Lecture #16: Generic Functions and Expressivity

Generic Programming

- Consider the function `find`:

```
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.

```
>>> "Hello"
'Hello'
>>> print(repr("Hello"))
'Hello'
>>> repr("Hello")     # What does the interpreter print?
```

- (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__`)

```
>>> 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,

```
>>> v = rational(3, 4)
>>> v.__str__ = lambda x: "FOO!"
>>> 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,

Method	Implements	
`__getitem__(S, k)`	`S[k]`	
`__setitem__(S, k, v)`	`S[k] = v`	
`__len__(S)`	`len(S)`	
`__bool__(S)`	`bool(S)`	True or False
`__add__(S, x)`	`S + x`	
`__sub__(S, x)`	`S - x`	
`__mul__(S, x)`	`S * x`	
`__ge__(S, x)`	`S >= x`	
...		
`__getattr__(S, N)`	`S.N`	Attributes
`__setattr__(S, N, v)`	`S.N = v`	

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:
    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)
```

- As a result,

```
>>> a = rational(3, 4)
>>> a.numer      # Calls a._getNumer()
3
>>> a.numer = 5  # Calls a._setNumer(5)
```

Properties (Short Form)

The built-in `property` function is also a decorator:

```
class rational:
    ...
    @property
    def numer(self): return self._num
    # Equivalent to
    # def TMPNAME(self): return self._num
    # numer = property(TMPNAME)
    # where TMPNAME is some identifier not used anywhere else.

    @numer.setter
    def numer(self, val):
    # Equivalent to
    # def TMPNAME(self, val): self._num = val / gcd(val, self._denom)
    # numer = numer.setter(TMPNAME)
```

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.

Iterators

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

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

- Types that implement `__iter__` are called *iterable*, and those that implement `__next__` are *iterators*.
- As usual, the builtin functions `iter(x)` and `next(x)` are defined to call `x.__iter__()` and `x.__next__()`.

Problem: Reconstruct the range class

- Want `Range(1, 10)` to *give* us something that behaves like a Python range, so that

  ```
  for x in Range(1, 10):
      print(x)
  ```

  prints 1–9.

  ```
  class Range:
       ???
  ```