

Object Oriented Python

Daniel Driver

E177-Advanced Matlab

April 21, 2015

Overview

- 1 Python OOP
- 2 Python Nitty Gritty

Python OOP

Python OOP

Defining a Class

Classic-style Class

- DON'T DO THIS
- Classic-style Class
- pre version 2.1
- Type always "instance"
- Removed in Python 3
- definition:
 - `class ExampleClass():`
- Creation
 - `>>>a=ExampleClass()`

New-Style Class

- ONLY USE THIS
- introduced in 2.2
- Type matches class name
- Created by inheriting object class
- definition:
 - `class ExampleClass(object):`
- Creation
 - `>>>a=ExampleClass()`

<https://docs.python.org/2/reference/datamodel.html#new-style-and-classic-classes>
python3 documentation

Adding a Method

Basic Example with Methods

```
class MethodBasicExample(object):
    def __init__(self,*arg):
        if len(arg)==1:
            self.x=arg[0]
        else:
            self.x=0
    def SquareX(self):
        return self.x*self.x
    def StoreY(self,y):
        self.y=y
    def SquareYAddInput(self,Input):
        return self.y*self.y+Input
```

- `__init__` special method
 - called during construction
 - used for initialization-create attributes and defaults
 - any valid python code can go here
- First input is object, convention to call "self"
- `*arg` is list of inputs
 - used for variable number of Inputs

Usage

```
>>>a=MethodBasicExample(3)
>>>SqrX=a.SquareX()
>>>print(SqrX)
9
>>>SqY=a.SquareYAddInput(5)
Traceback (most recent call last):
  File "OOPBasics.py", line 70, in <module>
    SqY=a.SquareYAddZ(5)
AttributeError: 'MethodBasicExample' object
        has no attribute 'SquareYAddZ'
>>>a.StoreY(4) #store value in self.y
>>>SqY=a.SquareYAddInput(5)
>>>print(SqY)
21
```

- Methods called with "dot" syntax
- Attributes can be added at anytime

Inheritance

Basic Example with Methods

```
class InheritBasicExample(MethodBasicExample):  
    def ThreeTimesX(self):  
        return 3*self.x
```

- Can inherit multiple classes
- simple syntax:
 - `class Classname(inherited1,inherited2)`
- To call parent class method
 - if not overridden just call it like normal
 - if overridden but still want parent version
 - use "super" (like @ in MATLAB)
 - `super().SuperClassFunction(input1,input2)`

Usage

```
>>>from OOPBasics import *  
>>>c=InheritBasicExample(2)  
>>>print(c.x)  
2  
>>>ThreeX=c.ThreeTimesX()  
>>>print(ThreeX)  
6  
>>>SqrX=c.SquareX()  
>>>print(SqrX)  
4
```

- All methods of MethodBasicExample
- No need to call parent `__init__` (Unlike MATLAB)

Abstract Methods

Basic Example with abc

```

from abc import ABCMeta, abstractmethod
class AbstractClass(object, metaclass=ABCMeta):
    @abstractmethod
    def TestFunc1(self, x):
        pass
    @abstractmethod
    def TestFunc2(self):
        return 'Parent String '
class ConcreteClass(AbstractClass):
    def TestFunc1(self, x):
        return x**3
    def TestFunc2(self):
        s=super().TestFunc2()
        return s+' and Child String '

```

- abc is “Abstract Base Class”
- @abstractmethod marks method as abstract
 - child must implement to be concrete

<https://docs.python.org/2/library/abc.html>
python 3 documentatation

Usage

```

>>>d=AbstractClass()
Traceback (most recent call last):
  File "OOPBasics.py", line 122,
in <module>
    d=AbstractClass()
TypeError: Can't instantiate
    abstract class
    AbstractClass with abstract
    methods TestFunc1, TestFunc2
>>>e=ConcreteClass()
>>>out1=e.TestFunc1(6)
>>>print(out1)
216
>>>out2=e.TestFunc2()
>>>print(out2)
Parent String  and Child String

```

Matlab Like Property with @property

Create Property x

```
class PropBasicExample(object):
    def __init__(self):
        self._x=0
    @property
    def x(self):
        "I am the 'x' property."
        return self._x
    @x.setter
    def x(self, value):
        if value >5:
            self._x = 5
        else:
            self._x = value
    @x.deleter
    def x(self):
        del self._x
```

Result

```
>>> from OOPBasics import PropBasicExample
>>> a=PropBasicExample()
>>> print(a.x)
0
>>> a.x=3
>>> print(a.x)
3
>>> a.x=100
>>> print(a.x)
5
```

- @property - def x function determines the getter and doc string in help
- @ is a decorator -@property replace x with x=property(x) where the x on the right comes from def x(self) more at [Stack Overflow Property Decorator link](#)
- @PropertyName.setter-adds setter to PropertyName
- No need to import property like abstractmethod. Loaded when interpreter starts

Access Control

- “We’re all consenting adults here”
 - No public or private
 - Programmer is responsible for not breaking things
- Python has syntax to indicate what should not be touched
- “_Something” means “I am not part of the API”-aka Not meant to be used by user but still available though hidden
- “__Something” Python renames method or attribute so it cannot(is hard to accidentally) overwrite in subclass
 - makes it so effectively only callable withing class that defines it
 - “obj.__Name” not available from interpreter either

<http://igorsobreira.com/2010/09/16/difference-between-one-underline-and-two-underlines-in-python.html>

Overloading operators

- “__Something__” - method python calls based on other syntax
 - used in overloading
 - special method
 - <https://docs.python.org/2/reference/datamodel.html#special-method-names>
 - [Python3 link](#)
- Numeric
 - `__add__` overload “+”
 - `__subtract__` overloads “-”
 - `__ge__` overloads “>”
 - `__len__` overloads “len()”
 - <https://docs.python.org/2/reference/datamodel.html#emulating-numeric-types>
 - [Python 3 link](#)
- Indexing
 - `__getitem__(self, key)` overloads “self[key]”
 - <https://docs.python.org/2/reference/datamodel.html#emulating-container-types>
 - [Python 3 link](#)

display: __repr__ vs __str__

how we match the behavior of “display” in matlab

- `__repr__` is called by when interpreter displays
 - if no `__repr__` just displays parent method object.`__repr__`
- `__str__` is called by `print`
 - If no `__str__` defaults to `__repr__`
- Both must return a string

Usage

```
>>> import Geometry
>>> a=Geometry.Point(1.0,0.0)
>>> print(a)
In __str__
Point=<1.0,0.0>
>>> a
In __repr__
Point=<1.0,0.0>
```

Example Geometry.py

Point

```
class Point(object):
    def __init__(self,*arg):
        Nargin=len(arg)
        if Nargin==0:
            self.x=0
            self.y=0
        else:
            self.x=arg[0]
            self.y=arg[1]
        @property
    def x(self):
        return self._x
    @x.setter
    def x(self,value):
        if not(isinstance(value,float)):
            raise ValueError('x must be a float')
        else:
            self._x=value
        @property
    def y(self):
        return self._y
    @y.setter
    def y(self,value):
        if not isinstance(value,float):
            raise ValueError('x must be a float')
        else:
            self._y=value
```

Example Geometry.py

Point(Cont.)

```
def __add__(self, PointIn):
    if not isinstance(PointIn, Point):
        raise ValueError('Inputs need to be points')
    else:
        return Point(self.x+PointIn.x, self.y+PointIn.y)
def __sub__(self, PointIn):
    if not isinstance(PointIn, Point):
        raise ValueError('Inputs need to be points')
    else:
        return Point(self.x-PointIn.x, self.y-PointIn.y)
def distance(self, Point2):
    from math import sqrt
    diff=self-Point2
    return sqrt(diff.x**2+diff.y**2)
def __repr__(self):
    return "In __repr__\n Point=<{0},{1}>"\
           .format(self.x, self.y)
def __str__(self):
    return "In __str__\n Point=<{0},{1}>"\
           .format(self.x, self.y)
```

Example Geometry.py

Line Segment

```
class LineSegment(object):
    def __init__(self, Point1, Point2):
        if isinstance(Point1, Point) and isinstance(Point2, Point):
            self._Start=Point1
            self._Finish=Point2
        else:
            raise ValueError('Inputs Must by Points Classes')
    @property
    def Start(self):
        return self._Start
    @Start.setter
    def Start(self, value):
        if not(isinstance(value, float)):
            raise ValueError('Start must be a float')
        else:
            self._Start=value
    @property
    def Finish(self):
        return self._Finish
    @Finish.setter
    def Finish(self, value):
        if not isinstance(value, Point):
            raise ValueError('Start must be a Point')
        else:
            self._Finish=value
```

Example Geometry.py

Line Segment(Cont.)

```
def length(self):
    return self.Start.distance(self.Finish)
def __repr__(self):
    import matplotlib.pyplot as pyplot
    pyplot.plot([self.Start.x, self.Finish.x],[self.Start.y, self.Finish.y])
    pyplot.xlabel('X')
    pyplot.ylabel('Y')
    pyplot.show()
    return "Plotted LineSegment"
```

Usage

```
import Geometry
a=Geometry.Point(0.0,0.0)
b=Geometry.Point(1.0,1.0)
c=a-b
print(c)
l=Geometry.LineSegment(a,b)
l.length()
print(l)
```

- `scriptsize` produces "c" with coordinates $\langle -1, -1 \rangle$
- `l` is `LineSegment` from $\langle 0, 0 \rangle$ to $\langle 1, 1 \rangle$ and `length`= $\sqrt{2}$
- `print(l)`- result is plot of line `l`

One Last Thing: Making A Package

Steps to Create a Python Package:

- 1 Create a directory and give it your package's name.
- 2 Put your classes in it.
- 3 Create a `__init__.py` file in the directory

<http://www.pythoncentral.io/how-to-create-a-python-package/>

Python Nitty Gritty

Python Nitty Gritty

Constructor: `__new__` vs `__init__`

`__new__`

- Use to control class creation
- actually constructs object
- Default: calls `__init__` and passes inputs along
- Almost never overridden (just allow default creation)
- Possible Use: Completely prevent allocation
 - Hard to find good examples why this would ever be used

`__init__`

- Will mostly use this
- initializes an instance
- takes instance created by `__new__` as input
 - this is why it has a “self” input
- returns nothing (not directly at least -any return goes to `__new__`)

<http://stackoverflow.com/questions/8106900/new-and-init-in-python>

<http://stackoverflow.com/questions/674304/pythons-use-of-new-and-init>

How is an Attribute Actually Retrieved

dot syntax

- like “ans=a.x”
- “.” overloaded by `__getattr__(self,name)`
 - name is 'x' in this case
- Default:
 - Look for 'x' in `a.__dict__`
 - Next look for 'x' in `a.__class__.__dict__`
 - If not found try to call `__getattr__(self,name)`

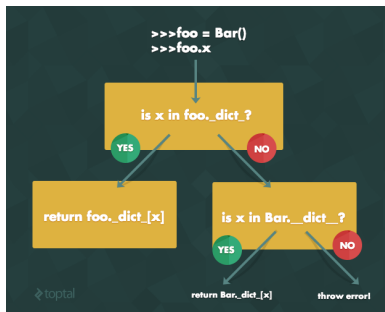


Figure : This is object.`__getattr__` but “throw error” should be replaced by “try Bar.`__getattr__`” If that fails THEN throw error!

<http://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide>

Example of getting Attribute

Class Definition

```
class GetAttrExampleClass(object):
    ClassData=8
    def __init__(self ,N, xIn):
        self.N=N
        self._HiddenDataForX=xIn
    def __getattr__(self ,name):
        if name=='x':
            print('x not found in
                Class or Instance Dictionaries
                so Now It defaulted to
                __getattr__ and will
                return self._HiddenDataForX ')
            return self._HiddenDataForX
        else:
            return object.\
                __getattr__(self ,name)
```

Get x attribute

```
>>>a=GetAttrExampleClass(5,3)
>>>print(a.x)
x not found in Class or
Instance Dictionaries so
Now It defaulted to __getattr__
nd will return self._HiddenDataForX
3
#Now Give x a value in the
#dictionary by setting it with
a.x=100
>>>a.x=100
>>>print(a.x)
100
```

- x not found in a.__dict__ or a.__class__.__dict__ so calls __getattr__
- after a.x=100
 - 'x' is in a.__dict__
 - so __getattr__ not called

Example: Class vs Instance Data

Class Definition

```
class GetAttrExampleClass(object):
    ClassData=8
    def __init__(self ,N, xIn):
        self.N=N
        self._HiddenDataForX=xIn
    def __getattr__(self ,name):
        if name=='x':
            print('x not found in
                  Class or Instance Dictionaries
                  so Now It defaulted to
                  __getattr__ and will
                  return self._HiddenDataForX')
            return self._HiddenDataForX
        else:
            return object.\
                __getattr__(self ,name)
```

Change ClassData

```
>>>print ( GetAttrExampleClass . ClassData )
8
>>>a=GetAttrExampleClass (5, 3)
>>>print (a . ClassData )
8
>>>print ( GetAttrExampleClass . ClassData )
8
>>>a . ClassData=79
>>>print (a . ClassData )
79
>>>print ( GetAttrExampleClass . ClassData )
8

#change the Class Data
>>>b=GetAttrExampleClass (10001, 53)
>>>print (b . ClassData )
8
>>>GetAttrExampleClass . ClassData=13
>>>print ( GetAttrExampleClass . ClassData )
13
>>>print (b . ClassData )
13
>>>print (a . ClassData )
79
```

Class Vs Instance Explained

- b.ClassData changes because not overwritten so still looking at b.__class__.__dict__ (same place as as GetAttrExampleClass.ClassData)
- a.ClassData changes because it is looking at a.__dict__
- Data like “ClassData” is often mistakenly though of as a default value
 - if a third instance was made “c=GetAttrExampleClass(1,1)”
 - print(c.ClassData) => 13
 - If it was a default we would expect 8
 - represents data shared by class

Property Alternative Syntax-Equivalent

Original with @property

```
class PropBasicExample(object):
    def __init__(self):
        self._x=0
    @property
    def x(self):
        "I am the 'x' property."
        return self._x
    @x.setter
    def x(self, value):
        if value >5:
            self._x = 5
        else:
            self._x = value
    @x.deleter
    def x(self):
        del self._x
```

Using property directly

```
class PropExample(object):
    def __init__(self):
        self._x=0
    def getx(self):
        return self._x
    def setx(self, value):
        if value >5:
            self._x = 5
        else:
            self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx,
                 "I'm the 'x' property.")
```

- Note:Example is based on help(property) examples
- Both produce the same functionality for attribute "x"
- Original uses Decorators to wrap functions and rename them as the property getter and setter functions (@ syntax) [Blog Post about Decorators](#)

Abstract Class - Duck Typing

duck typing

- “if it looks like a duck and quacks like a duck it is a duck”
 - Aka if it works then it is correct
 - if it fulfills the class duties then it is a member of the class
- documentation enforces interface
 - error if function is called that is not implemented
- How is this different @abstractmethod version
 - Basically what error is produced when an interface is not fulfilled
- With duck typing:
 - “Attribute not found” error is thrown when attempt to use
- with @abstractmethod
 - “AbstractMethod not implemented” error is thrown