# Python's Magic Methods

Reuven M. Lerner, PhD • reuven@lerner.co.il
http://lerner.co.il/

# Methods

- We define methods on a class
- We then invoke them via an instance
- (We're going to ignore static and class methods)
- The method is found via attribute lookup:
  - Object -> object's class -> super -> … -> object

# Method naming

- We can use whatever method names we want!
- So long as we take "self" as a parameter, Python doesn't really dictate the names that we use

# Except.

- Except for the so-called "magic methods"
- These methods are rarely called directly!
- Rather, they're called automatically by Python, typically as the result of invoking an operator

# __init__

- __init__ is the first method that many Python programmers learn about.
- It's not really a constructor, since the new object already exists when __init__ is invoked.

# __len__

- When you call len(x), it looks for __len__
- Don't call __len__ directly; rather use len()
- You can, of course, return whatever you want, which is the point — whatever is appropriate for your object

# Example

```
class Book(object):
    def __init__(self, title, author, num_pages):
        self.title = title
        self.author = author
        self.num_pages = num_pages
    def __len__(self):
        return self.num_pages


>>> b = Book('The Terrible Two', 'Mac Barnett', 224)
>>> len(b)
224
```

# Really magic methods

- Most people who learn about Python object learn about __init__, __len__, and probably __repr__ and __str__.
- But the magic methods go way beyond that!

# __add__

- Add two elements together
- __add__ gets two parameters
  - self
  - Another object
- How do you add them? That's up to you!
- Typically, you'll return a new instance of the same class

# Example

```
class Foo(object):

    def __init__(self, x):

       self.x = x

    def __add__(self, other):

       return Foo(self.x + other.x)

    def __repr__(self):

      return "Instance of f, x = {}".format(self.x)


f1 = Foo(10)

f2 = Foo(20)

print(f1 + f2)
```

# __iadd__

- We can also address what happens when += is invoked
- This is not the same as __add__!
- The expectation is that you'll change the state of the current object, and then return self.

# Example

```
class Foo(object):

    def __init__(self, x):

       self.x = x

    def __iadd__(self, other):

      self.x += other.x

      return self

    def __repr__(self):

       return "Instance of f, x = {}".format(self.x)


f1 = Foo(10)

f2 = Foo(20)

f1 += f2

print("Now f1 = {}".format(f1))
```

# Making __add__ flexible

- Remember, the second argument can be anything
- If we want, we can play games with that — checking for attributes

# Example

```
class Foo(object):

    def __init__(self, x):

        self.x = x

    def __add__(self, other):

        if hasattr(other, 'x'):

            return Foo(self.x + other.x)

        else:

            return Foo(self.x + other)

   def __repr__(self):

        return "Instance of f, x = {}".format(self.x)


f1 = Foo(10)

f2 = Foo(20)

print("f1 + f2 = {}".format(f1 + f2))

print("f1 + 10 = {}".format(f1 + 50))
```

# Reversible?

- What if we now say

```
print("10 + f1 = {}".format(50 + f1))
```

- What will Python do?

```
f1 + 10 = Instance of f, x = 60

Traceback (most recent call last):

  File "./foo.py", line 29, in <module>

    print("10 + f1 = {}".format(50 + f1))

TypeError: unsupported operand type(s) for
+: 'int' and 'Foo'
```

# __radd__

- Auto-reversing: We get self and other, and now invoke our previously defined __add__:

```
def __radd__(self, other):

    return self.__add__(other)
```

# How does this work?

- Python tries to do it the usual way
- It gets NotImplemented back
  - Not an exception!
  - An instance of Not ImplementedType!
- Python then tries (in desperation) to turn things around… and thus, __radd__

# Type conversions

- You probably know about __str__ already
- But what about __int__? Or even __hex__?

# Example

```
def __int__(self):
    return int(self.x)
def __hex__(self):
    return hex(int(self.x))
```

# Boolean conversions

- Your object will always be considered True in a boolean context, unless you define __nonzero__ (__bool__ in Python 3)

# Example

```
class Foo(object):

    pass

f = Foo()



>>> bool(f)

True
```

# But with __nonzero__...

```
class Foo(object):
    def __init__(self, x):
        self.x = x
    def __nonzero__(self):
        return bool(self.x)


>>> f = Foo(1)
>>> bool(f)
True
>>> f = Foo(0)
>>> bool(f)
False
```

# Format

- Originally, Python used the % syntax for pseudo-interpolation:

```
name = 'Reuven'

print('Hello %s' % name)
```

- But there are lots of problems with it, so it's better to use str.format

# str.format

```
name = 'Reuven'

print('Hello {0}'.format(name))
```

# Pad it!

```
name = 'Reuven'

print('Hello, {0:20}!'.format(name))
```

# Move right

```
print('Hello, {0:>20}!'.format(name))
```

# Custom formats

- It turns out that our objects can also handle these custom formats!
- If we define __format__ on our object, then it'll get the format code (i.e., whatever comes after the :)
- We can then decide what to do with it

```
class Person(object):

    def __init__(self, given, family):

        self.given = given

        self.family = family

    def __format__(self, format):

       if format == 'familyfirst':

            return "{} {}".format(self.family, self.given)

      elif format == 'givenfirst':

           return "{} {}".format(self.given, self.family)

      else:

          return "BAD FORMAT CODE"
```

# Using it

```
>>> p = Person('Reuven', 'Lerner')

>>> "Hello, {}".format(p)

'Hello, BAD FORMAT CODE'


>>> "Hello, {:familyfirst}".format(p)

'Hello, Lerner Reuven'


>>> "Hello, {:givenfirst}".format(p)

'Hello, Reuven Lerner'
```

# Pickle

- Pickle allows you to serialize, or marshall objects
- You can then store them to disk, or send them on the network
- Pickle works with most built-in Python data types, and classes built on those types

# Pickling simple data

```
import pickle

d = {'a':1, 'b':2}

p = pickle.dumps(d)

new_d = pickle.loads(p)
```

# Custom pickling

- Define some magic methods (of course) to customize your pickling:
- __getstate__ returns a dictionary that should reflect the object. Want to add to (or remove from) what is being pickled? Just modify a copy of self.__dict__ and return it!
- Don't modify the dictionary itself…

# Example

```
class Foo(object):
    def __init__(self, x):
        self.x = x
    def __getstate__(self):
        odict = self.__dict__.copy()
        odict['y'] = self.x * 2
        return odict
```

```
>>> f = Foo(10)
>>> p = pickle.dumps(f)
>>> new_f = pickle.loads(p)
>>> vars(new_f)
{'x': 10, 'y': 20}
```

# Equality

- What makes two object equal?
- By default in Python, they're only equal if their ids are equal
- (Yes, every object has a unique ID number — use the "id" function to find it!)
- You change this by defining __eq__!

# Changing equality

```
class Foo(object):

    def __init__(self, x):

        self.x = x

    def __eq__(self, other):

       return self.x == other.x


>>> f1 = Foo(10)

>>> f2 = Foo(10)

>>> f1 == f2

True
```

# Hashing

- If two objects are equal, then maybe they should hash to the same value, right?
- We can set the __hash__ attribute to be a method that returns whatever we want

# Playing with __hash__

```
class Foo(object):
    def __init__(self, x):
        self.x = x
    def __hash__(self):
        return hash(self.x)
>>> f = Foo('a')
>>> hash(f)
12416037344
>>> hash('a')
12416037344
f = Foo(1)
>>> hash(f)
1
```

# Messing around with hashing

- What if we have __hash__ return a random number each time?
- That will have some interesting consequences…

# Questions?

- Follow me: @reuvenmlerner
- Buy my book, "Practice Makes Python"
  - 10% off with offer code "webinar"
- Get my newsletter: http://lerner.co.il/newsletter
- Read my blog: http://blog.lerner.co.il/
- Learn a bit: http://DailyTechVideo.com/