SICP

God's Programming Book

Lecture-17 Representation

Representation

Slides Adapted from cs61a of UC Berkeley

String Representations

(Demo)

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.
```

The repr String for an Object

The result of calling **repr** on a value is what Python prints in an interactive session

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

Some objects do not 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:

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'1/2'
```

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

```
>>> print(half)
1/2
```

Polymorphic Functions

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

```
>>> half.__repr__()
'Fraction(1, 2)'
```

str invokes a zero-argument method __str__ on its argument

```
>>> half.__str__()
'1/2'
```

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?

```
def repr(x):
    return x.__repr__(x)
```

```
def repr(x):
    return x.__repr__()
```

```
def repr(x):
    return type(x).__repr__(x)
```

```
def repr(x):
    return type(x).__repr__()
```

```
def repr(x):
    return super(x).__repr__()
```

Implementing repr and str

The behavior of **str** is also complicated:

- An instance attribute called __str__ is ignored
- If no __str__ attribute is found, uses **repr** string
- (By the way, **str** is a class, not a function)
- Question: How would we implement this behavior?

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

Interfaces

Example:

Classes that implement __repr__ and __str__ methods that return

Python-interpretable and human-readable strings implement an

interface for producing string representations

Special Method Names

Special Method Names in Python

- Certain names are special because they have built-in behavior
- These names always start and end with two underscores

`__init__`	Method invoked automatically when an object is constructed
`__repr__`	Method invoked to display an object as a Python expression
`__add__`	Method invoked to add one object to another
`__bool__`	Method invoked to convert an object to True or False
`__float__`	Method invoked to convert an object to a float (real number)

```
>>> zero, one, two = 0, 1, 2
>>> one + two
3
>>> bool(zero), bool(one)
(False, True)
```

Same behavior using methods

```
>>> zero, one, two = 0, 1, 2
>>> one.__add__(two)
3
>>> zero.__bool__(), one.__bool__()
(False, True)
```

Special Methods

Adding instances of user-defined classes invokes either the __add__ or __radd__ method

```
>>> Ratio(1, 3) + Ratio(1, 6)
Ratio(1, 2)

>>> Ratio(1, 3).__add__(Ratio(1, 6))
Ratio(1, 2)

>>> Ratio(1, 6).__radd__(Ratio(1, 3))
Ratio(1, 2)
```

Generic Functions

A polymorphic function might take two or more arguments of different types

Type Dispatching: Inspect the type of an argument in order to select behavior

Type Coercion: Convert one value to match the type of another

```
>>> Ratio(1, 3) + 1
Ratio(4, 3)

>>> 1 + Ratio(1, 3)
Ratio(4, 3)

>>> from math import pi
>>> Ratio(1, 3) + pi
3.4749259869231266
```

Thanks for Listening