Classes & Objects:
A few more tricks of the trade

But first … which would you prefer (and why, or when)?

```
class Point:
    ...
    def move(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy
```

mutable style

```
class Point:
    ...
    def move(self, dx, dy):
        return Point(self.x+dx, self.y+dy)
```

immutable

Mutable and immutable classes in Python

Python has immutable classes

int, str, frozenset, tuple

Python has mutable classes

list, dict, set

Python has mutating and non-mutating versions of some methods

list.sort() and list.reverse()

vs. sorted(li) and reversed(li)

Some considerations

Sometimes a small change is cheaper than a copy (e.g., change one element of a list)

Sometimes mutation makes sharing (aliasing) difficult

The ‘str’ class of Python illustrates both:

- A loop to compose a string from letters is expensive
- But you don’t have to worry about accidental changes to another copy of a string!

Mutation gotcha

Suppose we reuse some Point objects in a Rect method, e.g.,

```
def scale(self, sfx, sfy):
    urx = self.ll.x + sfx * (self.ur.x – self.ll.x)
    ury = self.ll.y + sfy * (self.ur.y – self.ll.y)
    return Rect(self.ll, Point(urx, ury))
def move(self, dx, dy):
    self.ll.move(dx,dy)
    self.ur.move(dx,dy)
```

Rules of thumb

Immutability is nice if the cost is acceptable

Fewer nasty surprises; more flexible

Mutable objects must be clonable

Be clear:

Not returning a result is a clear indication of mutation.

If a method or function returns a result, it should either be non-mutating OR it should very clearly and explicitly document the change.

Magic methods! (aka special methods)

- `__repr__(self)`: "official" representation
- `__str__(self)`: "informal" representation
- `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`:
  - comparisons `<`, `<=`, `>`, `>=`, `==`, `!=`
  - (all independent except `__ne__` calls `__eq__`)
- `__bool__(self)`: truthiness, for use in
  - `if thing:`

Collection magic:

`__len__(self)` : number of elements

`__getitem__(self, key)` : thing[key]

`__setitem__(self, key, value)` :

thing[key] = value

`__iter__(self)` : iterator

for el in thing:

`__contains__(self, key)` : membership

if el in thing

Numeric and Logical operations

`__add__(self, other), __sub__, __mul__, __truediv__, __floordiv__, __mod__, ... :`

`+, -, *, /, //, %, ...`

`__and__(self, other), __xor__, __or__`

and, xor, or

And even more. (Don't try to memorize them.)

Why so much magic?

So that you can write natural-looking, understandable code operating on objects from custom classes.

E.g., maybe Rect + Point -> Rect?

Maybe len(Polyline) is the number of points?

Maybe Polyline[i] is the i'th point?

Maybe our collection has special properties, but can still be indexed like a list?

Class methods

Most methods are called through *objects,* like obj.m(...)

Sometimes we want a method in the *class* itself, typically as an alternative constructor, called a *factory method*

```
class Polyline:
  def __init__(self, points):
     ...
   @classsmethod
  def from_rect(cls, p):
          return cls([(p.llx, p.lly),
                      (p.urx, p.ury)])
```

What is the difference?

```
class Polyline:
    ...
   @classsmethod
   def from_rect(cls, p):
        return cls([(p.llx, p.lly),
                    (p.urx, p.ury)])
```

vs.

```
   @classsmethod
   def from_rect(cls, p):
        return Polyline([(p.llx, p.lly),
                         (p.urx, p.ury)])
```

Let's build a class (or classes)

Option 1: Bag of letters (for Scrabble, anagrams, …) [Collection class]

Option 2: Combinatorial logic gates (simulated circuit) [Related subclasses]

Option 3: <Your idea here>

Class design exercise (option)

For some word games (anagrams, scrabble, ...), a “bag of letters” class is useful

Bag: like a set, but with counts of elements

Operations:

- Create a bag from a string
- Can string s be made from bag b?
- (others ... what do you want?)

Class design exercise (option)

Combinatorial gates: And, Or, Not, ..., plus sources & sinks.

Each gate calculates an output signal based on its input signals.

Set sources to try a combination; sinks are where we read the outcome

Combinatorial as versus Sequential: No loops, steady output for steady input