

Lecture 20. Representation

String Representation

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

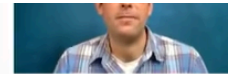
In Python, all objects produce two string representations:

- 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 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`.

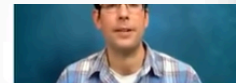
The result of calling **repr** on a value is what Python prints in an interactive session

```
>>> 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:

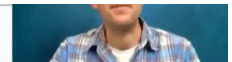
```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'1/2'
```

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

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

demo

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> half
Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> print(half)
1/2
>>> str(half)
'1/2'
>>> eval(repr(half))
Fraction(1, 2)
>>> eval(str(half))
0.5
```



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'
```

简单地来说，就是函数的多态，只不过python是轻类型语言，所以python的函数天生就是多态的。

Interfaces

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

简单地理解就是，`repr`和`str`分别是与计算机沟通的接口与与人沟通的接口，观察`Ratio`这个例子会理解的更加透彻

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

    def __repr__(self):
        return 'Ratio({0}, {1})'.format(self.numer, self.denom)
```

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

Special Method Names in Python

Special Method Names in Python

Certain names are special because they have built-in behavior

These names always start and end with two underscores

<code>__init__</code>	Method invoked automatically when an object is constructed
<code>__repr__</code>	Method invoked to display an object as a Python expression
<code>__add__</code>	Method invoked to add one object to another
<code>__bool__</code>	Method invoked to convert an object to True or False
<code>__float__</code>	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)
```

this is another example of using interface of in order to allow user defined objects to interact to the built-in system of python