OOP introduction (2/2)

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

alberto.manfreda@pi.infn.it

INFN–Pisa

Compiled on September 29, 2021

Introducing the Vector2d class

- Suppose we want to create a class for managing 2D vectors
- That's just for learning: there are already plenty of libraries for doing array operations - like numpy!
- Anyway let's start coding some useful methods for it

Introducing the Vector2d class

Naive version

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_naive.py

```python
import math

class Vector2d:
    """ Class representing a Vector2d. We use float() to make sure of storing
    the coordinates in the correct format """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def module(self):
        return math.sqrt(self.x**2 + self.y**2)

    def info(self):
        print ('Vector2d({}, {})'.format(self.x, self.y))

    def add(self, other):
        return Vector2d(self.x + other.x, self.y + other.y)

v = Vector2d(3., -1.)
v.info()
print(v.module())
z = Vector2d(1., 1.5)
t = v.add(z)
t.info()

[Output]
Vector2d(3.0, -1.0)
3.1622776601683795
Vector2d(4.0, 0.5)
```

The vector problem

- This kind of works but. . . ... isn't that ugly?
- Look at the lines *v.info()* or *v.module()*. It would be far more readible to just do *print(v)* and *abs(v)*
- And what about $t = v.add(z)$? Why not $t = v + z$?
- In Python there is a tool that allows you to do just that: special methods
- Last lesson we saw that special methods (or dunder methods or magic methods) are methods like *__init__* and got a special treatment by the Python interpeter
- There are a few tens of special methods in Python. Let's see how they work

Vector2d

A first look at special methods

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_2.py

```python
import math

class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __abs__(self):
        # Special method!
        return math.sqrt(self.x**2 + self.y**2)

v = Vector2d(3., -1.)
# The Python interpeter automatically replace abs(v) with Vector2d.__abs__(v)
print(abs(v))

[Output]
3.1622776601683795
```

More on special methods

- And what about *print()*?
- There are actually two special methods used for that: *__str__* and *__repr__*
- *__str__* is meant to return a concise string for the user; it is called with *str()*
- *__repr__* is meant to return a richer output for debug. It is called with *repr()*
- *print()* automatically tries to get a string out of the object using *__str__*
- If there isn't one, it searches for *__repr__*. A defealut *__repr__* is automatically generated for you, if you haven't defined one

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_printable.py

```
class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __repr__(self):
        # We don't want to hard-code the class name, so we dynamically get it
        class_name = type(self).__name__
        return ('{}({}, {})'.format(class_name, self.x, self.y))

    def __str__(self):
        """ We convert the coordinates to a tuple so that we can reuse the
        __str__ method of tuples, which already provides a nice formatting.
        Notice the two parenthesis: this line is equivalent to:
        temp_tuple = (self.x, self.y)
        return str(temp_tuple)
        """
        return str((self.x, self.y))

v = Vector2d(3., -1.)
print(v) # Is the same as print(str(v))
print(repr(v))
print('I got {} with __str__ and {!r} with __repr__'.format(v, v))

[Output]
(3.0, -1.0)
Vector2d(3.0, -1.0)
I got (3.0, -1.0) with __str__ and Vector2d(3.0, -1.0) with __repr__
```

Vector2d

Mathematical operations

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_math.py

```
class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __add__(self, other):
        return Vector2d(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar):
        return Vector2d(scalar * self.x, scalar * self.y)

    def __rmul__(self, scalar):
        # Right multiplication - because a * Vector is different from Vector * a
        return self * scalar # We just call __mul__, no code duplication!

    def __str__(self):
        # We keep this to show the results nicely
        return str((self.x, self.y))

v, z = Vector2d(3., -1.), Vector2d(-5., 1.)
print(v+z)
print(3 * v)
print(z * 5)

[Output]
(-2.0, 0.0)
(9.0, -3.0)
(-25.0, 5.0)
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_inplace.py

```python
class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __imul__(self, other):
        self.x *= other.x
        self.y *= other.y
        return self

    def __str__(self):
        return str((self.x, self.y))

v = Vector2d(3., -1.)
z = Vector2d(-5., 1.)
v += z
print(v)
v *= z
print(v)

[Output]
(-2.0, 0.0)
(10.0, 0.0)
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_comparable.py

```python
import math

class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __abs__(self):
        # We need this for __eq__
        return math.sqrt(self.x**2 + self.y**2)

    def __eq__(self, other):
        # Implement the '==' operator
        return ((self.x, self.y) == (other.x, other.y))

    def __ge__(self, other):
        # Implement the '>=' operator
        return abs(self) >= abs(other)

    def __lt__(self, other):
        # Implement the '<' operator
        return abs(self) < abs(other)

    def __repr__(self):
        # We define __repr__ for showing the results nicely
        class_name = type(self).__name__
        return ('{}({}, {})'.format(class_name, self.x, self.y))
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/test_vector2d_comparable.py

```
from vector2d_comparable import Vector2d

v, z = Vector2d(3., -1.), Vector2d(3., 1.)
print(v >= z, v == z, v < z)
# This works even if we don't define the __gt__ method explicitly
print(v > z)

vector_list = [Vector2d(3., -1.), Vector2d(-5., 1.), Vector2d(3., 0.)]
print(vector_list)
# Tho make the following line work we need to implement either __ge__ and __lt__
# or __gt__ and __le__ (we need a complementary pair of operator)
vector_list.sort()
print(vector_list)
# Note: we got the full power of timsort for free! Nice :)

[Output]
True False False
False
[Vector2d(3.0, -1.0), Vector2d(-5.0, 1.0), Vector2d(3.0, 0.0)]
[Vector2d(3.0, 0.0), Vector2d(3.0, -1.0), Vector2d(-5.0, 1.0)]
```

An hashable Vector2d

- Ok now let's try to make our vector2d *hashable*
- Hashable objects can be put in *sets* and used as keys for dictionaries
- To make an object hashable we need to fullfill 3 requirements:
  - It has to be immutable - otherwise you may not retrieve the correct hash
  - It needs to implement a *__eq__* function, so one can compare objects of this class
  - It needs a (reasonable) *__hash__* function
- Rules for a good hash function:
  - Must return the same value for objects that compare as equal
  - Should rarely return the same value for different objects
  - Should sample the result space uniformly

Vector2d

Hashable version

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_hashable.py

```python
class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        """" We tell the user that x and y are private"""
        self._x = float(x)
        self._y = float(y)

    @property
    def x(self):
        """ Provides read only access to x - since there is no setter"""
        return self._x

    @property
    def y(self):
        """ Provides read only access to y - since there is no setter"""
        return self._y

    def __eq__(self, other):
        return ((self.x, self.y) == (other.x, other.y))

    def __hash__(self):
        """ As hash value we provide the logical XOR of the hash of the two
        coordinates """
        return hash(self.x) ^ hash(self.y)

    def __repr__(self):
        # Again we neeed __repr__ to display the results nicely
        class_name = type(self).__name__
        return ('{}({}, {})'.format(class_name, self.x, self.y))
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_hashable_test.py

```python
from vector2d_hashable import Vector2d

v, t, z = Vector2d(3., -1.), Vector2d(-5., 1.), Vector2d(3., -1.)
# Check the equality
print(v == t, v == z, t == z)
# Check the hash: v and z are equal, so they will have the same hash
print(hash(v), hash(t), hash(z))
# v and t have different hash, so they can be in the same set
print({v, t})
# v and z have the same hash -- only one will be stored in the set!
print({v, z})

[Output]
False True False
-3 -6 -3
Vector2d(-5.0, 1.0), Vector2d(3.0, -1.0)
Vector2d(3.0, -1.0)
```

A n-elements vector

- 2d array are boring. . . why not a N-d array?
- Of course we cannot store the components explicitly like before
- We need a contaner for that and we will use *array* from the array library
- Question for you: why not a list or a tuple?
- *array* uses a typecode (a single character) for picking the type. 'd' is the typecode for float numbers in double precision.

Vector

A n elements vector

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector.py

```python
import math
from array import array

class Vector:
    """ Classs representing a multidimensional vector"""
    typecode = 'd'  #We use a class attribute to save the code required for array

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __repr__(self):
        """ Calling str() of an array produces a string like
        array('d', [1., 2., 3., ...]). We remove everything outside the
        square parenthesis and add our class name at the beginning."""
        components = str(self._components)
        components =  components[components.find('['): -1]
        class_name = type(self).__name__
        return '{}({})'.format(class_name, components)

    def __str__(self):
         return str(tuple(self._components)) # Using str() of tuples as before

v = Vector([5., 3., -1, 8.])
print(v)
print(repr(v))

[Output]
(5.0, 3.0, -1.0, 8.0)
Vector([5.0, 3.0, -1.0, 8.0])
```

- Now that we have an arbitrary number of components, we cannot access them like vector.x, vector.y, . . . anymore
- What we want is a syntax similar to that of lists: vector(0), vector(1) ans so on
- There are two magic methods for that: *__getitem__* for access and *__setitem__* for modifying
- While we are at it, we also implement the *__len__* method, which allows us to call *len(vector)*

Vector

List-style access

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector_random_access.py

```python
import math
from array import array

class Vector:
    """ Classs representing a multidimensional vector"""
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __getitem__(self, index):
        """ That's super easy, as we get to reuse the __getitem__ of array!"""
        return self._components[index]

    def __setitem__(self, index, new_value):
        """ Same as __getitem__, we just delegate to the __setitem__ of array"""
        self._components[index] = new_value

    def __len__(self):
        """ Did I just write that we like to delegate? """
        return len(self._components)

    def __repr__(self):
        components = str(self._components)
        components =  components[components.find('['): -1]
        class_name = type(self).__name__
        return '{}({})'.format(class_name, components)
```

Vector

List-style access

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/test_vector_random_access.py

```
from vector_random_access import Vector

v = Vector([5., 3., -1, 8.])

print(len(v))

print(v[0], v[1])

v[1] = 10.
print(v)

print (v[9]) # This will generate an error!

[Output]
4
5.0 3.0
Vector([5.0, 10.0, -1.0, 8.0])
Traceback (most recent call last):
  File "snippets/test_vector_random_access.py", line 12, in <module>
    print (v[9]) # This will generate an error!
  File "/data/work/teaching/cmepda/slides/latex/snippets/vector_random_access.py", line 13,
    return self._components[index]
IndexError: array index out of range
```

An iterable vector

- Now our vector behave a bit like a native python list
- However a list has a very powerful feature we miss: it's iterable
- An *iterable* in Python is something that has a *__iter__* method, which returns an iterator
- Technically, an iterator is an object that implements the *__next__* special method, which is used to retrieve elements one at a time
- We will not dicuss iterator any further here: instead, we will just exploit composition and borrow the *__iter__* method from the underlying array

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector_iterable.py

```python
import math
from array import array

class Vector:
    """ Classs representing a multidimensional vector"""
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        """ We don't need to code anything... an array is already iterable!"""
        return iter(self._components)

v = Vector([5.1, 3.7, -25.])
for component in v:
    print(component)

[Output]
5.1
3.7
-25.0
```

Duck typing

"If it looks like a duck and quacks like a duck, it must be a duck."

Duck typing

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/duck_typing.py

```python
class Duck:
    """ This is a duck - it quacks"""

    def quack(self):
        print('Quack!')

class Goose:
    """ This is a goose - it quacks too"""

    def quack(self):
        print('Quack!')

class Penguin:
    """ This is a penguin -- He doesn't quack!"""
    pass

birds = [Duck(), Goose(), Penguin()]

for bird in birds:
    bird.quack()

[Output]
Quack!
Quack!
Traceback (most recent call last):
  File "snippets/duck_typing.py", line 20, in <module>
    bird.quack()
AttributeError: 'Penguin' object has no attribute 'quack'
```

Polymorphism

- Reuse the same code for different things
- In statically typed languages this is tipically done with inheritance, e.g. we make Duck and Goose inherits from a base class QuackingBird() or something like that
- Python is dynamical, so we can use duck typing for that. We just need to implment the quack() method for both Ducks() and Goose() and we are done
- In other words we obtain polymorphism just by satsisfying the required interface (in this case the quack() function)

The power of iterables

- Having an iterable Vector (thanks to the *__iter__* magic method) makes all the difference in the world
- There are *a lot* of builtin and library functions in python accepting a generic iterable as input:
  - *sum*: Sum all the elements
  - *max/min*: Return the maximum/minimum
  - *enumerate*: Iterate with automatic counting of iterations
  - *map*: Apply a function to the elements one by one
  - *filter*: Iterate only on the elements passing a given condition
  - *zip*: Iterate over pairs of elements (requires two iterables)
- Countless others can be found in the *itertools* library
  - *islice*: Slice the loop with start, stop and step
  - *takewhile*: Stop looping when a condition becomes false
  - *chain*: Loop through many sequences one after another
  - *cycle*: Loop over the sequence repeatedly, indefinitely
  - *permutations*: Get all the permutations of a given length
  - And so on...
- With duck typing we can now use any of that for our Vector class – isn't that cool?

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector_iterable_test.py

```python
from vector_iterable import Vector
from itertools import permutations

vec = Vector([1., 2., 4.])

# Select only the elements passing a given condition
def filter_function(x):
    return x > 3.

filtered = [x for x in (filter(filter_function, vec))] # list comprehension
print(filtered)

# Print all the permutations of two elements
for p in permutations(vec, 2):
    print(p)

[Output]
[4.0]
(1.0, 2.0)
(1.0, 4.0)
(2.0, 1.0)
(2.0, 4.0)
(4.0, 1.0)
(4.0, 2.0)
```

A vector that behaves like a duck

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector_ducked.py

```python
import math
from array import array

class Vector:
    """ Classs representing a multidimensional vector"""
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __len__(self):
        return len(self._components)

    def __iter__(self):
        return iter(self._components)

    def __str__(self):
        return str(tuple(self)) # tuple() accept an iterable

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self)) # generator expression

    def __add__(self, other):
        """ zip returns a sequence of pairs from two iterables"""
        return Vector([x + y for x, y in zip(self, other)])

    def __eq__(self, other):
        return (len(self) == len(other)) and \
               (all(a == b for a, b in zip(self, other)))    # Efficient test!
```

Let's test it

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/test_vector_ducked.py

```
from vector_ducked import Vector

v = Vector([1., 2., 3.])
t = Vector([1., 2., 3., 4.])
z = Vector([1., 2., 5.])
u = Vector([1., 2., 3.])

print(v)
print(abs(v))
print(v == t, v == z, v == u)
print(v+z)
print(v+t) # Note the result: this is due to the behaviour of zip()!

[Output]
(1.0, 2.0, 3.0)
3.7416573867739413
False False True
(2.0, 4.0, 8.0)
(2.0, 4.0, 6.0)
```

Fucntion are classes

- Remember that in the past lesson I told you that functions are objects of the ‘function’ class.
- How are they implemented?
- With a special method - of course: __*call*__
- Every object implementing a __*call*__ method is called callable
- A vector is not a good example for implementing it, let’s try something different!

A simple call counter

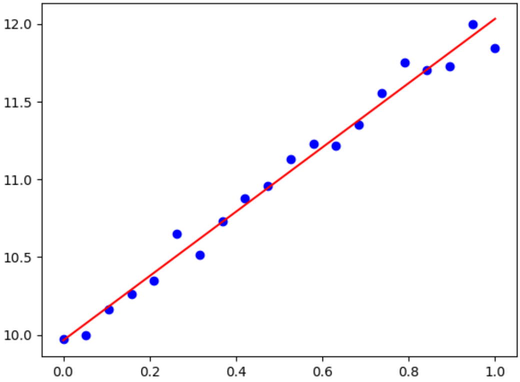
https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/callable.py

```python
class CallCounter:

    """Wrap a generic function and count the number of times it is called"""

    def __init__(self, func):
        # We accept as input a function and store it (privately)
        self._func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        """ This is the method doing the trick. We use *args and **kwargs to
        pass all possible arguments to the function that we are wrapping"""
        # We increment the counter
        self.num_calls += 1
        # And here we just return whatever the wrapped function returns
        return self._func(*args, **kwargs)

    def reset(self):
        self.num_calls = 0
```

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/test_callable.py

```python
import numpy
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt
from callable import CallCounter

def line(x, m, q):
    return m * x + q

# Generate the datasets: a straight line + gaussian fluctuations
x = numpy.linspace(0., 1., 20)
y = line(x, 2., 10.) + numpy.random.normal(0, 0.1, len(x))

# Fit
counting_func = CallCounter(line)
popt, pcov = curve_fit(counting_func, x, y, p0=[-1., -100.]) # p0 is mandatory here
print('Fitted with {} function calls'.format(counting_func.num_calls))

# Show the results
m, q = popt
plt.figure('fit with custom callable')
plt.plot(x, y, 'bo')
plt.plot(x, line(x, m, q), 'r-')
plt.show()

[Output]
Fitted with 9 function calls
```

Fit hacking



- ▷ The fit works as usual

Summary

- Special methods can beused to greatly enhance the readibility of the code
- There are tens of special methods in python, covering logical operations, mathematical operations, array-style access, iterations, formatting and many other things. . .
- Implementing the required interface in your classes you will be able to reuse a lot of code written for the standard containers thanks to duck typing, which is the pythonic way to polymorphism