6.009: Fundamentals of Programming

Week 8 Lecture: Custom Types

Adam Hartz

hz@mit.edu

19 October 2020

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

So we will spend time discussing (and practicing!):

- high-level design strategies
- ways to manage complexity
- details and "goodies" of Python
- a mental model of Python's operation
- testing and debugging strategies

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.

Framework for thinking about complicated systems (”PCAP”):

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**
- Meaningful **Patterns**

The Power of Abstraction

Thinking about complicated systems is *complicated*.

Thinking about simpler systems is often simpler.

Framework for thinking about complicated systems ("PCAP"):

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**
- Meaningful **Patterns**

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.

Framework for thinking about complicated systems ("PCAP"):

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**
- Meaningful **Patterns**

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`?
- Examples of creating new types and integrating them into Python

Example: 2-D Vectors

Over the next few 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(vec):
        return (vec.x**2 + vec.y**2) ** 0.5

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

print(Vec2D.mag(v))
```

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)

print(Vec2D.mag(v))

print(v.mag())
```

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* (in a function call, etc):

1. look in the current frame first
2. if not found, look in the *parent* frame
3. if not found, look in that's 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 with ”dot” notation):

1. look in the object itself (the instance)
2. if not found, look in the 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's 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 the class *Thing*:

```
Thing.foo(x, 1, 2, 3)
```

```
x.foo(1, 2, 3)
```

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