String Conversion in Python

We stated in the beginning of this chapter that an object value should behave like the kind of data it is meant to represent, including producing a string representation of itself. String representations of data values are especially important in an interactive language like Python, where the `read-eval-print` loop requires every value to have some sort of string representation.

String values provide a fundamental medium for communicating information among humans. Sequences of characters can be rendered on a screen, printed to paper, read aloud, converted to braille, or broadcast as Morse code. Strings are also fundamental to programming because they can represent Python expressions. For an object, we may want to generate a string that, when interpreted as a Python expression, evaluates to an equivalent object.

Python stipulates that all objects should produce two different string representations: one that is human-interpretable text and one that is a Python-interpretable expression. The constructor function for strings, `str`, returns a human-readable string. Where possible, the `repr` function returns a Python expression that evaluates to an equal object. The docstring for *repr* explains this property:

```
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
120000000000000.0
>>> print(repr(12e12))
120000000000000.0
```

In cases where no representation exists that evaluates to the original value, Python produces a proxy.

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

The `str` constructor often coincides with `repr`, but provides a more interpretable text representation in some cases. For instance, we see a difference between `str` and `repr` with dates.

```
>>> from datetime import date
>>> today = date(2011, 9, 12)
>>> repr(today)
'datetime.date(2011, 9, 12)'
>>> str(today)
'2011-09-12'
```

Defining the `repr` function presents a new challenge: we would like it to apply correctly to all data types, even those that did not exist when `repr` was implemented. We would like it to be a *polymorphic function*, one that can be applied to many (*poly*) different forms (*morph*) of data.

Message passing provides an elegant solution in this case: the `repr` function invokes a method called `__repr__` on its argument.

```
>>> today.__repr__()
'datetime.date(2011, 9, 12)'
```

By implementing this same method in user-defined classes, we can extend the applicability of `repr` to any class we create in the future. This example highlights another benefit of message passing in general, that it provides a mechanism for extending the domain of existing functions to new object types.

The `str` constructor is implemented in a similar manner: it invokes a method called `__str__` on its argument.

```
>>> today.__str__()
'2011-09-12'
```

These polymorphic functions are examples of a more general principle: certain functions should apply to multiple data types. The message passing approach exemplified here is only one of a family of techniques for implementing polymorphic functions. The remainder of this section explores some alternatives.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#string-conversion