Classes

SURP 2022 Python Bootcamp

Ohio State Astronomy

Slides by: James W. Johnson

What are they? Why should I care?

A *class* is how you implement a new object.

Python is an object-oriented programming language - if you're a python programmer and you've never written a class, you're missing out on the *single most powerful* aspect of the language *by far*.

Recall: The Python Model

Everything is an object in Python, whether you knew it or not.

Objects have …

- Attributes
- Functions
- Data

… which are unique to the object; also interactions with other objects.

Data Container

The simplest object – all it does is hold on to the attributes that you give it

class container: pass is all you need

- Doesn't *need* to be called container

The argument *pass* here means "do nothing." Objects with more specialization instead have *def* statements within them.

```
In [1]: import numpy as np

In [2]: class container:
   ...:     pass
   ...:

In [3]: x = container()

In [4]: x.masses = abs(np.random.normal(size = 10))

In [5]: x.radii = abs(np.random.normal(size = 10))

In [6]: x.luminosities = abs(np.random.normal(size = 10))

In [7]: x.masses
Out[7]:
array([0.63136228, 0.52197144, 0.08581841, 0.3795803 , 0.95837668,
       0.53274021, 0.55630111, 1.62337898, 2.37011049, 1.89884227])

In [8]: x.radii
Out[8]:
array([1.06055823, 1.9124843 , 0.84339752, 0.08461345, 0.2870194 ,
       0.76955582, 0.09024124, 1.46008861, 1.24039583, 0.62426455])

In [9]: x.luminosities
Out[9]:
array([1.6057846 , 0.03445241, 0.0756965 , 0.47629528, 0.58423559,
       0.63525491, 0.08993064, 0.307223  , 0.0490746 , 0.07160734])
```

An Example Object: A Dog

Attributes: color, breed, name, gender

Functions: bark, roll over, shake, eat, drink, chase their tail

Data: date of birth, veterinary records, previous owners

Interactions with other objects: play with other dogs/owner, chase cats

Dogs in Python

How you create instances of the class (i.e. objects) is determined by the __init__ function.

The first argument to __init__ should always be *self* – this is true of most functions in a class, and refers to the object itself being passed

```
class dog:

    r"""
    Implements a dog in python
    """

    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
```

```
In [2]: Spot = dog("Spot", "Dalmatian")

In [3]: Spot.name
Out[3]: 'Spot'

In [4]: Spot.breed
Out[4]: 'Dalmatian'

In [5]: Snoopy = dog("Snoopy", "Beagle")

In [6]: Snoopy.name
Out[6]: 'Snoopy'

In [7]: Snoopy.breed
Out[7]: 'Beagle'
```

Dogs in Python

How you create instances of the class (i.e. objects) is determined by the __init__ function.

The first argument to __init__ should always be *self* – this is true of most functions in a class, and refers to the object itself being passed

This gets the job done, but…

```
class dog:

    r"""
    Implements a dog in python
    """

    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
```

```
In [2]: Spot = dog("Spot", "Dalmatian")

In [3]: Spot.name
Out[3]: 'Spot'

In [4]: Spot.breed
Out[4]: 'Dalmatian'

In [5]: Snoopy = dog("Snoopy", "Beagle")

In [6]: Snoopy.name
Out[6]: 'Snoopy'

In [7]: Snoopy.breed
Out[7]: 'Beagle'
```

Dogs in Python

How you create instances of the class (i.e. objects) is determined by the __init__ function.

The first argument to __init__ should always be *self* – this is true of most functions in a class, and refers to the object itself being passed

This gets the job done, but it's easily broken

```
class dog:

    r"""
    Implements a dog in python
    """

    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
```

```
In [5]: Snoopy = dog("Snoopy", "Beagle")

In [6]: Snoopy.name
Out[6]: 'Snoopy'

In [7]: Snoopy.breed
Out[7]: 'Beagle'

In [8]: Snoopy.name = 3

In [9]: Snoopy.breed = [0, 3, 7]

In [10]: Snoopy.name
Out[10]: 3

In [11]: Snoopy.breed
Out[11]: [0, 3, 7]
```

Dogs in Python

Error-handling of attributes requires *property* and *setter* functions

self._property is a conventional way of storing *self.property* under the hood, protected by error-handling

This throws a *TypeError* whenever the user tries to set *name* or *breed* to something other than a *str*

```python
class dog:

    r"""
    Implements a dog in python
    """

    def __init__(self, name, breed):
        self.name = name
        self.breed = breed


    @property
    def name(self):
        r"""
        Type : str

        The name of the dog
        """
        return self._name


    @name.setter
    def name(self, value):
        if isinstance(value, str):
            self._name = value
        else:
            raise TypeError("Attribute 'name' must be a string. Got: %s" % (
                type(value)))


    @property
    def breed(self):
        r"""
        Type : str

        The breed of the dog
        """
        return self._breed

    @breed.setter
    def breed(self, value):
        if isinstance(value, str):
            self._breed = value
        else:
            raise TypeError("Attribute 'breed' must be a string. Got: %s" % (
                type(value)))
```

Dogs in Python

Error-handling of attributes requires *property* and *setter* functions

self._property is a conventional way of storing *self.property* under the hood, protected by error-handling

This throws a *TypeError* whenever the user tries to set *name* or *breed* to something other than a *str*

```
In [2]: Snoopy = dog("Snoopy", "Beagle")

In [3]: Snoopy.name
Out[3]: 'Snoopy'

In [4]: Snoopy.breed
Out[4]: 'Beagle'

In [5]: Snoopy.name = 3
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-88ab4b918e8c> in <module>
----> 1 Snoopy.name = 3

~/Work/Teaching/OSUAstroSURP2020/examples/classes/dog.py in name(self, value)
     27                 else:
     28                         raise TypeError("Attribute 'name' must be a string. Got: %s" % (
---> 29                                 type(value)))
     30
     31

TypeError: Attribute 'name' must be a string. Got: <class 'int'>

In [6]: Snoopy.breed = [0, 3, 7]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-abbb0c87354e> in <module>
----> 1 Snoopy.breed = [0, 3, 7]

~/Work/Teaching/OSUAstroSURP2020/examples/classes/dog.py in breed(self, value)
     45                 else:
     46                         raise TypeError("Attribute 'breed' must be a string. Got: %s" % (
---> 47                                 type(value)))
     48
     49

TypeError: Attribute 'breed' must be a string. Got: <class 'list'>
```

A Quick Note: Property vs. Stored Data

Important: the name of a *property* is NOT related to the variables stored as data by a class

- The connection between *obj.x* and *obj._x* is purely for readability – if a value stored under the hood may be of use to the user, it is simply convention to use *x* and *_x*

In this example, the values of certain properties are calculated "on the fly" based only one value actually stored as internal data

The *@property* decorator removes the need for parentheses

```
class example:

    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        return self._value

    @property
    def onemore(self):
        return self._value + 1

    @property
    def twomore(self):
        return self._value + 2
```

```
In [2]: x = example(4)

In [3]: x.value
Out[3]: 4

In [4]: x.onemore
Out[4]: 5

In [5]: x.twomore
Out[5]: 6
```

Dogs in Python

Classes can have functions too. These functions can have any number of subroutines, just like other functions, and can access properties and other functions via *self*

```
def speak(self):
    r"""
    Makes the dog bark when you tell it to speak.
    """
    print("%s says \"Woof!\"" % (self.name))
```

```
In [1]: from dog import dog

In [2]: snoopy = dog("Snoopy", "Beagle")

In [3]: snoopy.speak()
Snoopy says "Woof!"

In [4]: dog.speak(snoopy)
Snoopy says "Woof!"
```

The interpretation of *self*: refers to the *instance* of the class (*x.func()* is the same as *classname.func(x)*). On line 4 here, I've passed *snoopy* to *dog.speak* as *self*.

Static Methods

Functions which are bound to the class and not the object of the class

Can’t access or modify the class

Implemented as part of a class because it makes sense to do so

Created with the *@staticmethod* decorator

Dogs in Python

An example static method: *is_puppy* to determine if a dog is a puppy or not.

Do not take *self* as an argument, and are called with the name of the class

```
In [1]: from dog import dog

In [2]: dog.is_puppy(1)
Out[2]: True

In [3]: dog.is_puppy(5)
Out[3]: False
```

```python
@staticmethod
def is_puppy(age):
    r"""
    Determine if a dog is a puppy.

    Signature: dog.is_puppy(age)

    Parameters
    ----------
    age : int
        The age of the dog, in years

    Returns
    -------
    puppy : bool
        True if age < 2 years, otherwise False.
    """
    if isinstance(age, int):
        return age < 2
    else:
        raise TypeError("Must be an integer. Got: %s" % (type(age)))
```

Class Methods

Like static methods, are bound to the class rather than objects of the class

Can access and modify class state, unlike static methods

Return an instance of the class (i.e. an object)

Created with the @classmethod decorator

Dogs in Python

An example class method: create Snoopy

They take the class as the first argument (cls)

```
@classmethod
def snoopy(cls):
    r"""
    Returns Snoopy.
    """
    return cls("Snoopy", "Beagle")
```

```
In [1]: from dog import dog

In [2]: snoopy = dog.snoopy()

In [3]: snoopy.name
Out[3]: 'Snoopy'

In [4]: snoopy.breed
Out[4]: 'Beagle'
```

Syntactic Sugar

A line of code which is interpreted the same as another, but is more readable

You've been using it all along, you just didn't know it

To implement syntactic sugar, you as the programmer write functions with specific names often referred to as "magic methods"

Syntactic Sugar

Can be used to emulate array-like indexing, item assignment, calling, and more

With Syntactic Sugar	Without Syntactic Sugar
x[0]	x.__getitem__(0)
x(0)	x.__call__(0)
x[0] = 1	x.__setitem__(0, 1)
str(x)	x.__str__() and x.__repr__()

There are many other forms of syntactic sugar – here is a reference on many of the magic methods you can implement: https://www.tutorialsteacher.com/python/magic-methods-in-python

Example Usage of Syntactic Sugar: Polynomials

Goals:

- Properties: coefficients and the order of the polynomial
- Indexing – index *i* should return the *i*'th coefficient
- Calling – *f(x)* should evaluate the polynomial at the value of *x*
- Item Assignment – *f[i] = a* should assign the *i*'th coefficient to the value of *a*
- A string representation

Note: This object is scripted at examples/mypkg/mathlib/polynomial.py in the git repository

A Polynomial Object

The first pieces of the implementation: the __init__ function and the properties

```
class polynomial:

    r"""
    N-th degree mathematical polynomial functions f(x)

    Parameters
    ----------
    coeffs : array-like
        The coefficients of the polynomial. See attribute below.
        Must be either a list, tuple, or numpy array.

    Attributes
    ----------
    coeffs : numpy array
        The coefficients of the polynomial, in order of increasing exponent on
        the independent variable x.
    order : int
        The order of the polynomial.
    """

    def __init__(self, coeffs):
        self.coeffs = coeffs
```

```
    @property
    def coeffs(self):
        r"""
        Type : numpy array

        The numerical coefficients of the polynomial, in order of increasing
        exponent on x
        """
        return self._coeffs

    @coeffs.setter
    def coeffs(self, value):
        if isinstance(value, list) or isinstance(value, tuple):
            if all([isinstance(i, numbers.Number) for i in value]):
                self._coeffs = np.array(value)
            else:
                raise TypeError("Non-numerical value in coefficients.")
        elif isinstance(value, np.ndarray):
            if all([isinstance(i, numbers.Number) for i in value]):
                self._coeffs = value[:]
            else:
                raise TypeError("Non-numerical value in coefficients.")
        else:
            raise TypeError("""Attribute 'coeffs' must be either a list, \
tuple, or numpy array. Got: %s""" % (type(value)))

    @property
    def order(self):
        r"""
        Type : int

        The order of the polynomial
        """
        return len(self._coeffs) - 1
```

A Polynomial Object

Indexing – requires *__getitem__* function, which takes the index as a parameter

```
def __getitem__(self, index):
    if isinstance(index, int):
        # Don't need any more error handling - self._coeffs is a numpy
        # array and will raise errors for us
        return self._coeffs[index]
    else:
        raise IndexError("Index must be an integer.")
```

Calling – requires *__call__* function, which takes any number of parameters

- Here it should be a value *x* to evaluate the polynomial at

```
def __call__(self, x):
    result = 0
    for i in range(len(self._coeffs)):
        result += x**i * self._coeffs[i]
    return result
```

A Polynomial Object

Item assignment – requires *__setitem__* function, which takes the index and the value to assign, in that order.

```
def __setitem__(self, index, value):
    if isinstance(index, int):
        if 0 <= index <= self.order:
            if isinstance(value, numbers.Number):
                self._coeffs[index] = value
            else:
                raise TypeError("Must be a numerical value. Got: %s" % (
                    type(value)))
        else:
            raise IndexError("Index out of bounds.")
    else:
        raise IndexError("Must be an integer. Got: %s" % (type(index)))
```

A Polynomial Object

A string representation – requires *__str__* and *__repr__* functions, which do slightly different things.

- *__str__* is called when you type-cast to a string
- *__repr__* is called when you run a line with just the object in *ipython* or a notebook

```
def __repr__(self):
    rep = ""
    for i in range(self.order + 1):
        if i:
            if self._coeffs[i] > 0:
                rep += "+ %.2fx^%d " % (self._coeffs[i], i)
            elif self._coeffs[i] < 0:
                rep += "- %.2fx^%d " % (-self._coeffs[i], i)
            else:
                # don't print if the coefficient is zero
                pass
        else:
            if self._coeffs[i] > 0:
                rep += "%.2f " % (self._coeffs[i])
            elif self._coeffs[i] < 0:
                rep += "-%.2f " % (-self._coeffs[i])
            else:
                # don't print if the coefficient is zero
                pass
    return rep

def __str__(self):
    return self.__repr__()
```

A Polynomial Object

In action – the example has all of these features because of the magic methods implemented in the polynomial class

This is essentially a reimplementation of NumPy's *poly1d* object

```
In [1]: from mypkg.mathlib import polynomial

In [2]: example = polynomial([1, 2, -1, 3, -3])

In [3]: example
Out[3]: 1.00 + 2.00x^1 - 1.00x^2 + 3.00x^3 - 3.00x^4

In [4]: example[0]
Out[4]: 1

In [5]: example[1]
Out[5]: 2

In [6]: example(0)
Out[6]: 1

In [7]: example(3)
Out[7]: -164

In [8]: example[4] = 3

In [9]: example(3)
Out[9]: 322

In [10]: example.coeffs
Out[10]: array([ 1,  2, -1,  3,  3])

In [11]: example.order
Out[11]: 4

In [12]: example
Out[12]: 1.00 + 2.00x^1 - 1.00x^2 + 3.00x^3 + 3.00x^4
```

Emulating Numeric Types

Another application of syntactic sugar

There is also __*rsub*__ for -=, __*rmul*__ for *=, __*rdiv*__ for /=, etc., but unless otherwise specified these lines will call the corresponding function without the *r* in the name.

With Syntactic Sugar	Without Syntactic Sugar
x + y	x.__add__(y)
x += y	x.__radd__(y)
x - y	x.__sub__(y)
x * y	x.__mul__(y)
x / y	x.__div__(y)
x // y	x.__floordiv__(y)
x % y	x.__mod__(y)

Emulating Numeric Types: A Polynomial

Extend the polynomial object to allow addition, subtraction, and equivalence comparison with other polynomials

Some extra useful syntactic sugar elements in doing so:

With Syntactic Sugar	Without Syntactic Sugar
+x	x.__pos__()
-x	x.__neg__()
x == y	x.__eq__(y)
x != y	x.__ne__(y)

Emulating Numeric Types: A Polynomial

Unary +: The same as the original polynomial

Unary -: Each coefficient is the negative of the original

```
def __pos__(self):
    return self

def __neg__(self):
    return polynomial([-i for i in self._coeffs])
```

Magic methods can return anything, hence the need to specifically create a polynomial object. They don't call *__init__* automatically, so in theory you can have them do whatever you want. Why isn't this required for *__pos__*?

Emulating Numeric Types: A Polynomial

Adding polynomials: The coefficients of each power on *x* add

```
def __add__(self, other):
    if isinstance(other, polynomial):
        new_coeffs = (max(other.order, self.order) + 1) * [0.]
        for i in range(len(new_coeffs)):
            if i <= self.order: new_coeffs[i] += self[i]
            if i <= other.order: new_coeffs[i] += other[i]
        return polynomial(new_coeffs)
    else:
        raise TypeError("Must be a polynomial object. Got: %s" % (
            type(other)))
```

Emulating Numeric Types: A Polynomial

Subtracting polynomials: Use what we've already written to add the negative

```
def __sub__(self, other):
    if isinstance(other, polynomial):
        # The same as "return self + -other"
        return self.__add__(other.__neg__())
    else:
        raise TypeError("Must be a polynomial object. Got: %s" % (
            type(other)))
```

Emulating Numeric Types: A Polynomial

Equivalence comparison: If two polynomials have the same coefficients, say that they are equal to one another

```
def __eq__(self, other):
    if isinstance(other, polynomial):
        if self.order == other.order:
            # The numpy array does component-wise comparison here, hence
            # the usage of all
            return all(self.coeffs == other.coeffs)
        else:
            return False
    else:
        return False

def __ne__(self, other):
    return not self.__eq__(other)
```

Note: This __*ne*__ method actually isn't necessary. If you write an __*eq*__ method, the __*ne*__ method takes on this default form. Advice: If you ever override that, you should have a good reason for doing so.

Emulating Numeric Types: A Polynomial

In action – *x* and *y* have all of these features because of the magic methods we implemented

Features like this are also included in NumPy's *poly1d* object

```
In [1]: from mypkg.mathlib import polynomial

In [2]: x = polynomial([1, 2, 3])

In [3]: y = polynomial([1, 3, 4])

In [4]: x + y
Out[4]: 2.00 + 5.00x^1 + 7.00x^2

In [5]: x == y
Out[5]: False

In [6]: x != y
Out[6]: True

In [7]: x - y
Out[7]: - 1.00x^1 - 1.00x^2

In [8]: (x + y)(3)
Out[8]: 80.0

In [9]: (x - y)(1)
Out[9]: -2.0
```

Recall: Lists vs. Arrays

If it wasn't already, it should now be fairly clear what we mean when we say lists and arrays are different objects.

They are instances of different classes with different source code.