OOP introduction (2/2)

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

alberto.manfreda@pi.infn.it

INFN–Pisa

▷ Suppose we want to create a class for managing 2D vectors[1]

▷ 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

---

[1]The content of this lesson is vastly based on the the book '*Fluent Python*' by Luciano Ramalho

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)
```

▷ 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

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
```

▷ 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

```python
 1   class Vector2d:
 2       """ Class representing a Vector2d """
 3       def __init__(self, x, y):
 4           self.x = float(x)
 5           self.y = float(y)
 6
 7       def __repr__(self):
 8           # We don't want to hard-code the class name, so we dynamically get it
 9           class_name = type(self).__name__
10           return ('{}({}, {})'.format(class_name, self.x, self.y))
11
12       def __str__(self):
13           """ We convert the coordinates to a tuple so that we can reuse the
14           __str__ method of tuples, which already provides a nice formatting.
15           Notice the two parenthesis: this line is equivalent to:
16           temp_tuple = (self.x, self.y)
17           return str(temp_tuple)
18           """
19           return str((self.x, self.y))
20
21   v = Vector2d(3., -1.)
22   print(v) # Is the same as print(str(v))
23   print(repr(v))
24   print('I got {} with __str__ and {!r} with __repr__'.format(v, v))
25
26   [Output]
27   (3.0, -1.0)
28   Vector2d(3.0, -1.0)
29   I got (3.0, -1.0) with __str__ and Vector2d(3.0, -1.0) with __repr__
```

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

```python
1   class Vector2d:
2       """ Class representing a Vector2d """
3       def __init__(self, x, y):
4           self.x = float(x)
5           self.y = float(y)
6
7       def __add__(self, other):
8           return Vector2d(self.x + other.x, self.y + other.y)
9
10      def __mul__(self, scalar):
11          return Vector2d(scalar * self.x, scalar * self.y)
12
13      def __rmul__(self, scalar):
14          # Right multiplication - because a * Vector is different from Vector * a
15          return self * scalar # We just call __mul__, no code duplication!
16
17      def __str__(self):
18          # We keep this to show the results nicely
19          return str((self.x, self.y))
20
21  v, z = Vector2d(3., -1.), Vector2d(-5., 1.)
22  print(v+z)
23  print(3 * v)
24  print(z * 5)
25
26  [Output]
27  (-2.0, 0.0)
28  (9.0, -3.0)
29  (-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

```python
 1  from vector2d_comparable import Vector2d
 2
 3  v, z = Vector2d(3., -1.), Vector2d(3., 1.)
 4  print(v >= z, v == z, v < z)
 5  # This works even if we don't define the __gt__ method explicitly
 6  print(v > z)
 7
 8  vector_list = [Vector2d(3., -1.), Vector2d(-5., 1.), Vector2d(3., 0.)]
 9  print(vector_list)
10  # Tho make the following line work we need to implement either __ge__ and __lt__
11  # or __gt__ and __le__ (we need a complementary pair of operator)
12  vector_list.sort()
13  print(vector_list)
14  # Note: we got the full power of timsort for free! Nice :)
15
16  [Output]
17  True False False
18  False
19  [Vector2d(3.0, -1.0), Vector2d(-5.0, 1.0), Vector2d(3.0, 0.0)]
20  [Vector2d(3.0, 0.0), Vector2d(3.0, -1.0), Vector2d(-5.0, 1.0)]
```

▷ 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

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))
```

Vector2d
Hashable version

```python
1   from vector2d_hashable import Vector2d
2
3   v, t, z = Vector2d(3., -1.), Vector2d(-5., 1.), Vector2d(3., -1.)
4   # Check the equality
5   print(v == t, v == z, t == z)
6   # Check the hash: v and z are equal, so they will have the same hash
7   print(hash(v), hash(t), hash(z))
8   # v and t have different hash, so they can be in the same set
9   print({v, t})
10  # v and z have the same hash -- only one will be stored in the set!
11  print({v, z})
12
13  [Output]
14  False True False
15  -3 -6 -3
16  Vector2d(-5.0, 1.0), Vector2d(3.0, -1.0)
17  Vector2d(3.0, -1.0)
```

▷ 2d array are boring. . . why not a N-d array?

▷ Of course we cannot store the components explicitly like before

▷ We need a container for that and we will use *array* from the array library

▷ This is an example of composition

▷ Question for you: why not a list or a tuple?

▷ Note: *array* uses a typecode (a single character) for picking the type. 'd' is the typecode for float numbers in double precision.

```
     https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector.py
 1  import math
 2  from array import array
 3
 4  class Vector:
 5      """ Classs representing a multidimensional vector"""
 6      typecode = 'd' #We use a class attribute to save the code required for array
 7
 8      def __init__(self, components):
 9          self._components = array(self.typecode, components)
10
11      def __repr__(self):
12          """ Calling str() of an array produces a string like
13          array('d', [1., 2., 3., ...]). We remove everything outside the
14          square parenthesis and add our class name at the beginning."""
15          components = str(self._components)
16          components =  components[components.find('['): -1]
17          class_name = type(self).__name__
18          return '{}({})'.format(class_name, components)
19
20      def __str__(self):
21          return str(tuple(self._components)) # Using str() of tuples as before
22
23  v = Vector([5., 3., -1, 8.])
24  print(v)
25  print(repr(v))
26
27  [Output]
28  (5.0, 3.0, -1.0, 8.0)
29  Vector([5.0, 3.0, -1.0, 8.0])
```

A n-elements vector

List-style access

- ▷ 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)*

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

```python
 1  import math
 2  from array import array
 3
 4  class Vector:
 5      """ Classs representing a multidimensional vector"""
 6      typecode = 'd'
 7
 8      def __init__(self, components):
 9          self._components = array(self.typecode, components)
10
11      def __getitem__(self, index):
12          """ That's super easy, as we get to reuse the __getitem__ of array!"""
13          return self._components[index]
14
15      def __setitem__(self, index, new_value):
16          """ Same as __getitem__, we just delegate to the __setitem__ of array"""
17          self._components[index] = new_value
18
19      def __len__(self):
20          """ Did I just write that we like to delegate? """
21          return len(self._components)
22
23      def __repr__(self):
24          components = str(self._components)
25          components = components[components.find('['): -1]
26          class_name = type(self).__name__
27          return '{}({})'.format(class_name, components)
```

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

```
1   from vector_random_access import Vector
2
3   v = Vector([5., 3., -1, 8.])
4
5   print(len(v))
6
7   print(v[0], v[1])
8
9   v[1] = 10.
10  print(v)
11
12  print (v[9]) # This will generate an error!
13
14  [Output]
15  4
16  5.0 3.0
17  Vector([5.0, 10.0, -1.0, 8.0])
18  Traceback (most recent call last):
19    File "snippets/test_vector_random_access.py", line 12, in <module>
20      print (v[9]) # This will generate an error!
21    File "/data/work/teaching/cmepda/slides/latex/snippets/vector_random_access.py", line 13,
22      return self._components[index]
23  IndexError: array index out of range
```

▷ Now our vector behaves 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 discuss iterators 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
```

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

```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'
```

▷ Reuse the same code for different things

▷ In statically typed languages this is typically 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 implement the quack() method for both Ducks() and Goose() and we are done

▷ In other words we obtain polymorphism just by satisfying 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)
```

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.hypot(*self._components)

    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!
```

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

```python
1   from vector_ducked import Vector
2
3   v = Vector([1., 2., 3.])
4   t = Vector([1., 2., 3., 4.])
5   z = Vector([1., 2., 5.])
6   u = Vector([1., 2., 3.])
7
8   print(v)
9   print(abs(v))
10  print(sum(v))
11  print(v == t, v == z, v == u)
12  print(v+z)
13  print(v+t) # Note the result: this is due to the behaviour of zip()!
14
15  [Output]
16  (1.0, 2.0, 3.0)
17  3.741657386773941
18  6.0
19  False False True
20  (2.0, 4.0, 8.0)
21  (2.0, 4.0, 6.0)
```

▷ 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

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

```python
class Line:
    """Class representing a straight line"""
    def __init__(self, slope=1., intercept=0.):
        self.slope = slope
        self.intercept = intercept

    def __call__(self, x):
        return self.slope * x + self.intercept

    def __str__(self):
        return 'y = {} x + {}'.format(self.slope, self.intercept)

    def __repr__(self):
        return 'Slope = {}, Intercept = {}'.format(self.slope, self.intercept)

line = Line(slope=-2., intercept=1.)
print(line)
print(repr(line))
print(line(2.))

[Output]
y = -2.0 x + 1.0
Slope = -2.0, Intercept = 1.0
-3.0
```

Create a call counter

```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
```
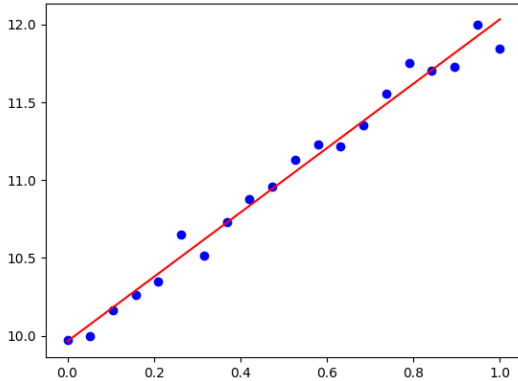
▷ The fit works as usual

▷ Special methods can be used to greatly enhance the readability 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