NYU Center for Data Science

DS-GA 1007
Programming for Data Science

Lecture 4
Python III – Classes and Objects

Reminders

- Survey 2

Reminders

- Survey 2
- Homework 2
  - JupyterHub and Gradescope
  - Grader Contact Information

Reminders

- Survey 2
- Homework
  - JupyterHub and Gradescope
  - Grader Contact Information
- Forum
  - Homework, Lab, Lecture

Reminders

- Survey 2
- Homework
  - JupyterHub and Gradescope
  - Grader Contact Information
- Forum
  - Homework, Lab, Lecture
- Project

Agenda

- Review
  - Functions

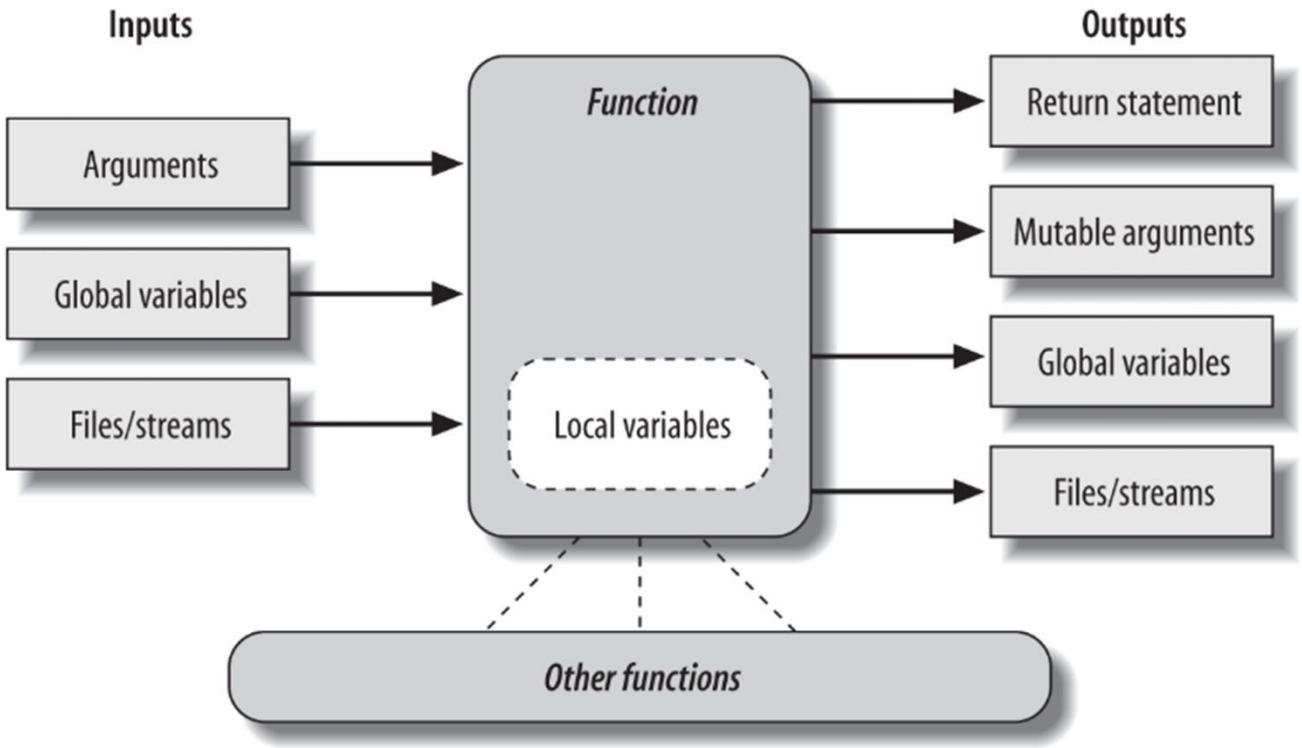
Agenda

- Review
- Lesson
  - Classes

Agenda

- Review
- Lesson
- Demo
  - Debugging

Functions


Inputs
Outputs
Function
Arguments
Global variables
Files/streams
Local variables
Return statement
Mutable arguments
Global variables
Files/streams
Other functions

Scope

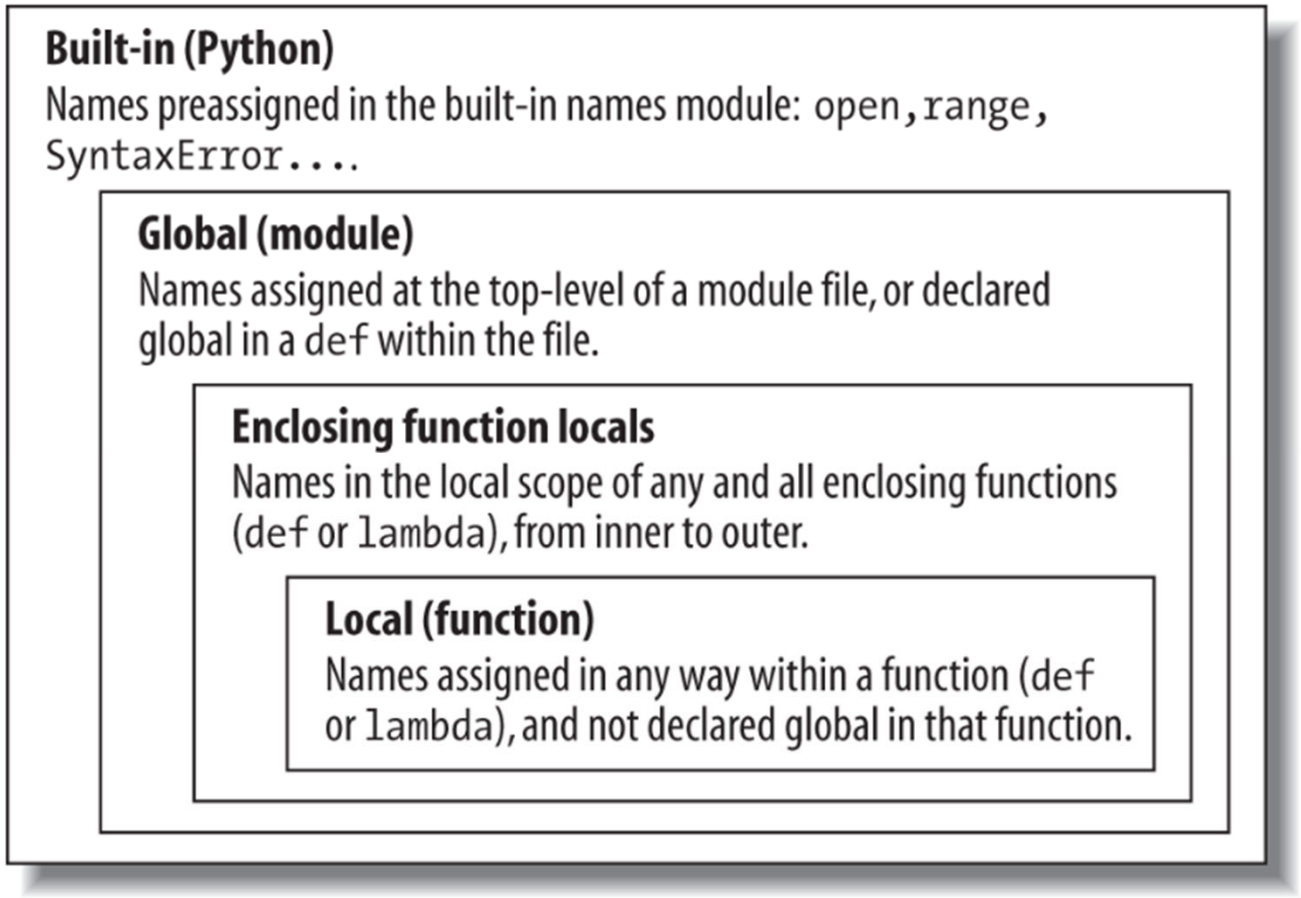
- Namespace
  - Program stores names in separate namespaces
  - Location of name in code determines namespace

Scope

- Namespace
  - Program stores names in separate namespaces
  - Location of name in code determines namespace
- Functions
  - If a variable is assigned inside a def, it is local to that function.
  - If a variable is assigned outside all defs, it is global to the entire file.

Scope

- LEGB Rule
  - Local
  - Enclosing
  - Global
  - Built-in



Pass by Reference vs Pass by Value

- Immutable arguments are “passed by value”
  - Objects such as integers and strings are passed by object reference instead of by copying
  - Since you can’t change immutable objects in place anyhow, the effect is much like making a copy.

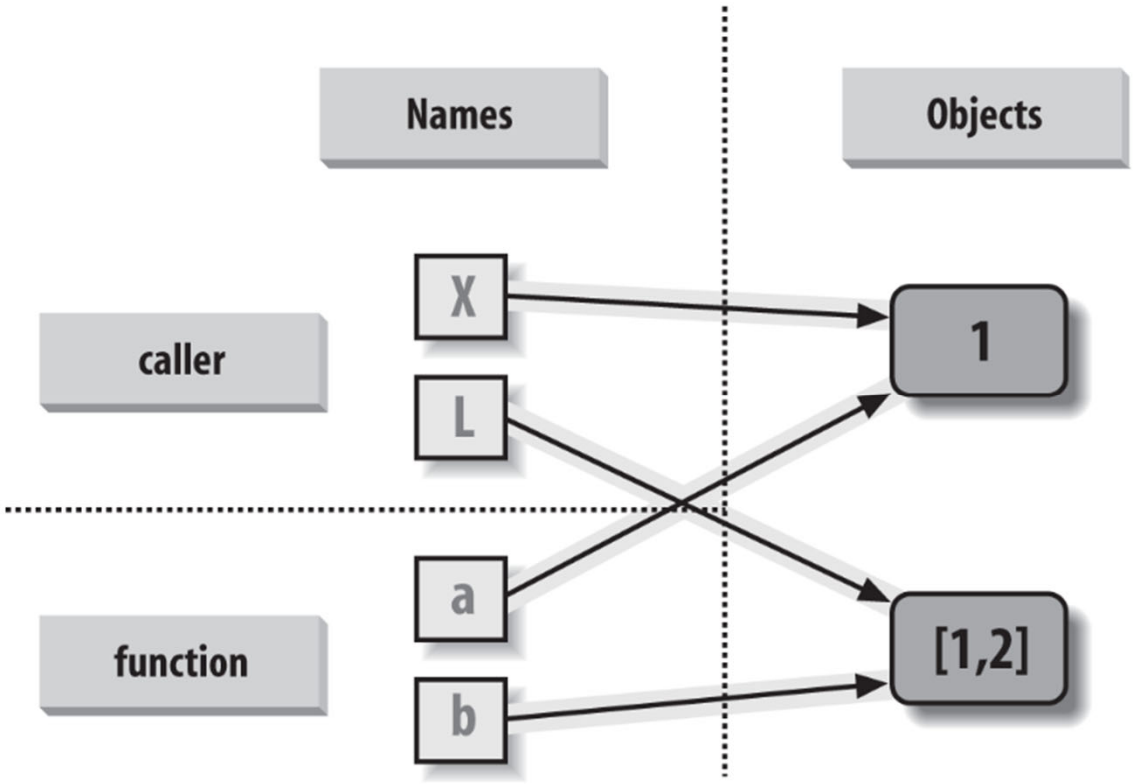
Pass by Reference vs Pass by Value

- Mutable arguments are “passed by reference”
  - Objects such as lists and dictionaries are passed by reference.
  - Mutable objects can be changed in place in the function

Pass by Reference vs Pass by Value

```
>>> def changer(a, b):
        a = 2
        b[0] = 'spam'

>>> X = 1
>>> L = [1, 2]
>>> changer(X, L)
>>> X, L
(1, ['spam', 2])
```



Positional Arguments vs Keyword Argument

Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name

Unpacking Arguments

`func(*iterable)`	Caller	Pass all objects in *`iterable`* as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in *`dict`* as individual keyword arguments

Default Arguments

`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call

Variable Number of Inputs

`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary

Recursion

```
>>> def mysum(L):
        print(L)
        if not L:
            return 0
        else:
            return L[0] + mysum(L[1:])

>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Storing Values in Functions

- Closure
  - Values of variables in local scope of function do not persist between calls. We can use inner functions to save values.
- Example (Currying functions)

```
def curry(first_argument, func):
    def new_func(*args):
        return func(first_argument, *args)

    return new_func

def adder(x, y):
    return x + y

curried_adder = curry(5, adder)
curried_adder(4)
```

Classes and Objects

- Objects
  - Date (called state or ***attributes***)
  - Functions (called behavior or ***methods***)
- Classes
  - Each object is ***instance*** of class
    - Specifies attributes and methods

Encapsulation

```
class atom(superclass, ...):
    class_attribute = value
    ...
    def method(self, ...):
        self.instance_attribute = value
        ...
```

- Class attribute
  - variable that is accessible by any instance of the class
  - All instances share the variable
- Instance attribute
  - only accessible by the instance that creates it
  - Like a local variable

Example (Instance attribute)

```
class atom:
    def init(self, atomic_number, x, y, z):
        self.atomic_number = atomic_number
        self.position = (x, y, z)
```

```
x = atom()
x.init(6, 0.0, 1.0, 2.0)  # Carbon (C)
y = atom()
y.init(10, 4.0, 3.0, 5.0) # Neon (Ne)
print(y.position)
```

Example (Class Attribute)

```
var = name()
```

```
var.class_attr = val
```

```
var.inst_attr = val
```

```
var.method(args...)
```

Inheritance

- Classes inherit common state and behavior from other classes
- Class that inherits from another class is called
  - subclass
  - Child class
  - Derived class
- Class that is inherited by other classes is called
  - superclass
  - Parent class
  - Base class

Example (Inheritance)

```
class Super:
  def __init__(self, x):
    … default code …

class Sub(Super):
    def __init__(self, x, y):
        Super.__init__(self, x)
        … custom code …
```

Example (Inheritance)

Methods named with double underscores __X__ are special hooks

Classes may override most built-in type operations

- __init__ for object construction
- __repr__ when printed or converted to a string
- __add__ for + operator X + Y
- __lt__, __gt__, etc for comparisons X < Y, X > Y, etc.

Example (Inheritance)

In Python classes anything with two leading underscores is private

__a_func __my_variable

Internally, these are replaced with a name that includes the class name

_*name*__a_func _*name*__my_variable

Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly

_b

Take-Aways

- Divide and Conquer
  - Split large problem into small problems. Call function recursively on smaller problems.

Take-Aways

- Divide and Conquer
  - Split large problem into small problems. Call function recursively on smaller problems.
- Encapsulation and Inheritance
  - Classes includes attributes and methods. Compare to Abstraction and Decomposition for functions

Take-Aways

- Divide and Conquer
  - Split large problem into small problems. Call function recursively on smaller problems.
- Encapsulation and Inheritance
  - Classes includes attributes and methods. Compare to Abstraction and Decomposition for functions
- Debugging
  - Halt the execution of code to locate bugs

Take-Aways

- Divide and Conquer
  - Split large problem into small problems. Call function recursively on smaller problems.
- Encapsulation and Inheritance
  - Classes includes attributes and methods. Compare to Abstraction and Decomposition for functions
- Debugging
  - Halt the execution of code to locate bugs