Lecture #15: Generic Functions and Expressivity
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?
'Hello'
```

- (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 \texttt{print} function prints a value, it calls the \_str\_() method to find out what string to print.
- The constructor for the string type, \texttt{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 \texttt{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__
<bound method rational.__str__ of ...>
>>> v.__str__ = lambda x: "FOO!"
>>> # __str__ is now an instance variable of v as well as a
>>> # a method of class rational.
>>> v.__str__
<function <lambda> at ...>
>>> str(v)
3/4
>>> c.__str__()
'FOO!'
```

- 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>_getitem_(S, k)</td>
<td>S[k]</td>
</tr>
<tr>
<td>_setitem_(S, k, v)</td>
<td>S[k] = v</td>
</tr>
<tr>
<td>_len_(S)</td>
<td>len(S)</td>
</tr>
<tr>
<td>_bool_(S)</td>
<td>bool(S)</td>
</tr>
<tr>
<td>_add_(S, x)</td>
<td>S + x</td>
</tr>
<tr>
<td>_sub_(S, x)</td>
<td>S - x</td>
</tr>
<tr>
<td>_mul_(S, x)</td>
<td>S * x</td>
</tr>
<tr>
<td>_ge_(S, x)</td>
<td>S &gt;= x</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>_getattr_(S, ‘N’)</td>
<td>S.N</td>
</tr>
<tr>
<td>_setattr_(S, ‘N’, v)</td>
<td>S.N = v</td>
</tr>
</tbody>
</table>

True or False

Attributes
Iterators and Iterables

• The for statement is actually a generic control construct with the following meaning:

```python
for x in C:
    S
    tmp_iter = iter(C)
    try:
        while True:
            x = tmp_iter.__next__()
            S
    except StopIteration:
        pass
```

• Types for which iter works are called iterable, and those that implement __next__ are iterators (returned by calling iter on an iterable).

• The built-in iter function first tries calling the method __iter__ on the object, so if you define a class containing `def __iter__(self):...`, you’ll have an iterable class.

• In addition, a type is considered iterable if it implements __getitem__, the method that implements the a[...] operator.
The Many Uses of Iterables

• Python cleanly integrates iterables into many contexts, showing the power of a good abstraction.

• The obvious:

   ```python
   for x in anIterable: ...
   L = [ f(x) for x in anIterable]
   ```

• Many functions take iterables as arguments rather than just lists:

   ```python
   list(anIterable)
   set(anIterable)
   map(f, anIterable)
   sum(anIterable)
   max(anIterable)
   all(anIterable)
   ```