Lecture #16: (Mostly) Interfaces and Generic Functions

Using Base Types

- Sometimes, we want an overriding method in a subtype to augment rather than totally replace an existing method.
- That means that we have to call the original version of the method within the overriding method somehow.
- Can't just do an ordinary method call on self, since that would cause infinite recursion.
- Fortunately, we can explicitly ask for the original version of the method by selecting from the class.

Example: "Memoization"

- Suppose we have
  ```python
class Evaluator:
    def value(self, x):
        # some expensive computation that depends only on x

class FastEvaluator(Evaluator):
    def __init__(self):
        self.__memo_table = {}  # Maps arguments to results
    def value(self, x):
        """A memoized value computation""
        if x not in self.__memo_table:
            self.__memo_table[x] = Evaluator.value(self, x)
        return self.__memo_table[x]
```
- `FastEvaluator.value` must call the `Evaluator.value` method of its base (super) class, but we can't just say `self.value(x)`, since that gives an infinite recursion.

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.

The Idea of an Interface

- In Python, this means any type that fits the following interface:
  ```python
class SequenceLike:
    def __len__(self):
        """My length, as a non-negative integer.""
    def __getitem__(self, k):
        """My kth element, where 0 <= k < self.__len__()"
        return self.__memo_table[k]
```
- This is one way to describe an interface, which in a programming language consists of
  - A syntactic specification (operation names, numbers of parameters), and
  - A semantic specification—its meaning or behavior (given here by English-language comments.)
- Generic functions are written assuming only that their inputs honor particular interfaces.
- The fewer the assumptions in those interfaces, therefore, the more general (and reusable) the function.

Supertypes as Interfaces

- We call the types that a Python class inherits from its supertypes or base types (and the defined class, therefore, is a subtype).
- Good programming practice requires that we treat our supertypes as interfaces, and adhere to them in the subtypes.
- For example, were we to write
  ```python
class MyQueue(SequenceLike):
    def __len__(self): ...
    def __getattr__(self, k): ...
```
then good practice says that `MyQueue.__len__` should take a single parameter and return a non-negative integer, and that `MyQueue.__getitem__` should accept an integer between 0 and the value of `self.__len__()`.
- Python doesn't actually enforce either of these provisions; it's up to programmers to do so.
- Other languages (like C++, Java, or Ada) enforce the syntactic part of the specification.
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 \texttt{find} function will work on any object that responds appropriately to \texttt{len()} and \texttt{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 \textit{is} a duck.”

Consequences of Good Practice

- If we obey the supertype-as-interface guideline, then we can pass any object that has a subtype of \texttt{SequenceLike} to \texttt{find} and expect it to work.
- This fact is an example of what is called the Liskov Substitution Principle, after Prof. Barbara Liskov of MIT, who is generally credited with enunciating it.

Interface as Documentation

- The interface (especially its documentation comments) provides a contract between clients of the interface and its subtypes—implementations of the interface:
  
  “I, the implementor, agree that all the subclasses I define will conform to the signature and comments in this interface, as long as you, the client, obey any restrictions specified in the interface.”
- Since Python does not check or enforce the consistency of supertypes and subtypes, use of the guideline is a matter of individual discipline.
- Enforced or not, the interface type provides a convenient place to document the contract.
- But even when using duck typing, good practice requires that we document the assumptions made by the implementor about parameters to methods (what methods they have, in particular).

Example: The \texttt{__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 \texttt{__repr__}() method of the value, which is supposed to return a string that suggests how you’d create the value in Python.

Example: The \texttt{str} Method

- When the \texttt{print} function prints a value, it calls the \texttt{__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 \texttt{__str__} on a class to control this behavior. (The default is just to call \texttt{__repr__}).

Example: The \texttt{__iter__} Method

- In the homework, we introduce the notion of iterators, another use of duck typing.
- The for statement is actually a generic control construct with the following meaning:

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

- Types that implement \texttt{__iter__} are called \textit{iterable}, and those that implement \texttt{__next__} are \textit{iterators}.
- As usual, the built-in functions \texttt{iter(x)} and \texttt{next(x)} are defined to call \texttt{x.__iter__()} and \texttt{x.__next__}. 

Iterators

- In the homework, we introduce the notion of iterators, another use of duck typing.
- 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 \texttt{__iter__} are called \textit{iterable}, and those that implement \texttt{__next__} are \textit{iterators}.
- As usual, the built-in functions \texttt{iter(x)} and \texttt{next(x)} are defined to call \texttt{x.__iter__()} and \texttt{x.__next__}. 

Problem: Reconstruct the range class

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

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

prints 1–9.

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