Generic Programming

• Consider the function `find`:
  ```python
def find(L, x, k):
    """Return the index in L of the kth occurrence of x (k>=0),
or None if there isn’t one."""
    for i in range(len(L)):
        if L[i] == x:
            if k == 0:
                return i
            k -= 1

• This same function works on lists, tuples, strings, and (if the keys are consecutive integers) dicts.

• In fact, it works for any list L for which `len` and indexing work as they do for lists and tuples.

That is, `find` is **generic** in the type of L.

Duck Typing

• A *statically typed language* (such as Java) requires that you specify a type for each variable or parameter, one that specifies all the operations you intend to use on that variable or parameter.

• To create a generic function, therefore, your parameters’ types must be subtypes of some particular interface.

• You can do this in Python, too, but it is not a requirement.

• In fact, our `find` function will work on any object that has `__len__` and `__getitem__`, regardless of the object’s type.

• This property is sometimes called **duck typing**: “This parameter must be a duck, and if it walks like a duck and quacks like a duck, we’ll say it *is* a duck.”

Example: The `_repr_` Method

• When the interpreter prints the value of an expression, it must first convert that value to a (printable) string.

• To do so, it calls the `_repr_` method of the value, which is supposed to return a string that suggests how you’d create the value in Python.

  ```python
  >>> "Hello"
  'Hello'
  >>> print(repr("Hello"))
  'Hello'
  >>> repr("Hello")  # What does the interpreter print?
  'du"ll"o'  # This would be printed.
  >>> repr("Hello")  # What does Python print?
  'du"ll"o'
  ```

• (As a convenience, the built-in function `repr(x)` calls the `_repr_` method.)

• User-defined classes can define their own `_repr_` method to control how the interpreter prints them.

Example: The `_str_` Method

• When the `print` function prints a value, it calls the `_str_` method to find out what string to print.

• The constructor for the string type, `str`, does the same thing.

• Again, you can define your own `_str_` on a class to control this behavior. (The default is just to call `_repr_`)

  ```python
  >>> class rational:
  ...     def __init__(self, num, den):
  ...     ...     def __str__(self):
  ...         if self.numer() == 0: return '0'
  ...         elif self.denom() == 1: return str(self.numer())
  ...         else: return '{0}/{1}'.format(self.numer(), self.denom())
  ...     ...     def __repr__(self):
  ...         return 'rational({}, {})'.format(self.numer(), self.denom())
  ...
  >>> print(rational(3,4))
  3/4
  >>> rational(3,4)
  rational(3, 4)
  >>> print(rational(5, 1))
  5
  ```

Aside: A Small Technical Issue

• *str*, *repr*, and *print* all call the methods `_str_` and `_repr_`, ignoring any instance variables of those names.

• For example,

  ```python
  >>> v = rational(3, 4)
  >>> v._str_ = lambda x: "FDD!"
  >>> str(v)
  3/4
  >>> c._str_()
  "FDD!"
  ```

• How could you implement *str* to do this?

• Hint: As in the homework, *type(x)* returns the class of *x*.
Other Generic Method Names

Just as defining `__str__` allows you to specify how your class is printed, Python has many other generic connections to its syntax, which allow programmers great flexibility in expressing things. For example:

<table>
<thead>
<tr>
<th>Method</th>
<th>Implements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__getitem__(S, k)</code></td>
<td><code>S[k]</code></td>
</tr>
<tr>
<td><code>__setitem__(S, k, v)</code></td>
<td><code>S[k] = v</code></td>
</tr>
<tr>
<td><code>__len__(S)</code></td>
<td><code>len(S)</code></td>
</tr>
<tr>
<td><code>__bool__(S)</code></td>
<td><code>bool(S)</code></td>
</tr>
<tr>
<td><code>__add__(S, x)</code></td>
<td><code>S + x</code></td>
</tr>
<tr>
<td><code>__sub__(S, x)</code></td>
<td><code>S - x</code></td>
</tr>
<tr>
<td><code>__mul__(S, x)</code></td>
<td><code>S * x</code></td>
</tr>
<tr>
<td><code>__ge__(S, x)</code></td>
<td><code>S &gt;= x</code></td>
</tr>
<tr>
<td><code>__getattr__(S, N)</code></td>
<td><code>S.N</code></td>
</tr>
<tr>
<td><code>__setattr__(S, N, v)</code></td>
<td><code>S.N = v</code></td>
</tr>
</tbody>
</table>

Properties

- I've said that generally, method calls are the preferred way for clients to access an object (rather than direct access to instance variables.)
- This practice allows the class implementor to hide details of implementation.
- Still it's cumbersome to have to say, e.g., `aPoint.getX()` rather than `aPoint.x`, and `aPoint.setX(v)` rather than `aPoint.x = v`.
- To alleviate this, Python introduced the idea of a property object.
- When a property object is an attribute of an object, it calls a function when it is fetched from its containing object by dot notation.
- The property object can also be defined to call a different function on assignment to the attribute.
- Attributes defined as property objects are called computed or managed attributes.

Properties (Long Form)

class rational:
```python
def __init__(self, num, den):
g = gcd(num, den)
    self._num, self._den = num/g, den/g
def _getNumer(self): return self._num
def _setNumer(self, val): self._num = val / gcd(val, self._denom)
numer = property(_getNumer, _setNumer)
    # Alternatively,
    # numer = property(_getNumer).setter(_setNumer)
```

Properties (Short Form)

```python
@property
def numer(self): return self._num
    # Equivalent to
    # def _getNumer(self): return self._num
    # _numer = property(_getNumer)
    # where _getNumer is some identifier not used anywhere else.
@numer.setter
def numer(self, val):
    # Equivalent to
    # def _setNumer(self, val): self._num = val / gcd(val, self._denom)
    # _numer = numer.setter
    # This is a bit obscure, but the idea is that every property object has a
    # setter method that turns out a new property object that governs both
    # getting and setting of a value.
```