Announcements

- Homework 5 is due Wednesday 10/15 @ 11:59pm
- Homework party Monday 10/13 6pm-8pm in 2050 VLSB
- Homework is graded on effort; you don't need to spend 8 hours on one problem
- Project 3 is due Thursday 10/23 @ 11:59pm
- Midterm 2 is on Monday 10/27 7pm-9pm
- Class Conflict? Fill out the conflict form at the top of http://cs61a.org
- Hog strategy contest winners will be announced on Wednesday 10/15 in Lecture
- Fireside chat with Dropbox CEO Drew Houston on Tuesday 10/14 @ 7pm in Wheeler

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
The str and repr strings are often the same, but not always

The repr String for an Object

The repr function returns a Python expression (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.

```
>>> repr(12e12)
'12000000000000.0'
>>> print(repr(12e12))
12000000000000.0
```

Some objects do not have a simple Python-readable string
```
>>> repr(min)
'built-in function min'
```

```
>>> today.__repr__()
'datetime.date(2014, 10, 13)'
>>> today.__str__()
'2014-10-13'
```

The str String for an Object

Human interpretable strings are useful as well:
```
>>> import datetime
>>> today = datetime.date(2014, 10, 13)
>>> repr(today)
'datetime.date(2014, 10, 13)'
>>> str(today)
'2014-10-13'
```

The result of calling str on the value of an expression is what Python prints using the print function:
```
>>> print(today)
2014-10-13
```

Polymorphic Functions

Polymorphic function: A function that applies to many (poly) different forms (morph) of data
str and repr are both polymorphic; they apply to any object
repr invokes a zero-argument method __repr__ on its argument
```
>>> today.__repr__()
'datetime.date(2014, 10, 13)'
```

str invokes a zero-argument method __str__ on its argument
```
>>> today.__str__()
'2014-10-13'
```
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! Only class attributes are found
- Question: How would we implement this behavior?

The behavior of `str` is also complicated:
- An instance attribute called `__str__` is ignored
- If no `__str__` attribute is found, uses `repr` string
- Question: How would we implement this behavior?
- `str` is a class, not a function

Example: Complex Numbers

---

Property Methods

Often, we want the value of instance attributes to stay in sync:

```python
>>> f = Rational(3, 5)
>>> f.numer = 4
>>> f.denom = 2
```

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)

---

Multiple Representations of Abstract Data

Rectangular and polar representations for complex numbers

Most programs don’t care about the representation

Some arithmetic operations are easier using one representation than the other

---

Implementing Complex Arithmetic

Assume that there are two different classes that both represent Complex numbers

<table>
<thead>
<tr>
<th>Number</th>
<th>Rectangular representation</th>
<th>Polar representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+0j</td>
<td>ComplexRI(1, 0)</td>
<td>ComplexMA(sqrt(2), pi/4)</td>
</tr>
</tbody>
</table>

Perform arithmetic using the most convenient representation

```python
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    def add(self, other):
        return ComplexRI(self.real + other.real, self.imag + other.imag)
    def mul(self, other):
        return ComplexMA(abs(self) * abs(other), self.angle + other.angle)
```

(Demo)

---

Complex Arithmetic Abstraction Barriers

<table>
<thead>
<tr>
<th>Parts of the program that…</th>
<th>Treat complex numbers as…</th>
<th>Using…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use complex numbers to perform computation</td>
<td>whole data values</td>
<td>s.add(y, s.mul(y))</td>
</tr>
</tbody>
</table>

Add complex numbers | real and imaginary parts | real, imag, ComplexRI |

Multiply complex numbers | magnitudes and angles | magnitude, angle, ComplexMA |

Implementation of the Python object system
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
All complex numbers should share an implementation of add and mul

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)

The property decorator allows zero-argument methods to be called without the standard call expression syntax, so that they appear to be simple attributes

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 either argument to add or mul:

class Complex:
    def add(self, other):
        return ComplexRI(self.real + other.real, self.imag + other.imag)
    def mul(self, other):
        return ComplexMA(self.magnitude * other.magnitude, self.angle + other.angle)

>>> from math import pi
>>> ComplexRI(1, 1).add(ComplexMA(3, pi/2))
(1+4·p, 1)
>>> ComplexRI(1, 1).mul(ComplexMA(1, 1))
(3.145926535897931, 3.145926535897931)