Ada Lovelace Day

Ada Lovelace, born 1815, was a writer, mathematician, and correspondent of Charles Babbage. Charles Babbage designed the "analytical engine." Ada wrote its first program (to compute Bernoulli numbers).

Generic Functions

An abstraction might have more than one representation:
- Python has many sequence types: tuples, ranges, lists, etc.
An abstract data type might have multiple implementations:
- Some representations are better suited to some problems
A function might want to operate on multiple data types

Today's Topics:
- Generic functions using message passing
- String representations of objects
- Multiple representations of abstract data types
- Property methods

String Representations

An object value should behave like the kind of data it is meant to represent:
For instance, by producing a string representation of itself:
Strings are important: they represent language and programs:
In Python, all objects produce two string representations:
- The "str" is legible to humans
- The "repr" is legible to the Python interpreter

When the "str" and "repr" strings are the same, we're doing something right in our programming language!

The "repr" String for an Object

The repr function returns a Python expression (as a string) that evaluates to an equal object:
repr(object) -> string
Return the canonical string representation of the object. For most object types, eval(repr(object)) == object.
The result of calling repr on the value of an expression is what Python prints in an interactive session:

```
>>> 32e12
12000000000000.0
>>> print(repr(32e12))
12000000000000.0
```

Some objects don't have a simple Python-readable string:

```
>>> repr(min)
'\'<built-in function min>\'
```

The "str" String for an Object

Human interpretable strings are useful as well:

```
>>> import datetime
>>> today = datetime.date(2011, 10, 7)
>>> repr(today)
'datetime.date(2011, 10, 7)'
>>> str(today)
'2011-10-07'
```
Message Passing Enables Polymorphic Functions

**Polymorphic function:** A function that can be applied to many (poly) different forms (morph) of data

str and repr are both polymorphic; they apply to anything;
repr invokes a zero-argument method __repr__ on its argument

```python
>>> today.__repr__()
'datetime.date(2011, 10, 7)'
```

str invokes a zero-argument method __str__ on its argument

```python
>>> today.__str__()
'2011-10-07'
```

Implementing repr and str

The behavior of repr is slightly more complicated than invoking __repr__ on its argument:
- An instance attribute called __repr__ is ignored (demo)
- **Question:** How would we implement this behavior?

The behavior of str:
- An instance attribute called __str__ is ignored
- If no __str__ attribute is found, uses repr string (demo)
- **Question:** How would we implement this behavior?

Interfaces

Message passing allows **different data types** to respond to the same message

A shared message that elicits similar behavior from different object classes is a powerful method of abstraction

An interface is a set of shared messages, along with a specification of what they mean

Classes that implement __repr__ and __str__ methods that return Python- and human-readable strings thereby implement an interface for producing Python string representations

Multiple Representations of Abstract Data

Rectangular and polar representations for complex numbers

Most operations don’t care about the representation

Some mathematical operations are easier on one than the other

An Interface for Complex Numbers

All complex numbers should produce real and imag components
All complex numbers should produce a magnitude and angle

**Demo**

Using this interface, we can implement complex arithmetic

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

```python
>>> def mul_complex(z1, z2):
    return ComplexMA(z1.magnitude * z2.magnitude, z1.angle + z2.angle)
```
The Rectangular Representation

The `@property` decorator allows zero-argument methods to be called without the standard call expression syntax.

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)

The Polar Representation

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)

Using Complex Numbers

Either type of complex number can be passed as either argument to `add_complex` or `mul_complex`.

```python
>>> 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)

>>> from math import pi

>>> add_complex(ComplexRI(1, 2), ComplexMA(2, pi/2))
ComplexRI(3.0, 2.0)

>>> mul_complex(ComplexRI(0, 1), ComplexMA(0, 1))
ComplexMA(1.0, 3.141592653589793)
```

Special Methods

Adding instances of user-defined classes use `__add__` method.

```python
>>> ComplexRI(1, 2) + ComplexMA(2, 0)
ComplexRI(3.0, 2.0)

>>> ComplexRI(0, 1) * ComplexRI(0, 1)
ComplexMA(1.0, 3.141592653589793)
```

http://diveintopython3.org/special-method-names.html
http://docs.python.org/py3k/reference/datamodel.html#special-method-names