Object-Oriented Programming Part 2

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Class hierarchy
  - inheritance, subclassing
- Special methods
  - conversion to string
  - length and comparison
- Operator overloading
  - Define your own `+ - * / % & | < > [] in`
  - Operator precedence and associativity

Inheritance

- A way to define a (sub)class by inheriting the common features of a base class
  - No need to rewrite same code - just inherit
  - New feature => override base-class behavior
- A instance of a subclass is also considered an instance of its base class(es)
  - use isa ("instanceof") operator to test instance's membership in a class or its superclass

Example class hierarchy: exceptions

- `BaseException`
- ` +-- SystemExit`
- ` +-- KeyboardInterrupt`
- ` +-- GeneratorExit`
- ` +-- Exception`
- `      +-- StopIteration`
- `      +-- StopAsyncIteration`
- `      +-- ArithmeticError`
- `      |    +-- FloatingPointError`
- `      |    +-- OverflowError`
- `      |    +-- ZeroDivisionError`

Example class hierarchy: exceptions

```
• BaseException
•  +-- SystemExit
•  +-- KeyboardInterrupt
•  +-- GeneratorExit
•  +-- Exception
•       +-- StopIteration
•       +-- StopAsyncIteration
•       +-- ArithmeticError
•       |    +-- FloatingPointError
•       |    +-- OverflowError
•       |    +-- ZeroDivisionError
```

base class (or **superclass**) of all exceptions

FloatingPointError, OverflowError, and ZeroDivisionError are all **subclasses** of ArithmeticError

`isinstance()` for checking class membership

- Syntax: `isinstance(`*objectRef*`,` *classRef*`)`
  - built-in function to determine if objectRef is an instance of classRef
  - i.e., if *classRef* is a class or superclass of *objectRef*

```
>>> a = ArithmeticError()
>>> b = FloatingPointError()
>>> c = OverflowError()
>>> d = ZeroDivisionError()
>>> isinstance(a, ArithmeticError)
True
>>> isinstance(b, ArithmeticError)
True
>>> isinstance(a, OverflowError)
False
```

```
+-- ArithmeticError
|    +-- FloatingPointError
|    +-- OverflowError
|    +-- ZeroDivisionError
```

```
>>> isinstance(b, OverflowError)
False
>>> isinstance(d, FloatingPointError)
False
>>> isinstance(d, ArithmeticError)
True
```

`issubclass()` for subclass relationship

- Syntax: `issubclass(`*sub*`, `*sup*`)`
  - built-in function to determine if *sub* is a subclass of *sup*
  - i.e., if *sup* is a superclass of *sup*

```
+-- ArithmeticError
|    +-- FloatingPointError
|    +-- OverflowError
|    +-- ZeroDivisionError
```

```
>>> issubclass(ArithmeticError, FloatingPointError)
False
>>> issubclass(OverflowError, ZeroDivisionError)
False
>>> issubclass(ZeroDivisionError, ArithmeticError)
True
```

Subclassing

- A way of extending an existing class
  - inherit features from the base class (super class)
  - subclass can add new features (methods, attributes)
  - subclass can overrides the base class's features
  - subclass remains compatible with base class code
    => anywhere base class is used, subclass can be used!!
- Advantages
  - only need to specify difference without copying code
  - Base class remains unmodified, remains stable

Syntax for subclassing

- Syntax

  -
    ```
    class Subclass(BaseClass):
        def method(self): # incl. __init__
            ...
    ```
- To call a method as base class,
  - `super(Subclass, self).method()`

    Or more simply,
  - `super().method()`

Example: MyList

- Want to inherit from built-in list class
- Features to customize
  - `__repr__()` to show `MyList([1, 2, 3])` instead of just `[1, 2, 3]`
  - `sort()` values of different types:
    - Primary order: `None` < numbers < strings < tuples < lists (for now)
  - `find()` to find value recursively

MyList class: __repr__() method

```
class MyList(list):           # inherits from built-in list class
    # also inherit the constructor - no work to do!!
    def __repr__(self):
        return self.__class__.__name__+ '('+super().__repr__()+')'
    # to be continued
if __name__ == '__main__':
    L = MyList([1, 2, 3])
```

Interpreter calls `MyList.__repr__()` instead of `list.__repr__()`

```
% python3 -i mylist.py
>>> L
MyList([1, 2, 3])
>>> L.append(MyList([4, 5, 6]))   # nested MyList!
>>> L
MyList([1, 2, 3, MyList([4, 5, 6])])
>>> L[3][2]   # all other functions/operators still work the same way
6
>>> len(L)
4
```

MyList class: find() method

```
class MyList(list):
    # __repr__() not shown here
    def find(self, val):
        # use an inner function from recursion example!
        def rec_find(L, val):
            if isinstance(L, list) or isinstance(L, tuple):
                for i, v in enumerate(L):
                    p = rec_find(v, val)
                    if p == True:
                        return (i,)
                    if p != False:
                        return (i,)+p
            return L == val
        return rec_find(self, val)
    # to be continued
if __name__ == '__main__':
    P = MyList([7.4, MyList([2, (5, 'a', 'z'), [4, [9, 7], 'b']]),\
                2, 'world', 'bye', (13, 24), 'bye', (14, 28), None])
```

```
% python3 -i mylist.py
>>> P.find(100), P.find('z'), P.find([9,7]), P.find((13, 24))
(False, (1, 1, 2), (1, 2, 1), (5, ))
```

MyList class: sort()

```
class MyList(list):           # inherits from built-in list class
    # __repr__() and find() not shown
    def sort(self):
        D = {'NoneType': 0, 'int': 1, 'float': 1, 'str': 2,
             'tuple': 3, 'list': 4}
        return super().sort(key=lambda x:(D.get(type(x).__name__,5),x))
    # to be continued
if __name__ == '__main__':
    L = MyList([1, 2, 3])
    M = MyList([3, 8, 'hello', 7.4, ('world', 15), [4, 7]])
    N = MyList([7.4, 2,'world','bye', (13, 24), 'bye', (14, 28), None])
```

```
% python3 -i mylist.py
>>> M.sort()
>>> M
MyList([3, 7.4, 8, 'hello', ('world', 15), [4, 7]])
>>> N.sort()
>>> N
MyList([None, 2, 7.4, 'bye', 'bye', 'world', (13, 24), (14, 28)])
```

- Sorted: numbers < strings < tuples < lists

Potential issue with sort()

- original list's sort takes other optional args
  - **sort**(self, /, *, key=None, reverse=False)
- Solution
  - if `key` is not `None` then combine (if possible)
  - `reverse` can be passed along

```
class MyList(list):           # inherits from built-in list class
    # __repr__() not shown
    def sort(self, key=None, reverse=False):
        D = {'NoneType': 0, 'int': 1, 'float': 1, 'str': 2,
             'tuple': 3, 'list': 4}
        return super().sort(key=lambda x: (D.get(type(x).__name__, 5),\
                                key(x) if key is not None else x), \
                            reverse=reverse)
```

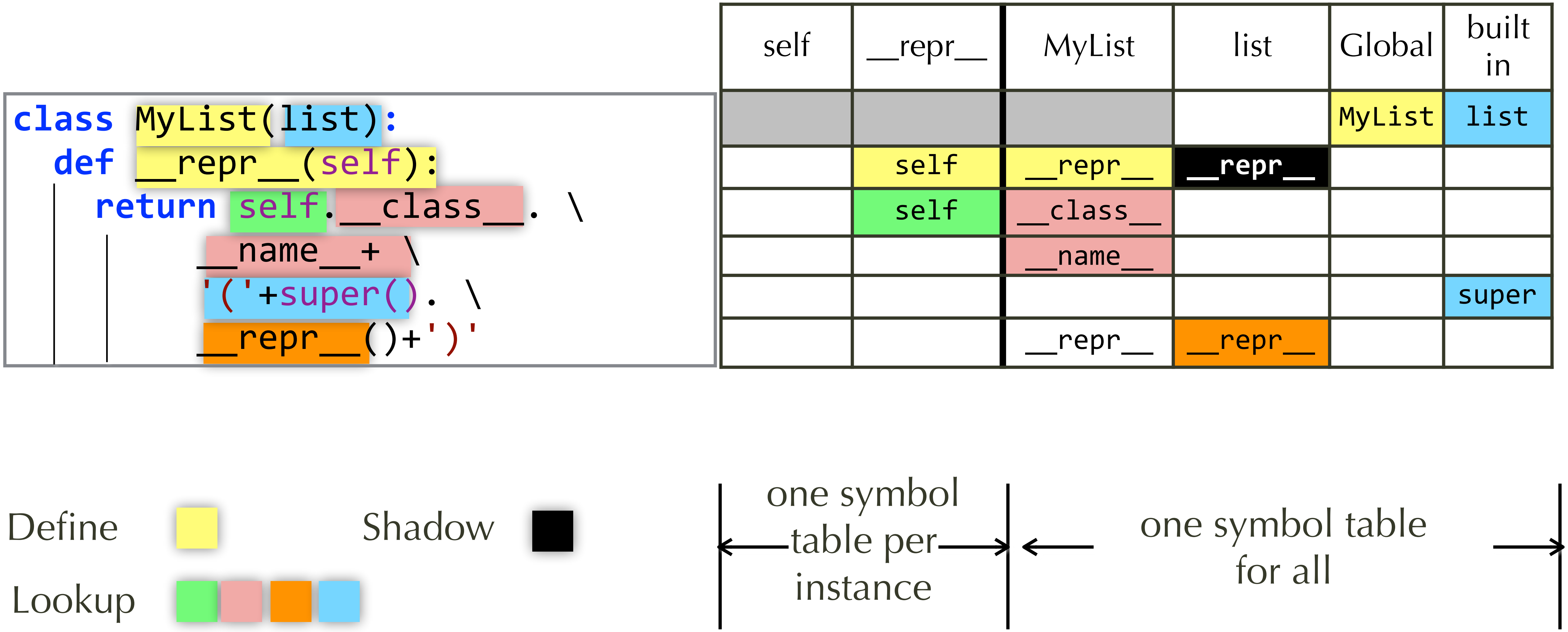
Search order of symbol binding

- Defining a new symbol
  - Always defines new symbols in the "most local" name space
- Looking up an existing symbol
  - instance looks in own symbol first
  - if not found, look in class's name space
  - if not found, look in its superclass's name space

Method calls in a class hierarchy

- All methods are defined in the class's namespace, not the instance's
- access by fully qualified notation
  - *baseClass*.*method*(*instance, args, ...*), *baseClass*.*attribute*
  - *instance.attribute*

Symbol tables for built-in, global, class, method, and instance



```
class MyList(list):
  def __repr__(self):
    return self.__class__. \
         __name__+ \
         '('+super(). \
         __repr__()+')'
```

self	__repr__	MyList	list	Global	built in
▨	▨	▨		MyList	list
	self	__repr__	__repr__		
	self	__class__			
		__name__			
					super
		__repr__	__repr__		

Define 🟨

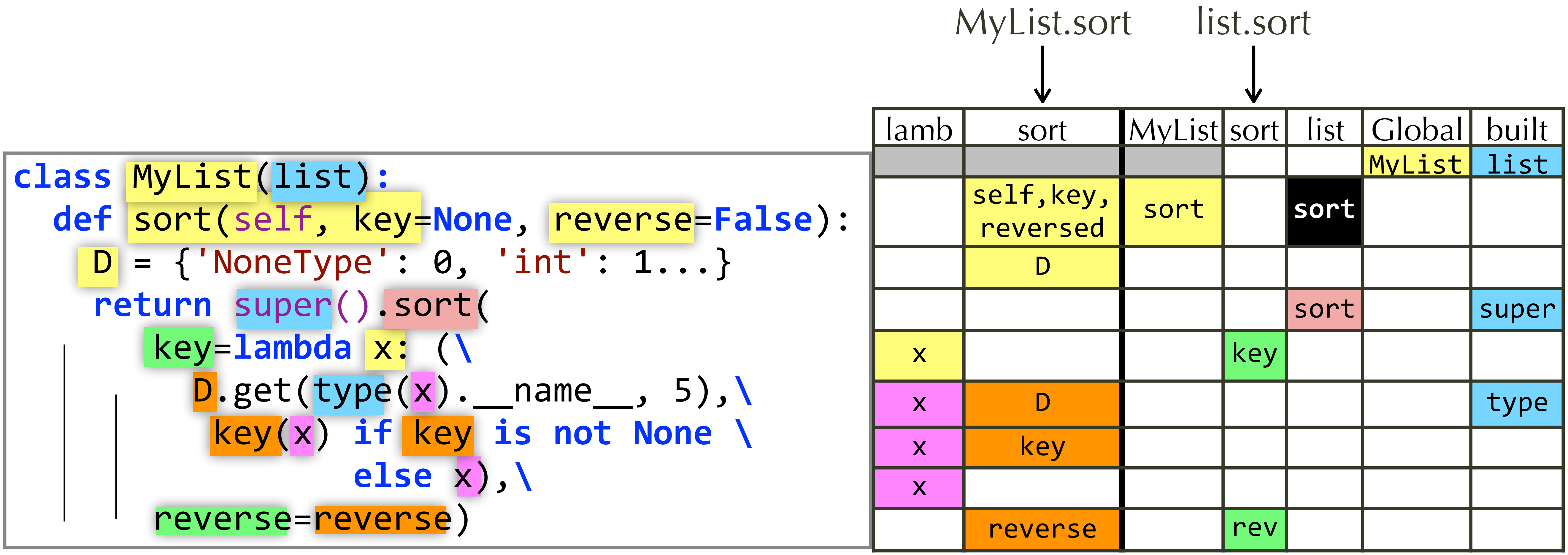
Shadow ⬛

Lookup 🟩 🟥 🟧 🟦

one symbol table per instance

one symbol table for all

Symbol tables for built-in, global, class, method, and instance



```
class MyList(list):
  def sort(self, key=None, reverse=False):
    D = {'NoneType': 0, 'int': 1...}
    return super().sort(
        key=lambda x: (\
          D.get(type(x).__name__, 5),\
           key(x) if key is not None \
                  else x),\
        reverse=reverse)
```

MyList.sort ↓

list.sort ↓

lamb	sort	MyList	sort	list	Global	built
					MyList	list
	self,key, reversed	sort		sort		
	D					
				sort		super
x			key			
x	D					type
x	key					
x						
	reverse		rev			

Define 🟨

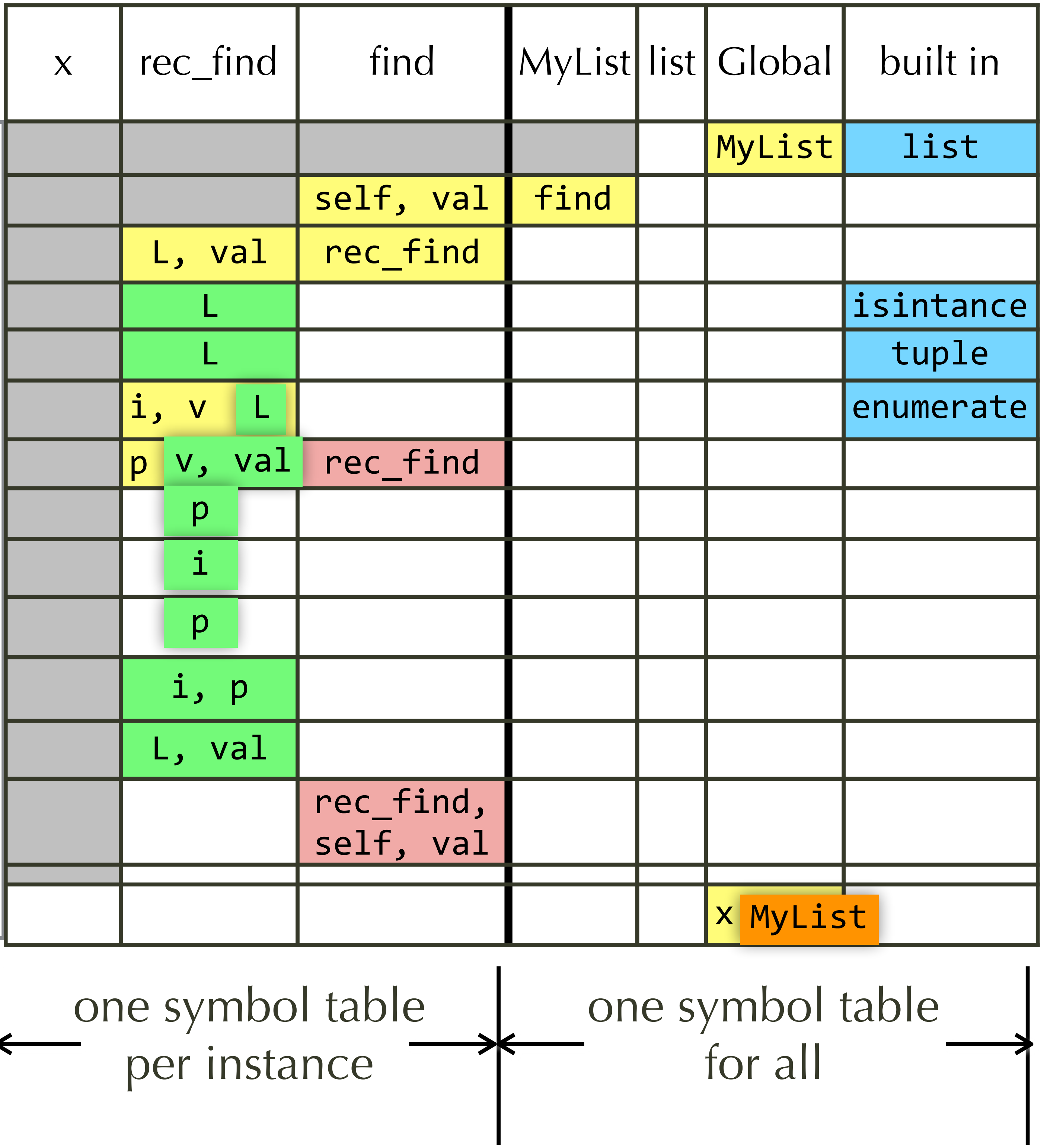
Shadow ⬛

Lookup 🟩🟥🟧🟦

Symbol tables for built-in, global, class, method, inner method, instance

```
class MyList(list):
  def find(self, val):
    def rec_find(L, val):
      if isinstance(L, list) or\
         isinstance(L, tuple):
        for i, v in enumerate(L):
          p = rec_find(v, val)
          if p == True:
            return (i,)
          if p != False:
            return (i,)+p
      return L == val
    return rec_find(self, val)

x = MyList([1, 2, 3])
```



x	rec_find	find	MyList	list	Global	built in
					MyList	list
		self, val	find			
	L, val	rec_find				
	L					isintance
	L					tuple
	i, v L					enumerate
	p v, val	rec_find				
	p					
	i					
	p					
	i, p					
	L, val					
		rec_find, self, val				
					x MyList	

one symbol table per instance

one symbol table for all

Define 🟨

Lookup 🟩 🟥 🟧 🟦

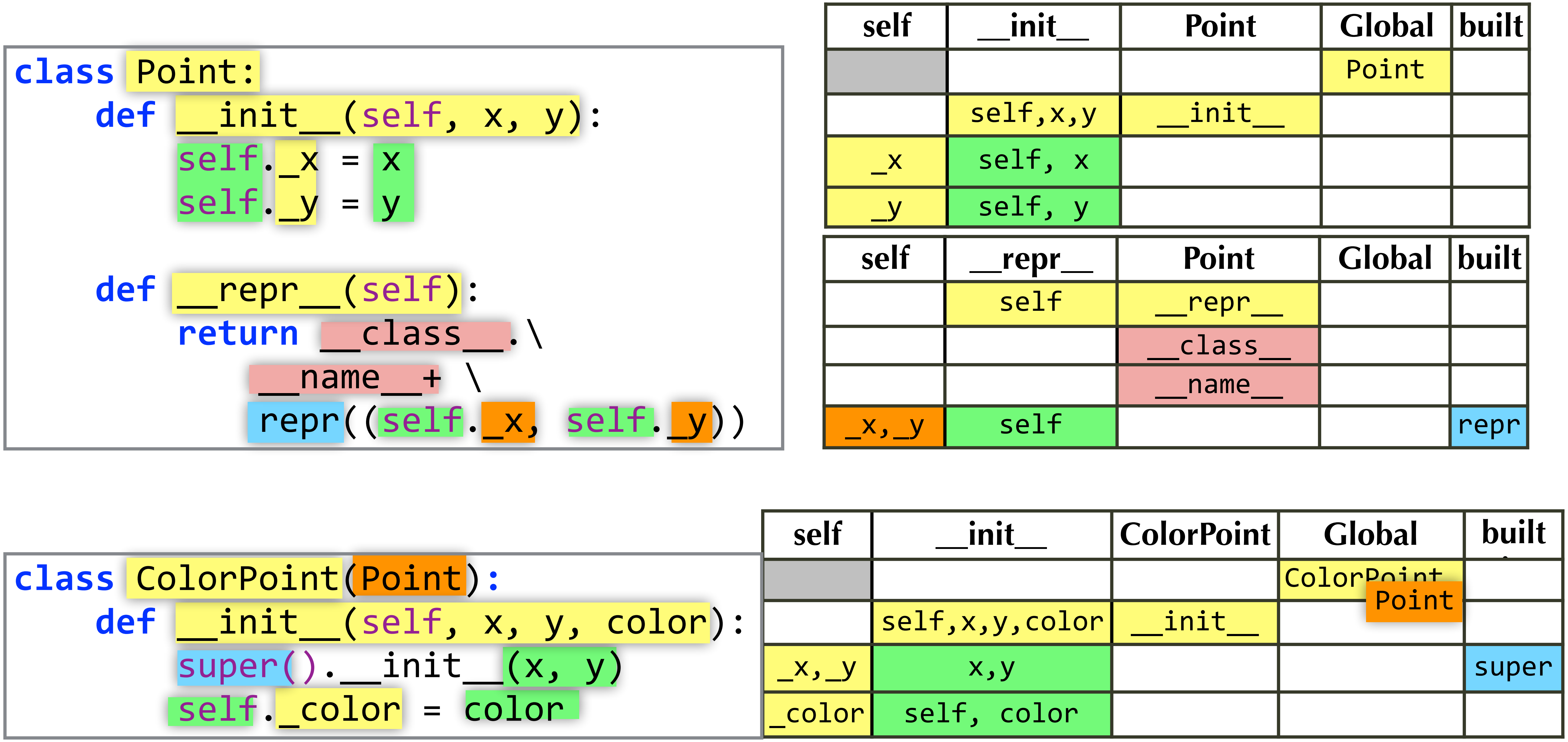
Example: ColorPoint class

- inherit from Point
  - add attribute of color, in addition to (x, y) coordinate
  - `super().__init__` to call the constructor as superclass instance!

```
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def __repr__(self):
        return __class__.__name__+ \
            repr((self._x, self._y))
class ColorPoint(Point):
    def __init__(self, x, y, color):
        super().__init__(x, y)
        self._color = color
    def __repr__(self):
        return __class__.__name__+\
             repr((self._x, self._y, \
                   self._color))
```

```
>>> p = Point(2, 3)
>>> p
Point(2, 3)
>>> q = ColorPoint(4, 5, 'black')
>>> q
ColorPoint(4, 5, 'black')
```

Symbol tables for built-in, global, class, method, and instance



```
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def __repr__(self):
        return __class__.\
          __name__+ \
          repr((self._x, self._y))
```

self	__init__	Point	Global	built
			Point	
	self,x,y	__init__		
_x	self, x			
_y	self, y			

self	__repr__	Point	Global	built
	self	__repr__		
		__class__		
		__name__		
_x,_y	self			repr

```
class ColorPoint(Point):
    def __init__(self, x, y, color):
        super().__init__(x, y)
        self._color = color
```

self	__init__	ColorPoint	Global	built
			ColorPoint Point	
	self,x,y,color	__init__		
_x,_y	x,y			super
_color	self, color			

Define 🟨

Lookup 🟩🟥🟧🟦

Concept of Polymorphism

- poly = many (多), morph = shape (形)
- in programming language => mean many "models" (多型)
  - more specifically, same name that encompasses many different models
- synonym: **overloading**
- example: `str(`*x*`)` function
  - *x* is `int` => `int.__str__(`*x*`)`
  - *x* is `float` => `float.__str__(`*x*`)`
  - *x* is `ColorPoint` => `ColorPoint.__str__(`*x*`)`

Polymorphism in Python

- overloaded functions:
  - `str()`, `repr()`, `len()`, `super()`, ... can be called on objects of different classes
- method invocation
  - you can call a method on any object as long as the method name is found
  - e.g., `.index(`*k*`)` method can be called on `list`, `tuple`, etc
- Operator overloading
  - e.g., `+` is "add" for numbers, "concatenate" for sequences, ..

Operator overloading

- Operators
  - `+ - * / % << >> & ^ | >= <= == != [ ]`
  - Plus keyword or function like: **`in`**, **`del`**, `abs()`, `divmod()`,
- Examples of overloading
  - *str* `+` *str* is concatenation
  - *str* `*` *num* is for duplicating string
  - *str* `%` *tuple* is for string formatting
- Python mechanism: special methods
  - can define for your own class, not just built-in type

Operator vs. method syntax

- Operator syntax
  - A `+` B
  - A `-` B
  - A `*` B
  - A `/` B
  - A `%` B
- Method syntax
  - A`.__add__(`B`)`
  - A`.__sub__(`B`)`
  - A`.__mul__(`B`)`
  - A`.__truediv__(`B`)`
  - A`.__mod__(`B`)`

Special methods for operators from arithmetic and bit syntax

special method	operator syntax	
`A.__add__(B)`	`A + B`	plus
`A.__sub__(B)`	`A - B`	minus
`A.__mul__(B)`	`A * B`	times
`A.__matmul__(B)`	`A @ B`	matrix mult.
`A.__truediv__(B)`	`A / B`	true-divide
`A.__floordiv__(B)`	`A // B`	floor-divide
`A.__mod__(B)`	`A % B`	modulo
`A.__pow__(B)`	`A ** B`	exponentiation
`A.__lshift__(B)`	`A << B`	left shift
`A.__rshift__(B)`	`A >> B`	right shift
`A.__and__(B)`	`A & B`	bitwise and
`A.__or__(B)`	`A | B`	bitwise or
`A.__xor__(B)`	`A ^ B`	bitwise xor

Can apply method syntax on built-in types!

```
>>> x = 2
>>> y = 3
>>> x + y         # regular operator syntax
5
>>> x.__add__(y)  # method syntax
5
>>> x.__mul__(5)  # same as 2 * 5
10
```

- however, be careful with syntax clash

```
>>> 2.__add__(3)         # python parser tries to parse float! 2.
  File "<stdin>", line 1
    2.__add__(3)
            ^
SyntaxError: invalid syntax
>>> 2..__add__(3)  # first dot is float "2.", ".__add__" is method
5.0
>>> (2).__add__(3)  # protect (2) if want integer 2
5
```

Example: Vector class

- like a list, but allow operation

```
>>> x = Vector(1, 2, 3)
>>> y = Vector(7, 4, 1)
>>> x + y
Vector(8, 6, 4)
>>> y - x
Vector(6, 2, -2)
>>> len(x)
3
>>> x += Vector(2, 4, 6)
>>> x
Vector(3, 6, 9)
>>> abs(x)
11.224972160321824
>>> x * y
Vector(14, 16, 6)
>>> 2 * y
Vector(14, 8, 2)
```

Example: Vector class

- like a list, but allow operation

```
>>> x = Vector(1, 2, 3)
>>> y = Vector(7, 4, 1)
>>> x + y
Vector(8, 6, 4)
>>> y - x
Vector(6, 2, -2)
```

```
import operator as op
class Vector:
    def __init__(self, *v):
        self._v = list(v) # comes in as tuple, convert to list
    def __repr__(self):
        return __class__.__name__+repr(tuple(self._v))
    def __add__(self, right):
        return Vector(*map(op.add, self._v, right._v))
        # op.add is same as lambda x,y: x+y
    def __sub__(self, right):
        return Vector(*map(op.sub, self._v, right._v))
```

How overloading works

```
>>> x = Vector(1, 2, 3)
>>> y = Vector(7, 4, 1)
>>> x + y
Vector(8, 6, 4)
>>> y - x
Vector(6, 2, -2)
```

- `x + y` is another way of saying `x.__add__(y)`
- result is another Vector, so need to call the constructor whose params are the pairwise sums

```
class Vector:
    ...
    def __add__(self, right):
        return Vector(*map(op.add, self._v, right._v))
```

Other overloadable operators

- Augmenting (in-place) assignments
  - e.g., A `+=` B
- Comparison, membership, indexing
  - e.g., A `>` B, A `==` B, B `in` A, A`[`i`]`
- unary operator and built-in functions
  - `-`A, `~`A, `abs(`A`)`, `len(`A`)`, …

Special methods for *in-place* augmented assignment

special method	operator syntax	
`A.__iadd__(B)`	`A += B`	
`A.__isub__(B)`	`A -= B`	
`A.__imul__(B)`	`A *= B`	
`A.__imatmul__(B)`	`A @= B`	matrix mult.
`A.__itruediv__(B)`	`A /= B`	
`A.__ifloordiv__(B)`	`A //= B`	
`A.__imod__(B)`	`A %= B`	
`A.__ipow__(B)`	`A **= B`	
`A.__ilshift__(B)`	`A <<= B`	
`A.__irshift__(B)`	`A >>= B`	
`A.__iand__(B)`	`A &= B`	
`A.__ior__(B)`	`A |= B`	
`A.__ixor__(B)`	`A ^= B`	

On augmenting assignment

```
>>> x = Vector(1, 2, 3)
>>> y = Vector(7, 4, 1)
>>> x += y
>>> x
Vector(8, 6, 4)
```

- Python knows how to do += even though we did not define `__iadd__()`. Why?
  - Because Python tries the next best match as x = x + y
  - However, this discards old x, sets x to new Vector => more garbage, less efficient

__add__ vs. __iadd__

```python
class Vector:
    ...
    def __add__(self, right):
        return Vector(*tuple(map(op.add, self._v, right._v)))

    def __iadd__(self, right):
        self._v[:] = map(op.add, self._v, right._v)
        return self
```

- x += y without `__iadd__`
  - Python calls `__add__` followed by `=` to mimic iadd. identity of x changes after x += y
- with `__iadd__`
  - `__iadd__` returns self, so x = modified x (same identity)

Special methods for indexing/ slicing and comparison operators

special method	operator syntax	
`A.__contains__(B)`	`B in A`	
`A.__getitem__(i)`	`A[i]`	incl. slicing
`A.__setitem__(i, B)`	`A[i] = B`	ternary
`A.__delitem__(i)`	`del A[i]`	incl. slicing
`A.__lt__(B)`	`A < B`	
`A.__le__(B)`	`A <= B`	
`A.__gt__(B)`	`A > B`	
`A.__ge__(B)`	`A >= B`	
`A.__eq__(B)`	`A == B`	
`A.__ne__(B)`	`A != B`	

Special methods for unary and function-like operators

special method	operator syntax	
`A.__inv__()`	`~A`	unary
`A.__neg__()`	`-A`	unary
`A.__pos__()`	`+A`	unary
`A.__abs__()`	`abs(A)`	built-in
`A.__round__(npos)`	`round(A, npos)`	built-in
`A.__trunc__()`	`math.trunc(A)`	math library
`A.__floor__()`	`math.floor(A)`	math library
`A.__ceil__()`	`math.ceil(A)`	math library
`A.__bool__()`	`bool(A)`	constructor
`A.__bytes__()`	`bytes(A)`	constructor
`A.__int__()`	`int(A)`	constructor
`A.__complex__()`	`complex(A)`	constructor
`A.__float__()`	`float(A)`	constructor

__getitem__ and __setitem__

- Want to apply operators similar to lists

```
>>> x = Vector(1, 2, 3)
>>> y = Vector(7, 4, 1)
>>> z = Vector(*range(1, 9))
>>> z
Vector(1, 2, 3, 4, 5, 6, 7, 8)
>>> z[2:5]                 # calls z.__getitem__(slice(2,5))
Vector(3, 4, 5)
>>> z[2:5] = (10, 11, 12)  # z.__setitem__(slice(2,5),(10,11,12))
>>> z
Vector(1, 2, 10, 11, 12, 6, 7, 8)
>>> z[7]                   # calls z.__getitem__(7)
8
>>> z[2:5] = y             # calls z.__setitem__(slice(2,5), y)
>>> z
Vector(1, 2, 7, 4, 1, 6, 7, 8)
```

__getitem__ and __setitem__

```
class Vector:
    ...
    def __len__(self):
        return len(self._v)
    def __getitem__(self, i):
        if type(i) == int:    # index
            return self._v[i]
        elif type(i) == slice:
            return Vector(*(self._v[i]))
        else:
            raise TypeError(type(i).__name__+' unsupported')
    def __setitem__(self, i, v):
        if type(i) == int:
            self._v[i] = v
        elif type(i) == slice:
            self._v[i] = v if not isinstance(v, Vector) \
                    else v._v[:]
        else:
            raise TypeError(type(i).__name__+' unsupported')
```

Example of operator precedence and associativity

- 2 + 5 * 3 ** 4 - 6
  = (2 + (5 * (3 ** 4))) - 6
  = (2 + (5 * (81))) - 6
  = (2 + 405) - 6
  = 407 - 6
  = 401

- 4 ** 3 ** 2 # right associative
  = 4 ** (3 ** 2)
  = 4 ** 9 = 262144, NOT (4**3) ** 2 = 4096

Operator Precedence

prec	operator	comment	associativity
highest	(x) [x] { x }	parentheses, brackets, braces	inside-out (unary)
	f (x), A[L:U], A[i]	function call, slice, index	left (binary)
	obj.attr	attribute access	left (binary)
	a ** b	exponent	right (binary)
	~x, -x, +x	bit invert, negate, uplus	right (unary)
	a*b, a / b, a // b, a % b	times, tdiv, fdiv, mod	left (binary)
	a + b, a - b	plus, minus	left
	a << b, a >> b	bit shift left or right	left
	a & b, a ^ b, a | b	bitwise and > xor > or	left
	a == b, !=, >, <, >=, <= in, not in, is, is not	comparison operators, membership, identity	left (binary)
	not x, x and y, x or y	logical not > and > or	right (unary), left (binary)
lowest	=, +=, -=, A[i]=B,	assignment operator	right

Operator precedence and associativity

- When in doubt, put subexpressions in ()
  - especially when mixing arithmetic and logic
  - also parenthesize to ensure associativity
- Especially right-associative
  - unary is considered right-associative
- Python special methods
  - invoked according to operator precedence and associativity

Notes about Operator Overloading

- @ is not a built-in operator!
  - class must define `__matmul__` to use it
  - meaning is entirely user defined!
- **`and`**, **`or`**, **`not`** (logic) can't be overloaded
  - No good reason to overload them!
  - instead, overload the `__bool__` for logic meaning
- **`=`** (regular assignment) can't be overloaded
  - however, can overload **in-place assignment**
    i.e. *augmenting assignment* += and `__setitem__` i.e.,(A[i] = B)

__str__ vs __repr__ for string type

```
>>> s = 'hello'.__str__()
>>> r = 'hello'.__repr__()
>>> r   # the repr string is the "python source" for the string
"'hello'"  # so it includes the quotation marks!!
>>> s      # but the str() is more like a type converter
'hello'    # so s is only 5-letter hello
>>> print(r)  # print the actual content of the string
'hello'       # the repr includes the quote marks!
>>> print(s)  # but str is just the letters, no quote marks
hello
>>> list(s)
['h', 'e', 'l', 'l', 'o']
>>> list(r)
["'", 'h', 'e', 'l', 'l', 'o', "'"]
```

built-in functions: repr() and str()

- call the respective __repr__ and __str__

```
>>> s = str('hello')  # same as 'hello'.__str__()
>>> r = repr('hello') # same as 'hello'.__repr__()
>>> r
"'hello'"
>>> s
'hello'
```

- `repr()` and `str()` are the preferred way to invoke an object's __repr__ and __str__

Summary: operator overloading

- Operator syntax:
  - more concise, may borrow intuition from built-in operators
- Python implementation
  - defining special methods for operators
    (not all special methods are operators though)
  - Follows same precedence & associativity