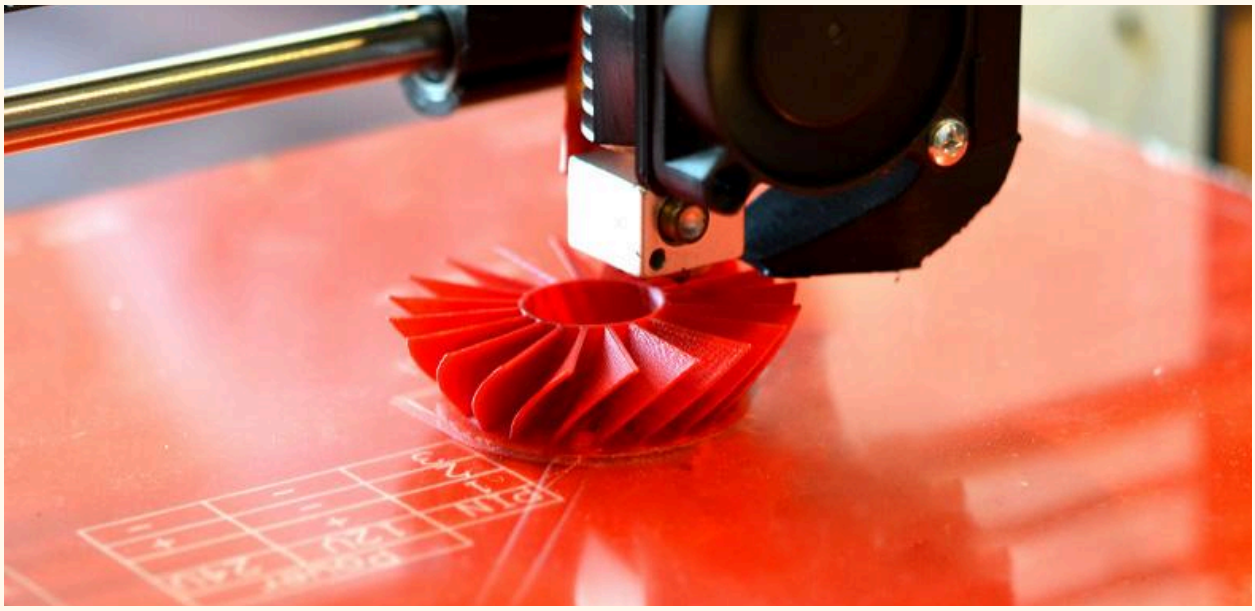
Object-oriented programming and its possible application in Additive Manufacturing

Juan Tomás Moreno — Universitat Politècnica de València (Spain)



etsinf

What is object-oriented programming?

The standard data type hierarchy used on most programming languages establishes four "shapes" data can assume (**float** and **integer** numbers, letter **strings** and **boolean** answers), as well as some possibilities to group and order them (**lists**, **arrays**, **sets**, **dictionaries**, **tuples**...). This can be a quite short range to work with; so that is the reason why **object-oriented programming** emerged. This not-so-new programming paradigm allows us to create the objects -*shapes*- we need, including data and code inside of them, in order to store the necessary data and process it exactly the way our task requires, as well as obtaining results related to the interaction of several objects between them.

An easy example — a car

A car-leasing business could store in the same object all the relevant information related to a car (wheel pressure, gasoline, mileage... These are called **attributes**) as well as the operative possibilities they consider (a current lease, any repair needed... These are called **methods**).

On the figure we can see some Python code for our object *Car* (here the objects are called **classes**). The initial function on line 2 includes all the attributes of our car. We can also see a method on line 15, called *newLease*.

```
1  class Car:
2      def __init__(self, fl, fr, bl, br, g, m, l=False):
3          self.wheelPressure = [fl, fr, bl, br]
4          self.gasoline = g
5          self.mileage = m
6          self.onLease = l
7          self.needGas = False
8          self.needCheck = False
9          if self.gasoline < 5:
10             self.needGas = True
11         for wheel in self.wheelPressure:
12             if wheel < 2 or wheel > 3:
13                 self.needCheck = True
14
15     def newLease(self, client, observations=''):
16         self.onLease = True
17         print(f'Lease by {client} accepted')
18         if observations != '':
19             print(observations)
20
```

And what is additive manufacturing?

In industrial design and material sciences, the usage of **computer-aided design** (or **CAD**) has grown exponentially, and has opened new possibilities. A new technique appears, whose goal is to produce items on-demand using a digital model and without the need for molds. So, all in all, we can understand additive manufacturing as the sum of CAD and **3D printing**. This new manufacturing concept can create objects by adding layers of material, usually metal and plastic, one on top of the other. This procedure system, when applied to mass production, can reduce costs, eliminate errors, and produce with greater agility and precision. As of today, additive manufacturing is particularly applied in sectors where customization and precision are crucial. For example, it is used to manufacture implants or surgical instruments for the healthcare sector or parts and components for the aerospace industry.

A key approach to powder-bed-based processes

The manufacture of smaller parts and pieces is usually performed using raw metal powders, in order to maximize the energy and power applied, relying on the conductivity and irradiation capacities different metals provide. As it is performed on a smaller scale, **Powder-Bed Fusion (PBF)** requires a key focus on **position** and **orientation**, in order to use the **least amount of support structure possible**. Beginning from a 3D piece, we could obtain points from R^3 and establish triangular partitions, from which we would obtain the necessary data related to its position in R^3 coordinates and calculate their inclination with respect to the horizontal plane. This is a theoretical approach coded in Python, which just shows the geometrical solution to a simple trigonometry problem: finding the angle between a vector and a plane. We would create the object ***Partition***, with three attributes (the three vertices of our triangle) and two methods: ***center***, which returns the central point of our partition, and ***angle***, which returns the inclination in degrees that it presents.

```python
import numpy as np

class Partition:
    def __init__(self, a, b, c): # Each vertex expressed as a (x,y,z) tuple
        self.v1 = np.array(a)
        self.v2 = np.array(b)
        self.v3 = np.array(c)

    def center(self):
        res = ((self.v1[0] + self.v2[0] + self.v3[0]) / 3,
               (self.v1[1] + self.v2[1] + self.v3[1]) / 3,
               (self.v1[2] + self.v2[2] + self.v3[2]) / 3)
        return (round(res[0], 3), round(res[1], 3), round(res[2], 3))

    def angle(self):
        planeV = [0, 0, 1] # Using the horizontal plane as our "surface"
        normalV = np.cross(self.v2-self.v1, self.v3-self.v1)
        angle_rad = np.arccos(np.dot(normalV, planeV) / (np.linalg.norm(normalV) * np.linalg.norm(planeV)))
        res = np.degrees(angle_rad) # Converting radian to hexadecimal
        return round(res, 3)

```

```
In [2]:
# TEST
v1 = (0, 0, 0)
v2 = (10, 0, 0)
v3 = (0, 10, 10)
test = Partition(v1, v2, v3)
a = test.angle()
a

Out[2]: 45.0

In [3]:
c = test.center()
c

Out[3]: (3.333, 3.333, 3.333)
```

We could test our two methods on a Jupyter Notebook, by creating a simple object whose orientation could be intuited with the naked eye. Our triangle here would have an inclination of 45 degrees.

Using an **iterative** function, we could obtain the percentage of the surface of the piece that would require external support. After retrieving points on the surface of our object as (x, y, z) tuples and adding them to a list, we would start building partitions. Our function receives as input the aforementioned list and a threshold angle. This last parameter will allow us to determine whether a partition would need support in order to stand still on a surface (e.g. if the inclination of an object is below 45°, it could fall, so it would need external support). Our threshold should probably be somewhere between 40 and 50 degrees, depending on the piece we are working with.

```
26  def supportLevel(incl, points):
27      supp = 0
28      for i in range(len(points)-2):
29          t = Partition(points[i], points[i+1], points[i+2])
30          if t.angle() < incl:
31              supp += 1
32      return (supp/len(points))*100
33
```

This function would return misleading results, as it reads the points on a FCFS basis, and not based on proximity. This is why we should include as a parameter a list/set of partitions instead of a list of points:

```
38  def supportLevel_alt(incl, partitions):
39      supp = 0
40      for p in partitions:
41          if p.angle() < incl:
42              supp += 1
43      return (supp/len(partitions))*100
44
```

Additionally, we could rescale our problem to work with bigger parts. If we retrieved too many points from a bigger surface, we could just reduce this amount using a **recursive** function; we would create partitions from our list of points and obtain their centers, in order to work with a shorter list.

```
46  def rescale(points, n):
47      if len(points) < n:
48          return points
49      partitions = []
50      centers = []
51      for i in range(len(points)-2):
52          t = Partition(points[i], points[i+1], points[i+2])
53          partitions.append(t)
54      for p in partitions:
55          c = t.center()
56          centers.append(c)
57      return rescale(centers, n)
58
```