

OOP introduction (2/2)

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

alberto.manfreda@pi.infn.it

INFN-Pisa

Compiled on October 3, 2021



Introducing the Vector2d class

- ▷ Suppose we want to create a class for managing 2D vectors¹
- ▷ 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

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



Introducing the Vector2d class

Naive version

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

```
1  import math
2
3  class Vector2d:
4      """ Class representing a Vector2d. We use float() to make sure of storing
5          the coordinates in the correct format """
6      def __init__(self, x, y):
7          self.x = float(x)
8          self.y = float(y)
9
10     def module(self):
11         return math.sqrt(self.x**2 + self.y**2)
12
13     def info(self):
14         print ('Vector2d({}, {})'.format(self.x, self.y))
15
16     def add(self, other):
17         return Vector2d(self.x + other.x, self.y + other.y)
18
19 v = Vector2d(3., -1.)
20 v.info()
21 print (v.module())
22 z = Vector2d(1., 1.5)
23 t = v.add(z)
24 t.info()
25
26 [Output]
27 Vector2d(3.0, -1.0)
28 3.1622776601683795
29 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 readable 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 interpreter
- ▷ 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

```
1  import math
2
3  class Vector2d:
4      """ Class representing a Vector2d """
5      def __init__(self, x, y):
6          self.x = float(x)
7          self.y = float(y)
8
9      def __abs__(self):
10         # Special method!
11         return math.sqrt(self.x**2 + self.y**2)
12
13  v = Vector2d(3., -1.)
14  # The Python interpreter automatically replace abs(v) with Vector2d.__abs__(v)
15  print(abs(v))
16
17  [Output]
18  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 default `__repr__` is automatically generated for you, if you haven't defined one



Vector2d

`__str__` and `__repr__`

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

```
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__
```



Vector2d

Mathematical operations

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

```
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)
```




Vector2d

In-place operations

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

```
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 __iadd__(self, other):
8          self.x += other.x
9          self.y += other.y
10         return self
11
12     def __imul__(self, other):
13         self.x *= other.x
14         self.y *= other.y
15         return self
16
17     def __str__(self):
18         return str((self.x, self.y))
19
20 v = Vector2d(3., -1.)
21 z = Vector2d(-5., 1.)
22 v += z
23 print(v)
24 v *= z
25 print(v)
26
27 [Output]
28 (-2.0, 0.0)
29 (10.0, 0.0)
```



Vector2d

Comparisons

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

```
1  import math
2
3  class Vector2d:
4      """ Class representing a Vector2d """
5      def __init__(self, x, y):
6          self.x = float(x)
7          self.y = float(y)
8
9      def __abs__(self):
10         # We need this for __eq__
11         return math.sqrt(self.x**2 + self.y**2)
12
13     def __eq__(self, other):
14         # Implement the '==' operator
15         return (self.x, self.y) == (other.x, other.y)
16
17     def __ge__(self, other):
18         # Implement the '>=' operator
19         return abs(self) >= abs(other)
20
21     def __lt__(self, other):
22         # Implement the '<' operator
23         return abs(self) < abs(other)
24
25     def __repr__(self):
26         # We define __repr__ for showing the results nicely
27         class_name = type(self).__name__
28         return ('{}({}, {})'.format(class_name, self.x, self.y))
```

Vector2d

Comparisons

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

```
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 # To 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)]
```



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 fulfill 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

```
1  class Vector2d:
2      """ Class representing a Vector2d """
3      def __init__(self, x, y):
4          """ We tell the user that x and y are private """
5          self._x = float(x)
6          self._y = float(y)
7
8      @property
9      def x(self):
10         """ Provides read only access to x - since there is no setter """
11         return self._x
12
13     @property
14     def y(self):
15         """ Provides read only access to y - since there is no setter """
16         return self._y
17
18     def __eq__(self, other):
19         return ((self.x, self.y) == (other.x, other.y))
20
21     def __hash__(self):
22         """ As hash value we provide the logical XOR of the hash of the two
23             coordinates """
24         return hash(self.x) ^ hash(self.y)
25
26     def __repr__(self):
27         # Again we need __repr__ to display the results nicely
28         class_name = type(self).__name__
29         return ('{}({},{})'.format(class_name, self.x, self.y))
```



Vector2d

Hashable version

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

```
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)
```



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 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.



Vector

A n elements vector

<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)` and 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

```
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)
```



Vector

List-style access

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
```



An iterable vector

- ▷ 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 dicuss iterators any further here: instead, we will just exploit composition and borrow the `__iter__` method from the underlying array



Vector iterable

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

```
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 __iter__(self):
12          """ We don't need to code anything... an array is already iterable!"""
13          return iter(self._components)
14
15  v = Vector([5.1, 3.7, -25.])
16  for component in v:
17      print(component)
18
19  [Output]
20  5.1
21  3.7
22  -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

```
1 class Duck:
2     """ This is a duck - it quacks"""
3
4     def quack(self):
5         print('Quack!')
6
7 class Goose:
8     """ This is a goose - it quacks too"""
9
10    def quack(self):
11        print('Quack!')
12
13 class Penguin:
14     """ This is a penguin -- He doesn't quack!"""
15     pass
16
17 birds = [Duck(), Goose(), Penguin()]
18
19 for bird in birds:
20     bird.quack()
21
22 [Output]
23 Quack!
24 Quack!
25 Traceback (most recent call last):
26   File "snippets/duck_typing.py", line 20, in <module>
27     bird.quack()
28 AttributeError: 'Penguin' object has no attribute 'quack'
```



Polymorphism

- ▷ 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 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?



The power of iterables

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

```
1  from vector_iterable import Vector
2  from itertools import permutations
3
4  vec = Vector([1., 2., 4.])
5
6  # Select only the elements passing a given condition
7  def filter_function(x):
8      return x > 3.
9
10 filtered = [x for x in filter(filter_function, vec)] # list comprehension
11 print(filtered)
12
13 # Print all the permutations of two elements
14 for p in permutations(vec, 2):
15     print(p)
16
17 [Output]
18 [4.0]
19 (1.0, 2.0)
20 (1.0, 4.0)
21 (2.0, 1.0)
22 (2.0, 4.0)
23 (4.0, 1.0)
24 (4.0, 2.0)
```



A vector that behaves like a duck

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

```
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 __len__(self):
12         return len(self._components)
13
14     def __iter__(self):
15         return iter(self._components)
16
17     def __str__(self):
18         return str(tuple(self)) # tuple() accept an iterable
19
20     def __abs__(self):
21         return math.hypot(*self._components)
22
23     def __add__(self, other):
24         """ zip returns a sequence of pairs from two iterables """
25         return Vector([x + y for x, y in zip(self, other)])
26
27     def __eq__(self, other):
28         return (len(self) == len(other)) and \
29             (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

```
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)
```



Function 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 simple callable for a straight line

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

```
1 class Line:
2     """Class representing a straight line"""
3     def __init__(self, slope=1., intercept=0.):
4         self.slope = slope
5         self.intercept = intercept
6
7     def __call__(self, x):
8         return self.slope * x + self.intercept
9
10    def __str__(self):
11        return 'y = {} x + {}'.format(self.slope, self.intercept)
12
13    def __repr__(self):
14        return 'Slope = {}, Intercept = {}'.format(self.slope, self.intercept)
15
16    line = Line(slope=-2., intercept=1.)
17    print(line)
18    print(repr(line))
19    print(line(2.))
20
21    [Output]
22    y = -2.0 x + 1.0
23    Slope = -2.0, Intercept = 1.0
24    -3.0
```

Create a call counter

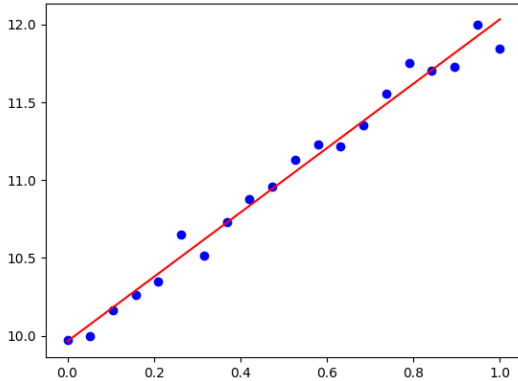
<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/callable.py>

```
1  class CallCounter:
2
3      """Wrap a generic function and count the number of times it is called"""
4
5      def __init__(self, func):
6          # We accept as input a function and store it (privately)
7          self._func = func
8          self.num_calls = 0
9
10     def __call__(self, *args, **kwargs):
11         """ This is the method doing the trick. We use *args and **kwargs to
12         pass all possible arguments to the function that we are wrapping"""
13         # We increment the counter
14         self.num_calls += 1
15         # And here we just return whatever the wrapped function returns
16         return self._func(*args, **kwargs)
17
18     def reset(self):
19         self.num_calls = 0
```

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

```
1 import numpy
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4 from callable import CallCounter
5
6 def line(x, m, q):
7     return m * x + q
8
9 # Generate the datasets: a straight line + gaussian fluctuations
10 x = numpy.linspace(0., 1., 20)
11 y = line(x, 2., 10.) + numpy.random.normal(0, 0.1, len(x))
12
13 # Fit
14 counting_func = CallCounter(line)
15 popt, pcov = curve_fit(counting_func, x, y, p0=[-1., -100.]) # p0 is mandatory here
16 print('Fitted with {} function calls'.format(counting_func.num_calls))
17
18 # Show the results
19 m, q = popt
20 plt.figure('fit with custom callable')
21 plt.plot(x, y, 'bo')
22 plt.plot(x, line(x, m, q), 'r-')
23 plt.show()
24
25 [Output]
26 Fitted with 9 function calls
```


Fit hacking



▷ The fit works as usual



Summary

- ▷ 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