Object-Oriented Programming

Erin Rossiter

Namespace and Scope

- Namespace: “mapping from names to objects”

Namespace and Scope

- Namespace: “mapping from names to objects”
- Scope: level at which “a namespace is directly accessible”

Namespace and Scope

- Namespace: “mapping from names to objects”
- Scope: level at which “a namespace is directly accessible”
- Python follows the hierarchy:

Namespace and Scope

- Namespace: “mapping from names to objects”
- Scope: level at which “a namespace is directly accessible”
- Python follows the hierarchy:
  - Innermost scope: local names

Namespace and Scope

- Namespace: “mapping from names to objects”
- Scope: level at which “a namespace is directly accessible”
- Python follows the hierarchy:
  - Innermost scope: local names
  - Scopes of enclosing functions, innermost first

Namespace and Scope

- Namespace: “mapping from names to objects”
- Scope: level at which “a namespace is directly accessible”
- Python follows the hierarchy:
  - Innermost scope: local names
  - Scopes of enclosing functions, innermost first
  - Next-to-last scope: Global names in current module

Namespace and Scope

- Namespace: “mapping from names to objects”
- Scope: level at which “a namespace is directly accessible”
- Python follows the hierarchy:
  - Innermost scope: local names
  - Scopes of enclosing functions, innermost first
  - Next-to-last scope: Global names in current module
  - Outermost scope: Built-in names such as `int()`, `sum()`

Namespace and Scope

- Namespace: “mapping from names to objects”
- Scope: level at which “a namespace is directly accessible”
- Python follows the hierarchy:
  - Innermost scope: local names
  - Scopes of enclosing functions, innermost first
  - Next-to-last scope: Global names in current module
  - Outermost scope: Built-in names such as `int()`, `sum()`

Namespace and Scope

- Namespace: "mapping from names to objects"
- Scope: level at which "a namespace is directly accessible"
- Python follows the hierarchy:
  - Innermost scope: local names
  - Scopes of enclosing functions, innermost first
  - Next-to-last scope: Global names in current module
  - Outermost scope: Built-in names such as `int()`, `sum()`

Source: `https://docs.python.org/2/tutorial/classes.html`

Namespace: How does it work?

```
#A silly function that prints an integer.

def print_int(int):
    print 'Here is an integer: %s' %int

print_int(1)
print_int('b')
```

- But what is wrong with this?

Namespace: How does it work?

```
#A silly function that prints an integer.

def print_int(int):
    print 'Here is an integer: %s' %int

print_int(1)
print_int('b')
```

- But what is wrong with this?
- This works because the function first searches local scope of function then global scope.

Namespace: How does it work?

```
#A silly function that prints an integer.

def print_int(int):
    print 'Here is an integer: %s' %int

print_int(1)
print_int('b')
```

- But what is wrong with this?
- This works because the function first searches local scope of function then global scope.
- But, do not do this!

Namespace: How does it work?

```
#Function that returns the product of random draws from a uniform distribution.
def random_product(lower,upper):
    random1
    random2
    return random1 * random2

print random_product(0,1)

#NameError: global name 'random1' is not defined
```

Namespace: How does it work?

```
#We need to define numbers random1 and random2.
#We need to import the module random.

import random

def random_product(lower,upper):
    random1=uniform(lower,upper)
    random2=uniform(lower,upper)
    return random1 * random2

print random_product(0,1)

#NameError: global name 'uniform' is not defined
```

Namespace: How does it work?

```
#We need to add the module name before the global name.

import random

def random_product(lower,upper):
    random1=random.uniform(lower,upper)
    random2=random.uniform(lower,upper)
    return random1 * random2

print random_product(0,1)
```

Like the syntax `plyr::ddply()` in R

Namespace: How does it work?

```
#Alternatively, we can import a particular function.

from random import uniform

def random_product(lower,upper):
    random1=uniform(lower,upper)
    random2=uniform(lower,upper)
    return random1 * random2

print random_product(0,1)

#Use the following to import all functions of a module.

from random import *
```

Like the syntax `library(plyr)` in R

Class and Instance

- Classes helps you create your own objects with

Class and Instance

- Classes helps you create your own objects with
  - certain attributes

Class and Instance

- Classes helps you create your own objects with
  - certain attributes
  - ability to perform certain functions.

Class and Instance

- Classes helps you create your own objects with
  - certain attributes
  - ability to perform certain functions.
- An instance is a particular realization of a class.
- You use attributes and methods of classes all the time in R

Example

```
> ols <- lm(1:10 ~ seq(2,20,2))
> class(ols)
[1] "lm"
> plot(ols)
> summary(ols)
Call:
lm(formula = 1:10 ~ seq(2, 20, 2))

Residuals:
Min          1Q      Median          3Q         Max
-5.661e-16 -1.157e-16  4.273e-17  2.153e-16  4.167e-16

Coefficients:
Estimate Std. Error    t value Pr(>|t|)
(Intercept)   1.123e-15  2.458e-16 4.571e+00  0.00182 **
...
...
```

Class and Instance: How to do it?

```
#Create a class

class human(object):

    latin_name='homo sapien' #Attribute for the class

#Create an instance of a class and name it 'me'.

me=human()
```

Class and Instance: How to do it?

```
class human(object):

    latin_name='homo sapien' #Attribute for the class

    #Add attributes for the instances.
    def __init__(self, age, sex, name): #initializer or constructor
        self.age = age
        self.name = name
        self.sex = sex
```

Class and Instance: How to do it?

- You can set default values for attributes.

Class and Instance: How to do it?

- You can set default values for attributes.
- Make sure you list non-default arguments first.

Class and Instance: How to do it?

- You can set default values for attributes.
- Make sure you list non-default arguments first.

```
class human(object):

    latin_name='homo sapien' #Attribute for the class

    #Add attributes for the instances.
    def __init__(self, age, sex, name=None): #initializer or constructor
        self.age = age
        self.name = name
        self.sex = sex
```

Class and Instance: How to do it?

```
class human(object):

    latin_name='homo sapien' #Attribute for the class

    #Add attributes for the instances.
    def __init__(self, age, sex, name=None): #initializer or constructor
        self.age = age
        self.name = name
        self.sex = sex

    #Add some functions

    def speak(self, words):
        return words

    def introduce(self):
        if self.sex=='Female': return self.speak("Hello, I'm Ms. %s" % self.name)
        elif self.sex=='Male': return self.speak("Hello, I'm Mr. %s" % self.name)
        else: return self.speak("Hello, I'm %s" % name)
```

`dir(human)` lists all the methods of the class.

Inheritance and Polymorphism

- Inheritance enables you to create sub-classes that inherit the methods of another class.

Inheritance and Polymorphism

- Inheritance enables you to create sub-classes that inherit the methods of another class.
- Polymorphism adapts a given method of a class to its sub-classes.

Inheritance and Polymorphism

- Inheritance enables you to create sub-classes that inherit the methods of another class.
- Polymorphism adapts a given method of a class to its sub-classes.
- Keep it DRY (don't repeat yourself)