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.

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

Some objects do not have a simple Python-readable string:

```python
>>> repr(min)

'<built-in function min>'
```

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:

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

`str` invokes a zero-argument method `__str__` on its argument:

```python
>>> 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.

Interfaces

Message passing: Objects interact by looking up attributes on each other (passing messages).
The attribute look-up rules allow different data types to respond to the same message.

A shared message (attribute name) 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.

Example:
Classes that implement `__repr__` and `__str__` methods that return Python- and human-readable strings implement an interface for producing string representations.

Property Methods

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

```python
>>> f = Rational(/, 5)
>>> f.float_value
0.6
>>> f.numer = 4
>>> 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.

Example: Complex Numbers

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+i√2</td>
<td>ComplexRI(1, 1)</td>
<td>ComplexMA(sqrt(2), pi/4)</td>
</tr>
</tbody>
</table>

Perform arithmetic using the most convenient representation.

```python
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)
```

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
```python
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:g}, {1:g})'.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

Using Complex Numbers
```
Either type of complex number can be either argument to add or mul:
```