61A Lecture 20

Monday, October 21

Announcements

- Homework 6 is due Tuesday 10/22 @ 11:59pm
  - Includes a mid-semester survey about the course so far
- Project 3 is due Thursday 10/24 @ 11:59pm
  - Extra reader office hours this week:
    - Tuesday 6-7:30 in Soda 405
    - Wednesday 5:30-7 in Soda 405
    - Thursday 5:30-7 in Soda 320
- Midterm 2 is on Monday 10/28 7pm-9pm
  - Topics and locations are posted on the course website
  - Have an unavoidable conflict? Fill out the conflict form by Friday 10/25 @ 11:59pm
  - Review session on Saturday 10/26 1pm-4pm in 1 Pimentel
  - Student-organized "engineering bowl" about midterm 2 on Tuesday 4pm-6pm in 240 Bechtel
- Homework 7 is due Tuesday 11/5 @ 11:59pm (Two weeks)

Generic Functions

Generic Functions

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
- String representations of objects
- Property methods
- Multiple representations of data using the Python object system

String Representations

String Representations

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**, that's a sign that a programming language is legible to humans!

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.

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
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(2013, 10, 21)
>>> repr(today)
'datetime.date(2013, 10, 21)'
>>> str(today)
'2013-10-21'
```

The result of calling str on the value of an expression is what Python prints using the `print` function.

(Demo)

Implementing str and repr

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.

```
>>> today.__repr__()
'datetime.date(2012, 10, 8)'
```

str invokes a zero-argument method __str__ on its argument.

```
>>> today.__str__()
'2012-10-08'
```


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?
- str is a class, not a function

Interfaces

Interfaces

Message passing: Objects interact by passing messages, such as attribute names.

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.**

Examples:

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

Classes that implement __len__ and __getitem__ are sequences.

Property Methods

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numer = 4
>>> f.float_value
0.8
>>> f.denom -= 3
>>> f.float_value
2.0
```

The `@property` decorator on a method designates that it will be called whenever it is *looked up* on an instance.

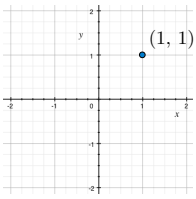
It allows zero-argument methods to be called without an explicit call expression.

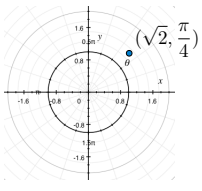
(Demo)

Example: Complex Numbers

Multiple Representations of Abstract Data

Rectangular and polar representations for complex numbers

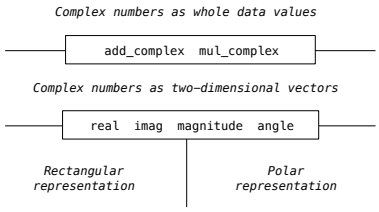




Most operations don't care about the representation.

Some mathematical operations are easier on one than the other.

Arithmetic Abstraction Barriers



Implementing Complex Numbers

An Interface for Complex Numbers

All complex numbers should have real and imag components.

All complex numbers should have a magnitude and angle.

(Demo)

Using this interface, 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 Rectangular Representation

```
class ComplexRI:

    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)
```

Property decorator: "Call this function on attribute look-up"

math.atan2(y,x): Angle between x-axis and the point (x,y)

The Polar Representation

```
class ComplexMA:

    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:

```
>>> 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(1.0000000000000002, 4.0)
>>> mul_complex(ComplexRI(0, 1), ComplexRI(0, 1))
ComplexMA(1.0, 3.141592653589793)
```