Multiple Representations in Python

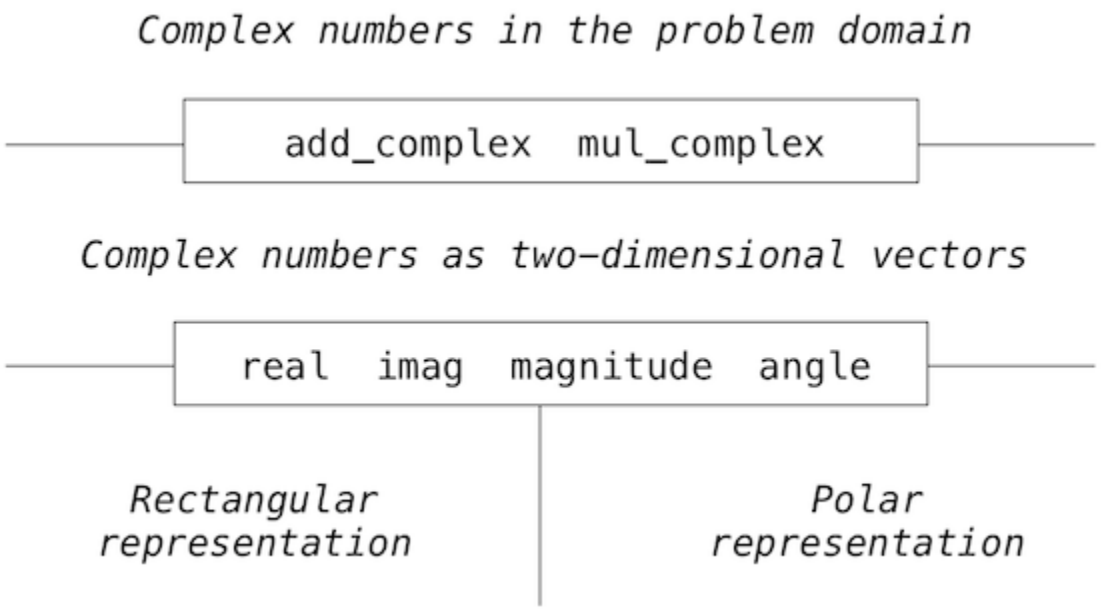
Data abstraction, using objects or functions, is a powerful tool for managing complexity. Abstract data types allow us to construct an abstraction barrier between the underlying representation of data and the functions or messages used to manipulate it. However, in large programs, it may not always make sense to speak of "the underlying representation" for a data type in a program. For one thing, there might be more than one useful representation for a data object, and we might like to design systems that can deal with multiple representations.

To take a simple example, complex numbers may be represented in two almost equivalent ways: in rectangular form (real and imaginary parts) and in polar form (magnitude and angle). Sometimes the rectangular form is more appropriate and sometimes the polar form is more appropriate. Indeed, it is perfectly plausible to imagine a system in which complex numbers are represented in both ways, and in which the functions for manipulating complex numbers work with either representation.

More importantly, large software systems are often designed by many people working over extended periods of time, subject to requirements that change over time. In such an environment, it is simply not possible for everyone to agree in advance on choices of data representation. In addition to the data-abstraction barriers that isolate representation from use, we need abstraction barriers that isolate different design choices from each other and permit different choices to coexist in a single program. Furthermore, since large programs are often created by combining pre-existing modules that were designed in isolation, we need conventions that permit programmers to incorporate modules into larger systems additively, that is, without having to redesign or re-implement these modules.

We begin with the simple complex-number example. We will see how message passing enables us to design separate rectangular and polar representations for complex numbers while maintaining the notion of an abstract "complex-number" object. We will accomplish this by defining arithmetic functions for complex numbers

(`add_complex`, `mul_complex`) in terms of generic selectors that access parts of a complex number independent of how the number is represented. The resulting complex-number system contains two different kinds of abstraction barriers. They isolate higher-level operations from lower-level representations. In addition, there is a vertical barrier that gives us the ability to separately design alternative representations.



As a side note, we are developing a system that performs arithmetic operations on complex numbers as a simple but unrealistic example of a program that uses generic operations. A complex number type is actually built into Python, but for this example we will implement our own.

Like rational numbers, complex numbers are naturally represented as pairs. The set of complex numbers can be thought of as a two-dimensional space with two orthogonal axes, the real axis and the imaginary axis. From this point of view, the complex number`z = x + y * i` (where `i*i = -1`) can be thought of as the point in the plane whose real coordinate is `x` and whose imaginary coordinate is `y`. Adding complex numbers involves adding their respective `x` and `y` coordinates.

When multiplying complex numbers, it is more natural to think in terms of representing a complex number in polar form, as a magnitude and an angle. The product of two

complex numbers is the vector obtained by stretching one complex number by a factor of the length of the other, and then rotating it through the angle of the other.

Thus, there are two different representations for complex numbers, which are appropriate for different operations. Yet, from the viewpoint of someone writing a program that uses complex numbers, the principle of data abstraction suggests that all the operations for manipulating complex numbers should be available regardless of which representation is used by the computer.

Interfaces. Message passing not only provides a method for coupling behavior and data, it allows different data types to respond to the same message in different ways. A shared message that elicits similar behavior from different object classes is a powerful method of abstraction.

As we have seen, an abstract data type is defined by constructors, selectors, and additional behavior conditions. A closely related concept is an *interface*, which is a set of shared messages, along with a specification of what they mean. Objects that respond to the special `__repr__` and `__str__` methods all implement a common interface of types that can be represented as strings.

In the case of complex numbers, the interface needed to implement arithmetic consists of four messages: `real`, `imag`,`magnitude`, and `angle`. We can implement addition and multiplication in terms of these messages.

We can have two different abstract data types for complex numbers that differ in their constructors.

- `ComplexRI` constructs a complex number from real and imaginary parts.
- `ComplexMA` constructs a complex number from a magnitude and angle.

With these messages and constructors, we can implement complex arithmetic.

```
>>> def add_complex(z1, z2):
        return ComplexRI(z1.real + z2.real, z1.imag + z2.imag)
>>> def mul_complex(z1, z2):
```

```
        return ComplexMA(z1.magnitude * z2.magnitude, z1.angle +
z2.angle)
```

The relationship between the terms "abstract data type" (ADT) and "interface" is subtle. An ADT includes ways of building complex data types, manipulating them as units, and selecting for their components. In an object-oriented system, an ADT corresponds to a class, although we have seen that an object system is not needed to implement an ADT. An interface is a set of messages that have associated meanings, and which may or may not include selectors. Conceptually, an ADT describes a full representational abstraction of some kind of thing, whereas an interface specifies a set of behaviors that may be shared across many things.

Properties. We would like to use both types of complex numbers interchangeably, but it would be wasteful to store redundant information about each number. We would like to store either the real-imaginary representation or the magnitude-angle representation.

Python has a simple feature for computing attributes on the fly from zero-argument functions. The `@property` decorator allows functions to be called without the standard call expression syntax. An implementation of complex numbers in terms of real and imaginary parts illustrates this point.

```
>>> from math import atan2
>>> class ComplexRI(object):
        def __init__(self, real, imag):
            self.real = real
            self.imag = imag
        @property
        def magnitude(self):
            return (self.real ** 2 + self.imag ** 2) ** 0.5
        @property
        def angle(self):
            return atan2(self.imag, self.real)
        def __repr__(self):
            return    'ComplexRI({0},    {1})'.format(self.real,
self.imag)
```

A second implementation using magnitude and angle provides the same interface because it responds to the same set of messages.

```
>>> from math import sin, cos
```

```
>>> class ComplexMA(object):
        def __init__(self, magnitude, angle):
            self.magnitude = magnitude
            self.angle = angle
        @property
        def real(self):
            return self.magnitude * cos(self.angle)
        @property
        def imag(self):
            return self.magnitude * sin(self.angle)
        def __repr__(self):
            return  'ComplexMA({0},  {1})'.format(self.magnitude,
self.angle)
```

In fact, our implementations of `add_complex` and `mul_complex` are now complete; either class of complex number can be used for either argument in either complex arithmetic function. It is worth noting that the object system does not explicitly connect the two complex types in any way (e.g., through inheritance). We have implemented the complex number abstraction by sharing a common set of messages, an interface, across the two classes.

```
>>> from math import pi
>>> add_complex(ComplexRI(1, 2), ComplexMA(2, pi/2))
ComplexRI(1.0000000000000002, 4.0)
>>> mul_complex(ComplexRI(0, 1), ComplexRI(0, 1))
ComplexMA(1.0, 3.141592653589793)
```

The interface approach to encoding multiple representations has appealing properties. The class for each representation can be developed separately; they must only agree on the names of the attributes they share. The interface is also *additive*. If another programmer wanted to add a third representation of complex numbers to the same program, they would only have to create another class with the same attributes.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#multiple-representations