Converting Values to Strings

<table>
<thead>
<tr>
<th>str() Function</th>
<th>Backquotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Usage: str(expression)</td>
<td>• Usage: &quot;&lt;expression&gt;&quot;</td>
</tr>
<tr>
<td>• Evaluates the expression</td>
<td>• Evaluates the expression</td>
</tr>
<tr>
<td>• Converts it into a string</td>
<td>• Converts it into a string</td>
</tr>
<tr>
<td>• How does it convert?</td>
<td>• How does it convert?</td>
</tr>
<tr>
<td>• str(True) → 'True'</td>
<td>• '1' → '1'</td>
</tr>
<tr>
<td>• str('abc') → 'abc'</td>
<td>• 'True' → 'True'</td>
</tr>
<tr>
<td>• str(Point()) → (0.0,0.0,0.0)</td>
<td>• 'abc' → 'abc'</td>
</tr>
<tr>
<td>• str(P) → '&lt;class Point&gt;(0.0,0.0,0.0)&gt;'</td>
<td>• Point() → '&lt;class Point&gt;(0.0,0.0,0.0)&gt;'</td>
</tr>
</tbody>
</table>

Special Methods in Python

• Have seen three so far
  • __init__ (for initialize)
  • __str__ (for str())
  • __repr__ (for backquotes)
• Start/end w/ two underscores
• This is standard in Python
• Used in all special methods
• Also for special attributes
• For a complete list, see http://docs.python.org/reference/datamodel.html

Operator Overloading: Multiplication

class Fraction(object):
  """Instances are points in 3D space""
  def __init__(self, x=0, y=0, z=0):
    """Initializer: makes new Point""
    self.x = x
    self.y = y
    self.z = z
  def __str__(self):
    """Returns: string with contents""
    return "<class Point>(%s,%s,%s)" % (self.x, self.y, self.z)
  def __repr__(self):
    """Returns: unambiguous string""
    return '<Point object at %s>' % id(self)

Operator Overloading: Addition

class Fraction(object):
  """Instances are points in 3D space""
  def __init__(self, top, bottom):
    """Initializes a new Fraction""
    self.top = top
    self.bot = bottom
  def __add__(self, value):
    """Returns: sum of self, another Fraction""
    if type(value) == Fraction:
      return self + value
    else:
      return self + value

Comparing Objects for Equality

• Earlier in course, we saw ==
• Not limited to simple attribute comparison
• Ex: cross multiplying

What Does str() Do On Objects?

• Does NOT display contents
  >>> p = Point(1,2,3)
  >>> str(p)
  'Point object at 0x1007a90' |

• Must add a special method
  • __str__ for str()
  • __repr__ for backquotes
• Could get away with just one
  • Backquotes require __repr__
  • str() can use __str__

  (if __str__ is not there)
Issues With Overloading ==

- Overloading == does not also overload comparison !=
  - Must implement __ne__
  - Why? Will see later
- But (x == y) is okay!
- What if you still want to compare Folder names?
  - Use in operator on variables
  - (x in y) True if x, y contain the same folder name
- Check if variable is empty: x is None

Defining a Class

class Fraction(object):
  ...
  def __eq__(self, q):
    ... Returns: True if self, q equal.
    ... False if not, or q not a Fraction
    ... if type(q) != Fraction
  return False

Hiding Methods From Access

- Put underscore in front of a hidden method will make it hidden
- Will not show up in help()
- But it is still there…
- Hidden methods
  - Can be used as helpers inside of the same class
  - But it is bad style to use them outside of this class
- Can do same for attributes
  - Underscore makes it hidden
  - Do not use outside of class

class Fraction(object):
  ...__eq__(self, q):
    ..."""Returns: False if self, q equal, True if not, or q not a Fraction"
    ...if type(q) != Fraction:
      ...return False
  ...__ne__(self, q):
    ..."""Returns: True if d valid denominator, False if not, or q not a Fraction"
    ...if ...}

From Last Time: Data Encapsulation

class Fraction(object):
  ...docstring
  ..."""Instance attributes:
  ..._pre: numerator [int]
  ..._post: denominator [int > 0]"
  ...
  ...def numerator(value):
  ...  ...return _pre
  ...
  ...def denominator(value):
  ...  ...return _post

Properties: Invisible Setters and Getters

class Fraction(object):
  ...docstring
  ..."""Instance attributes:
  ..._pre: numerator [int]
  ..._post: denominator [int > 0]"
  ...
  ..._pre = value
  ...
  ...def num_setter(self, value):
  ...  ...return _pre
  ...
  ...assert type(value) == int
  ...
  ...def denom_setter(self, value):
  ...  ...return _post
  ...
  ...assert type(value) == int

Properties: Invisible Setters and Getters

class Fraction(object):
  ...docstring
  ..."""Instance attributes:
  ..._pre: numerator [int]
  ..._post: denominator [int > 0]"
  ...
  ..._pre = value
  ...
  ...def num_setter(self, value):
  ...  ...return _pre
  ...
  ...assert type(value) == int
  ...
  ...def denom_setter(self, value):
  ...  ...return _post
  ...
  ...assert type(value) == int

Structure of a Proper Python Class

- Do this for all of your attributes
- Naming Convention
  - The underscore means “should not access the attribute directly.”

Properties:

- Instance attributes:
  - numerator: top [int]
  - denominator: bottom [int > 0]"
  ...
  ...def __init__(self, numerator=0, denominator=1):
  ...  ...assert numerator == numerator
  ...  ...assert denominator == denominator
  ...
  ...def __eq__(self, q):
  ...  ..."""Returns: False if self, q equal, True if not, or q not a Fraction"
  ...  ...if type(q) != Fraction:
  ...    ...return False
  ...  ...else:
  ...    ...return (self._pre == q._pre) and (self._post == q._post)