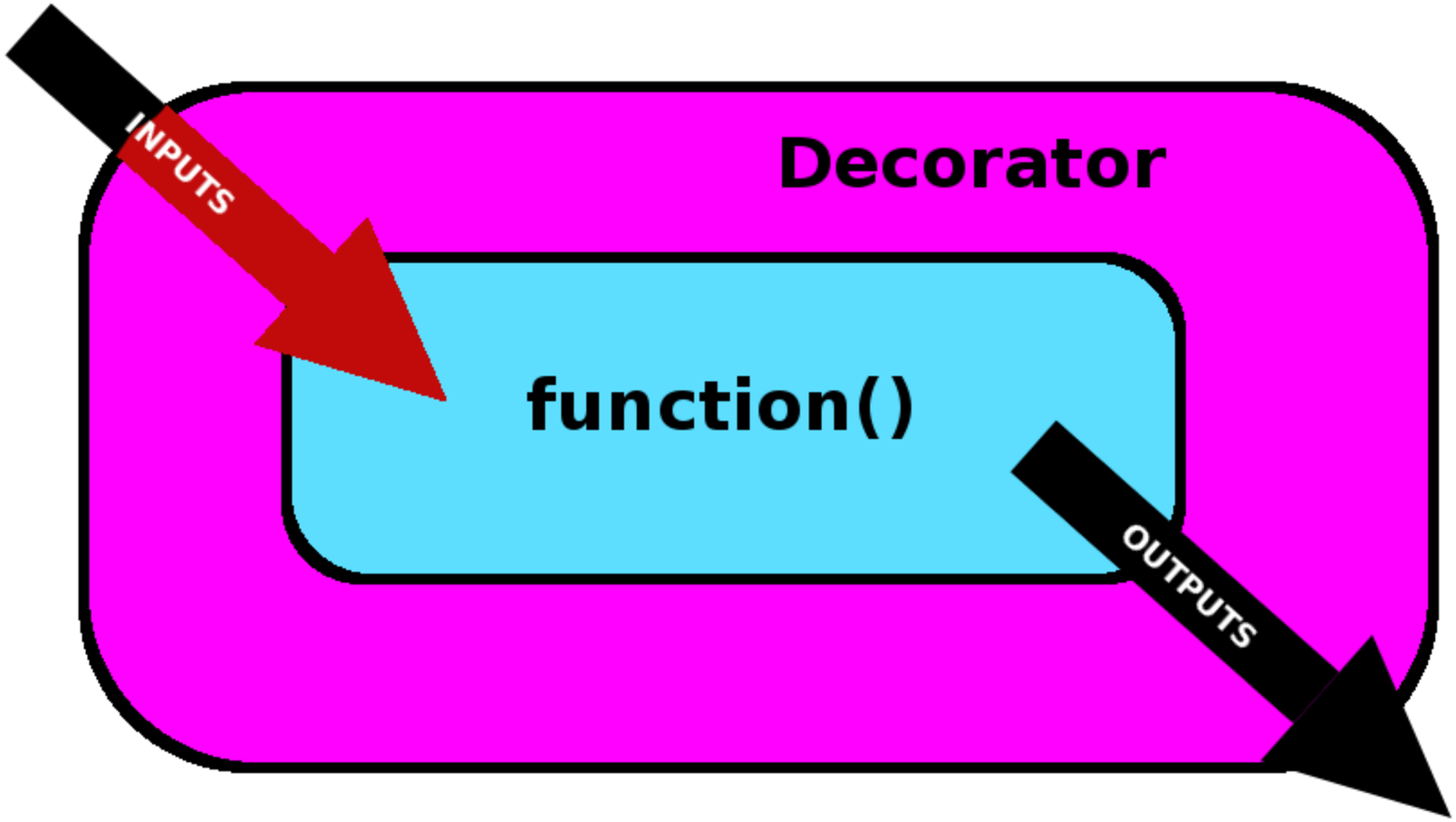


Decorators

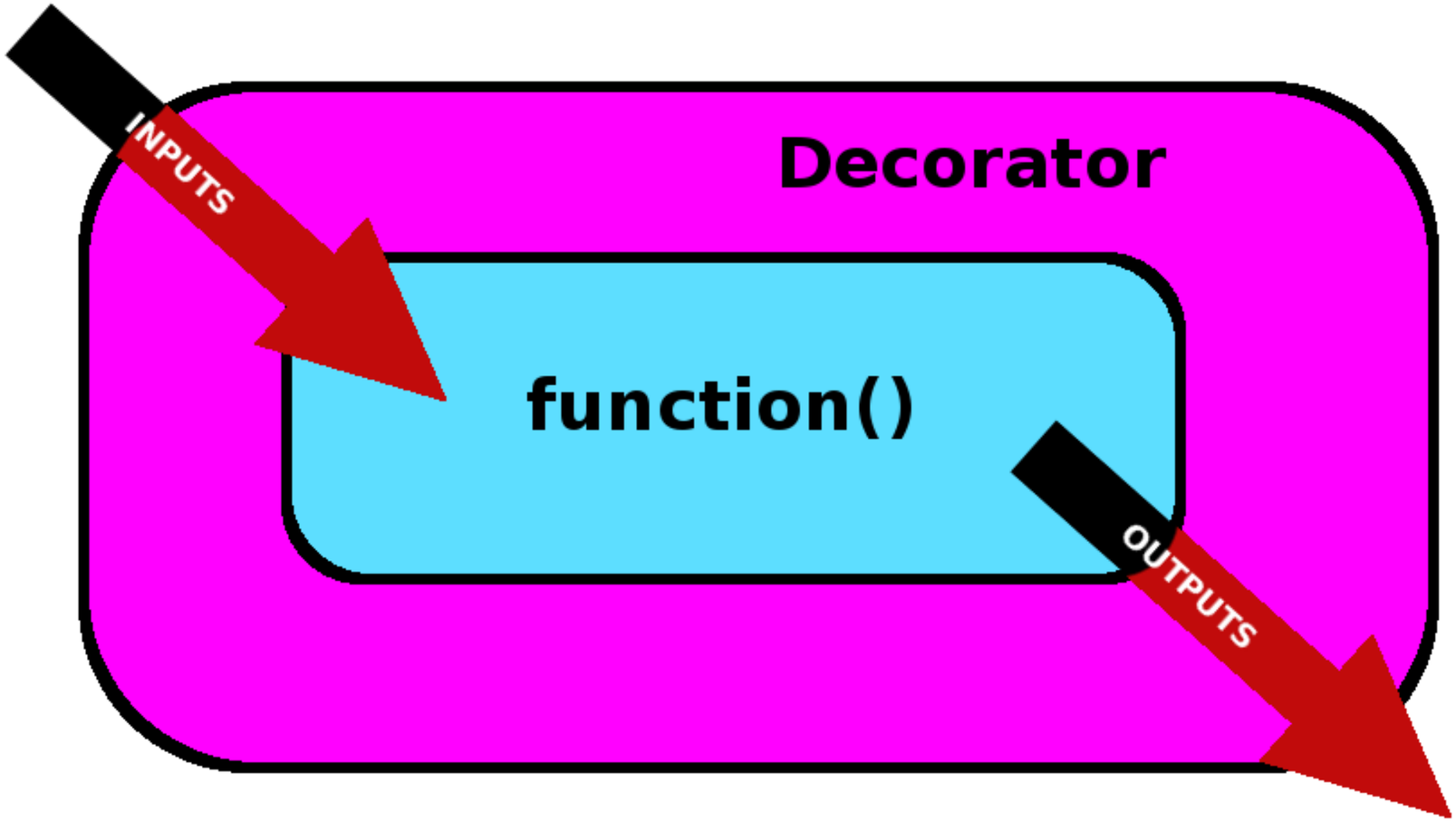
is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.



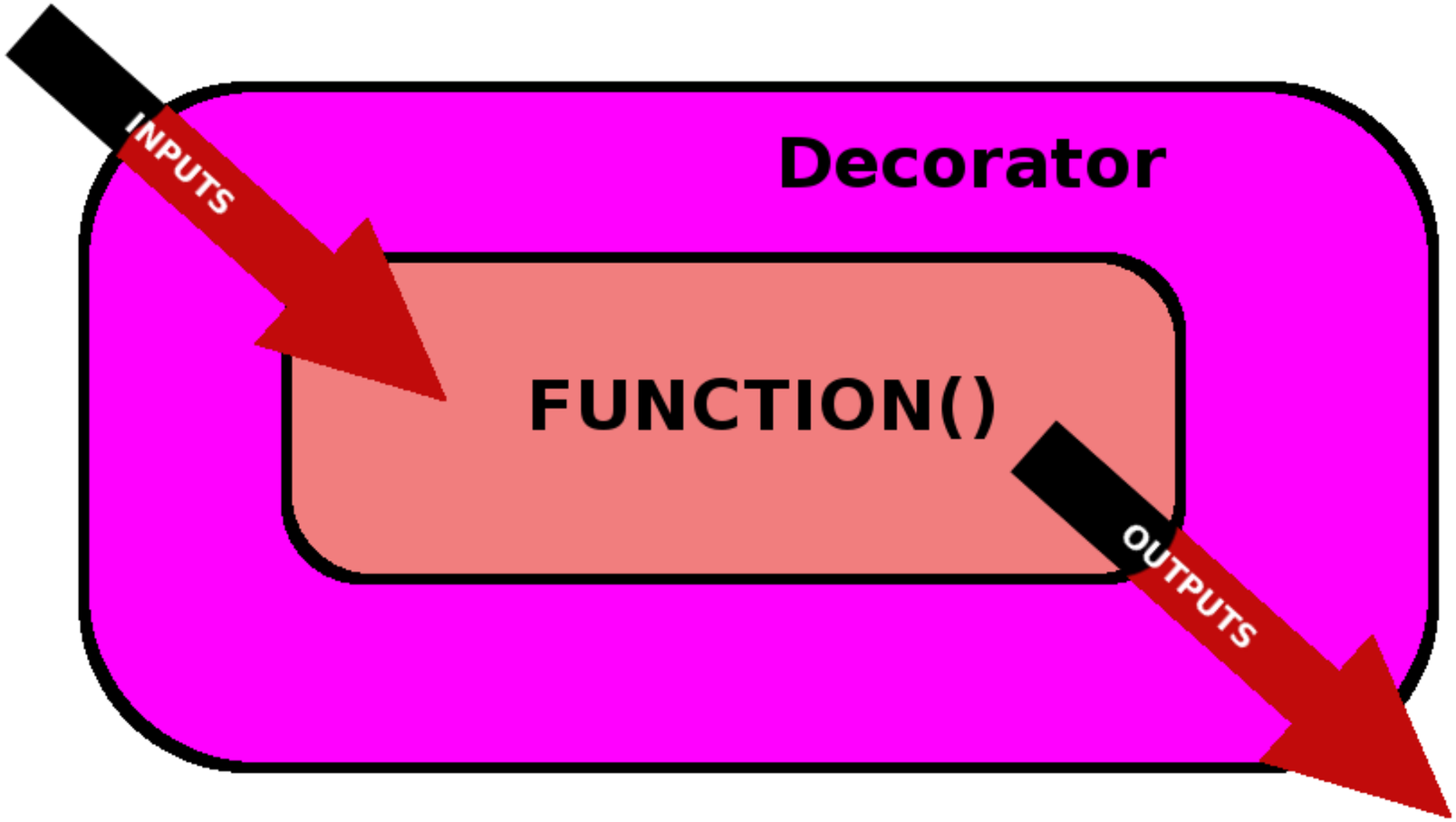
Modify inputs



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