Lec 17 - classes + custom transformers

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

Classes

Basic syntax

These are the basic component of Python's object oriented system - we've been using them regularly all over the place and will now look at how they are defined and used.

```
class rect:
  """An object representation of a rectangle"""

  # Attributes
  p1 = (0,0)
  p2 = (1,2)

  # Methods
  def area(self):
    return ((self.p1[0] - self.p2[0]) *
            (self.p1[1] - self.p2[1]))

  def set_p1(self, p1):
    self.p1 = p1

  def set_p2(self, p2):
    self.p2 = p2
```

```
x = rect()
x.area()
```

```
## 2
```

```
x.set_p2((1,1))
x.area()
```

```
## 1
```

```
x.p1
```

```
## (0, 0)
```

```
x.p2
```

```
## (1, 1)
```

```
x.p2 = (0,0)
```

Instantiation (constructors)

When instantiating a class object (e.g. `rect()`) we invoke the `__init__()` method if it is present in the classes' definition.

```
class rect:
  """An object representation of a rectangle"""

  # Constructor
  def __init__(self, p1 = (0,0), p2 = (1,1)):
    self.p1 = p1
    self.p2 = p2

  # Methods
  def area(self):
    return ((self.p1[0] - self.p2[0]) *
            (self.p1[1] - self.p2[1]))

  def set_p1(self, p1):
    self.p1 = p1

  def set_p2(self, p2):
    self.p2 = p2
```

```
x = rect()
x.area()
```

```
## 1
```

```
y = rect((0,0), (3,3))
y.area()
```

```
## 9
```

```
z = rect((-1,-1))
z.p1
```

```
## (-1, -1)
```

```
z.p2
```

```
## (1, 1)
```

Method chaining

We've seen a number of objects (i.e. Pandas DataFrames) that allow for method chaining to construct a pipeline of operations. We can achieve the same by having our class methods return `self`.

```
class rect:
  """An object representation of a rectangle"""

  # Constructor
  def __init__(self, p1 = (0,0), p2 = (1,1)):
    self.p1 = p1
    self.p2 = p2

  # Methods
  def area(self):
    return ((self.p1[0] - self.p2[0]) *
            (self.p1[1] - self.p2[1]))

  def set_p1(self, p1):
    self.p1 = p1
    return self

  def set_p2(self, p2):
```

```
rect().area()
```

```
## 1
```

```
rect().set_p1((-1,-1)).area()
```

```
## 4
```

```
rect().set_p1((-1,-1)).set_p2((2,2)).area()
```

```
## 9
```

Class object string formating

All class objects have a default print method / string conversion method, but the default behavior is not very useful,

```
print(rect())
```

```
## <__main__.rect object at 0x290aa1a60>
```

```
str(rect())
```

```
## '<__main__.rect object at 0x290aa1ca0>'
```

Both of the above are handled by the `__str__()` method which is implicitly created for our class - we can override this,

```
def rect_str(self):
  return f"Rect[{self.p1}, {self.p2}] => area={self.area()}"

rect.__str__ = rect_str
```

```
rect()
```

```
## <__main__.rect object at 0x290a98070>
```

```
print(rect())
```

Class representation

There is another special method which is responsible for the printing of the object (see `rect()` above) called `__repr__()` which is responsible for printing the classes representation. If possible this is meant to be a valid Python expression capable of recreating the object.

```
def rect_repr(self):
  return f"rect({self.p1}, {self.p2})"

rect.__repr__ = rect_repr
```

```
rect()
```

```
## rect((0, 0), (1, 1))
```

```
repr(rect())
```

```
## 'rect((0, 0), (1, 1))'
```

Inheritance

Part of the object oriented system is that classes can inherit from other classes, meaning they gain access to all of their parents attributes and methods. We will not go too in depth on this topic beyond showing the basic functionality.

```
class square(rect):
    pass
```

```
square()
```

```
## rect((0, 0), (1, 1))
```

```
square().area()
```

```
## 1
```

```
square().set_p1((-1,-1)).area()
```

```
## 4
```

Overriding methods

```
class square(rect):
    def __init__(self, p1=(0,0), l=1):
      assert isinstance(l, (float, int)), \
             "l must be a numnber"

      p2 = (p1[0]+l, p1[1]+l)

      self.l  = l
      super().__init__(p1, p2)

    def set_p1(self, p1):
      self.p1 = p1
      self.p2 = (self.p1[0]+self.l, self.p1[1]+se
      return self

    def set_p2(self, p2):
      raise RuntimeError("Squares take l not p2")

    def set_l(self, l):
      assert isinstance(l, (float, int)), \
             "l must be a numnber"

      self.l  = l
      self.p2 = (self.p1[0]+l, self.p1[1]+l)
```

```
square()
```

```
## square((0, 0), 1)
```

```
square().area()
```

```
## 1
```

```
square().set_p1((-1,-1)).area()
```

```
## 1
```

```
square().set_l(2).area()
```

```
## 4
```

```
square((0,0), (1,1))
```

```
## AssertionError: l must be a numnber
```

```
square().set_l((0,0))
```

```
## AssertionError: l must be a numnber
```

Making an object iterable

When using an object with a for loop, python looks for the `__iter__()` method which is expected to return an iterator object (e.g. `iter()` of a list, tuple, etc.).

```
class rect:
  """An object representation of a rectangle"""

  # Constructor
  def __init__(self, p1 = (0,0), p2 = (1,1)):
    self.p1 = p1
    self.p2 = p2

  # Methods
  def area(self):
    return ((self.p1[0] - self.p2[0]) *
            (self.p1[1] - self.p2[1]))

  def __iter__(self):
    return iter( [
      self.p1,
      (self.p1[0], self.p2[1]),
      self.p2,
      (self.p2[0], self.p1[1])
    ] )
```

```
for pt in rect():
  print(pt)
```

```
## (0, 0)
## (0, 1)
## (1, 1)
## (1, 0)
```

Fancier iteration

A class itself can be made iterable by adding a `__next__()` method which is called until a `StopIteration` exception is encountered. In which case, `__iter__()` is still needed but should just `return self`.

```
class rect:
  def __init__(self, p1 = (0,0), p2 = (1,1)):
    self.p1 = p1
    self.p2 = p2
    self.vertices = [self.p1, (self.p1[0], self.p
                     self.p2, (self.p2[0], self.p
    self.index = 0

  # Methods
  def area(self):
    return ((self.p1[0] - self.p2[0]) *
            (self.p1[1] - self.p2[1]))

  def __iter__(self):
    return self

  def __next__(self):
    if self.index == len(self.vertices):
```

```
r = rect()
for pt in r:
  print(pt)
```

```
## (0, 0)
## (0, 1)
## (1, 1)
## (1, 0)
```

```
for pt in r:
  print(pt)
```

```
## (0, 0)
## (0, 1)
## (1, 1)
## (1, 0)
```

Generators

There is a lot of bookkeeping in the implementation above - we can simplify this significantly by using a generator function with `__iter__()`. A generator is a function which using `yield` instead of `return` which allows the function to preserve state between `next()` calls.

```
class rect:
  """An object representation of a rectangle"""

  # Constructor
  def __init__(self, p1 = (0,0), p2 = (1,1)):
    self.p1 = p1
    self.p2 = p2

  # Methods
  def area(self):
    return ((self.p1[0] - self.p2[0]) *
            (self.p1[1] - self.p2[1]))

  def __iter__(self):
    vertices = [ self.p1, (self.p1[0], self.p2[1]
                 self.p2, (self.p2[0], self.p1[1]

    for v in vertices:
```

```
r = rect()

for pt in r:
  print(pt)
```

```
## (0, 0)
## (0, 1)
## (1, 1)
## (1, 0)
```

```
for pt in r:
  print(pt)
```

```
## (0, 0)
## (0, 1)
## (1, 1)
## (1, 0)
```

Class attributes

We can examine all of a classes' methods and attributes using `dir()`,

```
dir(rect)
```

```
## ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattril
```

Where did `p1` and `p2` go?

```
dir(rect())
```

```
## ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattril
```

Custom Transformers

FunctionTransformer

The simplest way to create a new transformer is to use `FunctionTransformer()` from the preprocessing submodule which allows for converting a Python function into a transformer.

```
from sklearn.preprocessing import FunctionTransformer

X = pd.DataFrame({"x1": range(1,6), "x2": range(5, 0, -1)})
```

```
log_transform = FunctionTransformer(np.log)
lt = log_transform.fit(X)
lt.transform(X)
```

```
##          x1        x2
## 0  0.000000  1.609438
## 1  0.693147  1.386294
## 2  1.098612  1.098612
## 3  1.386294  0.693147
## 4  1.609438  0.000000
```

```
lt
```

```
## FunctionTransformer(func=<ufunc 'log'>)
```

```
lt.get_params()
```

```
## {'accept_sparse': False, 'check_inverse': True, 'fun
```

```
dir(lt)
```

```
## ['__class__', '__delattr__', '__dict__', '__dir__',
```

Input types

```
def interact(X, y = None):
  return np.c_[X, X[:,0] * X[:,1]]

X = pd.DataFrame({"x1": range(1,6), "x2": range(5, 0, -1)})
Z = np.array(X)
```

```
FunctionTransformer(interact).fit_transform(X)
```

```
## InvalidIndexError: (slice(None, None, None), 0)
```

```
FunctionTransformer(interact).fit_transform(Z)
```

```
## array([[1, 5, 5],
##        [2, 4, 8],
##        [3, 3, 9],
##        [4, 2, 8],
##        [5, 1, 5]])
```

```
FunctionTransformer(
  interact, validate=True
).fit_transform(X)
```

```
## array([[1, 5, 5],
##        [2, 4, 8],
##        [3, 3, 9],
##        [4, 2, 8],
```

```
FunctionTransformer(
  interact, validate=True
).fit_transform(Z)
```

```
## array([[1, 5, 5],
##        [2, 4, 8],
##        [3, 3, 9],
##        [4, 2, 8],
```

The `validate` argument both checks that `X` is 2d as well as converts it to a np.array

Build your own transformer

For a more full features transformer, it is possible to construct it as a class that inherits from `BaseEstimator` and `TransformerMixin` classes from the base submodule.

```
from sklearn.base import BaseEstimator, Transform

class scaler(BaseEstimator, TransformerMixin):
  def __init__(self, m = 1, b = 0):
    self.m = m
    self.b = b

  def fit(self, X, y=None):
    return self

  def transform(self, X, y=None):
    return X*self.m + self.b
```

```
X = pd.DataFrame({"x1": range(1,6), "x2": range(5

double = scaler(2)
double.fit_transform(X)
```

```
##    x1  x2
## 0   2  10
## 1   4   8
## 2   6   6
## 3   8   4
## 4  10   2
```

```
double.get_params()
```

```
## {'b': 0, 'm': 2}
```

```
double.set_params(b=-3).fit_transform(X)
```

```
##    x1  x2
```

What else do we get?

```
print(
  pd.DataFrame(np.array(dir(double)).reshape(-1,4)).to_string(index=False, header=False, col_space=20)
)
```

```
##            __class__          __delattr__            __dict__             __dir__
##              __doc__              __eq__          __format__              __ge__
##     __getattribute__         __getstate__              __gt__            __hash__
##             __init__    __init_subclass__              __le__              __lt__
##           __module__              __ne__              __new__          __reduce__
##        __reduce_ex__             __repr__         __setattr__         __setstate__
##           __sizeof__              __str__     __subclasshook__          __weakref__
## _check_feature_names    _check_n_features     _get_param_names            _get_tags
##           _more_tags          _repr_html_     _repr_html_inner    _repr_mimebundle_
##       _validate_data                    b                  fit        fit_transform
##           get_params                    m          set_params            transform
```

Demo - Interaction Transformer

Useful methods

We employed a couple of special methods that are worth mentioning in a little more detail.

- `_validate_data()` & `_check_feature_names()` are methods that are inherited from `BaseEstimator` they are responsible for setting and checking the `n_features_in_` and the `feature_names_in_` attributes respectively.
- In general one or both is run during `fit()` with `reset=True` in which case the respective attribute will be set.
- Later, in `tranform()` one or both will again be called with `reset=False` and the properties of `X` will be checked against the values in the attribute.
- These are worth using as they promote an interface consistent with sklearn and also provide convenient error checking with useful warning and error messages.

check_is_fitted()

This is another useful helper function from `sklearn.utils` - it is fairly simplistic in that it checks for the existence of the specified attribute. If not attribute is given then it checks for any attributes ending in _ that do not begin with __.

Again this is useful for providing a consistent interface and useful error / warning messages.