6.009: Fundamentals of Programming

Week 7 Lecture: Custom Types

Adam Hartz

hz@mit.edu

5 April 2021

6.009: Goals

Our goals involve helping you develop your programming skills, in multiple aspects:

- **Programming:** analyzing problems, developing plans
- **Coding:** translating plans into Python
- **Debugging:** developing test cases, verifying correctness, finding and fixing errors

The main high-level focus in 6.009 is on **managing complexity of our programs** as they grow in terms of scope and scale.

To this end, we have spent time discussing (and practicing!):

- high-level design strategies
- details and ”goodies” of Python
- good style
- testing and debugging strategies

One of our main tools here has been a **mental model** of Python’s operation.

The Power of Abstraction

Thinking about complicated systems is *complicated*.

The Power of Abstraction

Thinking about complicated systems is *complicated*.

Thinking about simpler systems is often simpler.

The Power of Abstraction

Thinking about complicated systems is *complicated*.
Thinking about simpler systems is often simpler.

What tools do we have for combining simple ideas into more complex ideas?

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**

The Power of Abstraction

Thinking about complicated systems is *complicated*.
Thinking about simpler systems is often simpler.

What tools do we have for combining simple ideas into more complex ideas?

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**

Example (Python procedures):

- Primitives: `+`, `*`, `==`, `!=`, `...`
- Combination: `if`, `while`, `f(g(x))`, `...`
- Abstraction: `def`

The Power of Abstraction

Thinking about complicated systems is *complicated*.
Thinking about simpler systems is often simpler.

What tools do we have for combining simple ideas into more complex ideas?

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**

Example (Python types):

- Primitives: `int`, `float`, `str`, ...
- Combination: `list`, `set`, `dict`, ...
- Abstraction: `class`

Custom Types

Python provides a means of creating custom types: the `class` keyword

Today:

- Extending our mental model to include classes
- What is `self`?

Example: 2-D Vectors

On the next several slides, we will create a *class* to represent the general notion of 2-dimensional vectors.

Once we have created such a class, we can make *instances* of that class to represent specific 2-dimensional vectors.

Example: 2-D Vectors

```
class Vec2D:
    pass
```

Example: 2-D Vectors

```
class Vec2D:
    pass

v = Vec2D()
```

Example: 2-D Vectors

```
class Vec2D:
    pass

v = Vec2D()

v.x = 3
v.y = 4
```

Example: 2-D Vectors

```
class Vec2D:
    pass

v = Vec2D()

v.x = 3
v.y = 4

def mag(vec):
    return (vec.x**2 + vec.y**2) ** 0.5
```

Example: 2-D Vectors

```
class Vec2D:
    pass

v = Vec2D()

v.x = 3
v.y = 4

def mag(vec):
    return (vec.x**2 + vec.y**2) ** 0.5

print(mag(v))
```

Example: 2-D Vectors

```
class Vec2D:
    ndims = 2

    def mag(vec):
        return (vec.x**2 + vec.y**2) ** 0.5
```

Example: 2-D Vectors

```
class Vec2D:
    ndims = 2

    def mag(vec):
        return (vec.x**2 + vec.y**2) ** 0.5

v = Vec2D()
v.x = 3
v.y = 4
print(v.x)
print(v.ndims)
```

Example: 2-D Vectors

```
class Vec2D:
    ndims = 2

    def mag(self):
        return (self.x**2 + self.y**2) ** 0.5

v = Vec2D()
v.x = 3
v.y = 4
print(v.x)
print(v.ndims)

print(Vec2D.mag(v))

print(v.mag())
```

Example: 2-D Vectors

```
class Vec2D:
    ndims = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def mag(self):
        return (self.x**2 + self.y**2) ** 0.5
```

Example: 2-D Vectors

```
class Vec2D:
    ndims = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def mag(self):
        return (self.x**2 + self.y**2) ** 0.5

v = Vec2D(3, 4)
```

Example: 2-D Vectors

```
class Vec2D:
    ndims = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def mag(self):
        return (self.x**2 + self.y**2) ** 0.5

v = Vec2D(3, 4)

print(v.mag())
```

Summary: Variable and Attribute Lookup

Looking up a *variable*:

1. look in the current frame first
2. if not found, look in the *parent* frame
3. if not found, look in that frame's parent frame
4. ...
5. if not found, look in the global frame
6. if not found, look in the builtins
7. if not found, raise a `NameError`

Summary: Variable and Attribute Lookup

Looking up an *attribute* (in an object, using ”dot” notation):

1. look in the object itself
2. if not found, look in that object’s *class*
3. if not found, look in that class’s superclass
4. if not found, look in *that* class’s superclass
5. ...
6. if not found and no more superclasses, raise an `AttributeError`

Summary: `self`

Additional weirdness: when looking up a class method by way of an instance, that instance will automatically be passed in as the first argument.

For example, the following two pieces of code will do the same thing, if `x` is an instance of class `Foo`:

```
Foo.bar(x, 1, 2, 3)
```

```
x.bar(1, 2, 3
```

By convention, this first parameter is usually called `self`. Even though it's not strictly necessary, it's a good idea to follow that convention.