CS61A Lecture 19

Friday, October 11th, 2019

Announcements

- Several deadlines pushed back.
- Online discussion section today at 5-6:30pm via Zoom.

String Representations

In the object-oriented programming metaphor, we say an object should behave like real-world objects they represent. One way they do that is how they produce string representations of itself.

In Python, this distinction is built into the language. All objects produce 2 string representations:

1. The `__str__` string is meant to be legible to humans, and it is often just text.
2. And the `__repr__` string is meant to be legible to the interpreter, which means it should be an expression.

The `str` and `repr` strings are often the same, but not always, because Python was fundamentally designed to have its code be human-readable.

The `__repr__` String for an Object

The `repr` function returns a Python expression as a string that evaluates to an equal object.

If we call `help` on `repr`, this is what it returns:

```
repr(object) -> string
Return the canonical string representation of an object.
For most object types, eval(repr(object)) == object.
```

The result of calling `repr` on a value is what Python prints in an interactive session. For example,

```
>>> 12e2
1200.0
>>> print(repr(12e2))
1200,0
```

Some objects do not have a simple Python-readable string, which are usually compound objects. For example, functions and classes:

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

There is no way to put `min` into a Python expression, so it just returns a stand-in.

The `__str__` String for an Object

Human interpretable strings are useful as well, because sometimes we want to communicate things to the user.

For example,

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

A call to `repr` just returns the class constructor itself, byt that's not how humans write fractions.

```
>>> str(half)
'1/2'
```

`str` is a built-in function that takes an object and returns a string that is a human-readable representation of the object itself.

The result of calling `str` on the value of an expression is what Python prints out when you call `print` .

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

Notice `str` has quotation marks while `print` does not.

```
>>> eval(str(half))
0.5
```

This just happens to be coincidentally a float, because `str` strings aren't usually designed to be able to be evaluated.

```
>>> s = "Hello, World"
>>> s
'Hello, World'
>>> print(repr(s))
'Hello, World'
>>> print(s)
Hello, World
>>> print(str(s))
Hello, World
>>> str(s)
'Hello, World'
>>> repr(s)
"'Hello World'"
```

What's going in with `repr` ? `repr` is giving back a string that when evaluated, returns the original string.

```
>>> repr(repr(repr(s)))
'\'"\\\'Hello, World\\\'"\''
```

This is a mess because the backslash is an escape key that helps Python separate which quotation marks are on which level.

```
>>> eval(s)
NameError: name 'Hello' is not defined
```

Polymorphic functions

Polymoprhic functions are functions that apply to many different forms of data.

`str` and `repr` are both polymorphic; they apply to any object. How can this work?

`repr` invokes a zero-argument method `__repr__` on its argument.

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

It's really the `Fraction` class that knows how to print out the `repr`, not the `repr` method.

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

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

So there's a really important idea, like `str` or `repr`, that really doesn't have much logic itself, but defers to the argument that comes in to decide what to do by invoking a method on it.

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 invoked by the `repr` function.
- How would we implement this behavior?

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

`type(x)` skips instance attributes by looking up the type of the argument, then calls the `__repr__` function, which is not a bound method, so we have to explicitly pass in `x`.

The behavior of `str` is even more complicated:

- An instance attribute called `__str__` is ignored! Only class attributes are invoked by the `repr` function.
- If no `__str__` attribute is found, uses `repr` string.
- `str` is a class, not a function, so when you're calling `str,`, you're calling the built-in constructor for the default string type.

- How would we implement this? Let’s use an example:

```
>>> class Bear:
...     def __repr__(self):
...         return 'Bear()'
...
>>> oski = Bear() #Let's try to invoke this several different ways
>>> print(oski)
Bear()
>>> print(str(oski))
Bear()
>>> print(repr(oski))
Bear()
>>> print(oski.__str__())
Bear()
>>> print(oski.__repr__())
Bear()
```

So, right now, they all return exactly the same thing, because the `str` and `repr` functions both refer to the same `__repr__` method.

What if we changed our implementation?

```
>>> class Bear:
...     def __repr__(self):
...         return 'Bear()'
...     def __str__(self):
...         return 'a bear'
...
>>> oski = Bear() #Let's try to invoke this several different ways
>>> print(oski)
a bear
>>> print(str(oski))
a bear
>>> print(repr(oski))
Bear()
>>> print(oski.__str__())
a bear
>>> print(oski.__repr__())
Bear()
```

We can observe the different behavior of here. Just printing `oski` prints whatever is defined in the `__str__` method, as well as anytime we explicitly call for `str` .

To get even more variety, we can define a new `__init__` method.

```
>>> class Bear:
...     def __init__(self):
...         self.__repr__ = lambda: 'oski'
...         self.__str__ = lambda: 'this bear'
...     def __repr__(self):
...         return 'Bear()'
...     def __str__(self):
...         return 'a bear'
...
>>> oski = Bear() #Let's try to invoke this several different ways
>>> print(oski)
a bear
>>> print(str(oski))
a bear
>>> print(repr(oski))
Bear()
>>> print(oski.__str__())
this bear
>>> print(oski.__repr__())
oski
```

Printing out `oski` and the `str` of `oski` will always result in the same thing, as there's no way to make them different.

The `repr` and `str` functions ignore the instance attribute and go straight to the class. But the traditional name lookup will check the instance attributes first.

As a final step to prove our understanding, let's write our own `repr` and `str` functions:

```
def repr(x):
    return type(x).__repr(x)__

def str(x):
    t = type(x)
    if hasattr(t, '__str__'):
        return t.__str__(x)
    else:
        return repr(x)
```

Interfaces

We said that central to OOP metaphor is that objects pass messages between each other, and the method in the language is that objects look up attributes on each other.

The attribute lookup rules allow different data types to respond to the same message, just by having the same attribute name,

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-interpretable and human-readable strings implement an interface for producing string representations.

Let's see if we can build a class that exhibits this interface:

```
class Ratio:
    def __init__(self,n,d):
        self.numer = n
        self.denom = d

    def __repr__(self):
        return 'Ratio({0},{1})'.format(self.numer, self.denom)
        # The {0} and {1} are gaps that can be filled with the format method

    def __str(self)__:
        return '{0}/{1}'.format(self.numer,self.denom)
```

This all works the same way as the built-in fraction class we saw earlier.

Special Method Names

This is a topic special to the Python language. Certain names are special because they have built-in behavior.

These names always start and end with two underscores, that just indicates it interacts with the built-in object system in some way.

- `__init__` is a regular method other than the fact it is invoked automatically when an object is constructed.
- `__repr__` is invoked to display an object as a Python expression.
- `__add__` is a two-argument method invoked to add one object to another.
- `__bool__` is a method invoked to convert an object to True or False.
- `__float__` is a method invoked to convert an object to a float.

For example,

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

What happens when you have two instances of user-defined classes, added together? There are two methods, `__add__` and `__radd__` .

`add` takes the argument on the left and adds it to the one on the right, while `radd` takes the argument on the right and adds it to the one on the left.

For example, we could extend our ratio class before:

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

How can we do this?

```
    def __add__(self,other):
        n = self.numer * other.denom + self.denom * other.numer
        d = self.denom * other.denom
        g = gcd(n,d)
        return Ratio(n//g,d//g)

def gcd(n,d):
    while n!= d:
        n,d = min(n,d), abs(n-d)
```

There's one flaw with this method as it is, because we always assume that we are adding two ratios together. We can extend our definition of `__add__` :

```
def __add__(self,other):
    if isinstance(other,int):
        n = self.numer + self.denom * other
        d = self.denom
    elif isinstance(other,Ratio):
        n = self.numer * other.denom + self.denom * other.numer
        d = self.denom * other.denom
    g = gcd(n,d)
    return Ratio(n//g,d//g)

__radd__ = __add__
```

What happens if we try to add a float? It doesn't seem to make sense that we try to return a ratio again, so let's just return a float.

```
def __add__(self,other):
    if isinstance(other,int):
        n = self.numer + self.denom * other
        d = self.denom
    elif isinstance(other,Ratio):
        n = self.numer * other.denom + self.denom * other.numer
        d = self.denom * other.denom
    elif isinstance(other,float):
        return float(self) + other
    g = gcd(n,d)
    return Ratio(n//g,d//g)

__radd__ = __add__
```

But how do we define the `float` function?

```
def __float__(self):
    return self.numer / self.denom
```

We have used two important ideas here in this example. The `__add__` method shows us type dispatching, where we inspect the type of the argument coming in to decide what to do.

The `float` instance is called type coercion, where we take a value and convert it to another type in order to be able to combine it with some other value.