Operator Overloading

Presented By,
Asst. Prof. Shruti Deshmukh

General concepts

- Operator overloading lets classes intercept normal Python operations.
- Classes can overload all Python expression operators.
- Classes can also overload built-in operations such as printing, function calls, attribute access, etc.
- Overloading makes class instances act more like built-in types.
- Overloading is implemented by providing specially named methods in a class.

Simple example

```
class Number:
    def __init__(self, start):
        self.data = start
    def __sub__(self, other):
        return Number(self.data - other)

>>> from number import Number
>>> X = Number(5)            # Number.__init__(X, 5)
>>> Y = X - 2                # Number.__sub__(X, 2)
>>> Y.data                   # Y is new Number instance
3
```

Common operator overloading methods

__init__	Constructor	Object creation: X = Class(args)
__del__	Destructor	Object reclamation of X
__add__	Operator +	X + Y, X += Y if no __iadd__
__or__	Operator	(bitwise OR)	X	Y, X |= Y if no __ior__
__repr__, __str__	Printing, conversions	print(X), repr(X), str(X)
__call__	Function calls	X(*args, **kargs)
__getattr__	Attribute fetch	X.undefined
__setattr__	Attribute assignment	X.any = value
__delattr__	Attribute deletion	del X.any
__getattribute__	Attribute fetch	X.any
__getitem__	Indexing, slicing, iteration	X[key], X[i:j], for loops and other iterations if no __iter__
__setitem__	Index and slice assignment	X[key] = value, X[i:j] = iterable
__delitem__	Index and slice deletion	del X[key], del X[i:j]

Common operator overloading methods

__len__ Length len(X), truth tests if no __bool__ __bool__ Boolean tests bool(X), truth tests

__lt__, __gt__, __le__, __ge__, __eq__, __ne__
Comparisons X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y

__radd__ Right-side operators Other + X

__iadd__ In-place augmented operators X += Y (or else __add__)

__iter__, __next__ Iteration contexts I=iter(X), next(I); for loops, in if no __contains__, all comprehensions, map(F,X), others

__contains__ Membership test item in X (any iterable)

__index__ Integer value hex(X), bin(X), oct(X), O[X], O[X:]

__enter__, __exit__ Context manager (Chapter 34) with obj as var:

__get__, __set__,

__delete__ Descriptor attributes (Chapter 38) X.attr, X.attr = value, del X.attr

__new__ Creation (Chapter 40) Object creation, before __init__

Indexing and Slicing: __getitem__ and __setitem__

```
class Indexer:
    def __getitem__(self, index):
        return index ** 2
```

>>> X = Indexer() >>> X[2] *# X[i] calls X.__getitem__(i)*

4

>>> for i in range(5):

print(X[i], end=' ') *# Runs __getitem__(X, i)*
each time

0 1 4 9 16

Indexing and Slicing: __getitem__ and __setitem__

```
>>> class Indexer:
        data = [5, 6, 7, 8, 9]
        def __getitem__(self, index):     # Called for index or slice
            print('getitem:', index)
            return self.data[index]       # Perform index or slice
>>> X = Indexer()
>>> X[0]                # Indexing sends __getitem__ an integer
getitem: 0      #5
>>> X[1]
getitem: 1      #6
>>> X[-1]
getitem: −1     #9
```

Indexing and Slicing: __getitem__ and __setitem__

```
>>> X[2:4]              # Slicing sends __getitem__ a slice object
getitem: slice(2, 4, None)       #[7, 8]
>>> X[1:]
getitem: slice(1, None, None)     #[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, −1, None)   #[5, 6, 7, 8]
>>> X[::2]
getitem: slice(None, None, 2)     #[5, 7, 9]
```

```
class IndexSetter:
    def __setitem__(self, index, value):    # Intercept index or slice assignment
        ...
        self.data[index] = value                # Assign index or slice
```

Code one, get a bunch free

```
class StepperIndex:
    def __getitem__(self, i):
        return self.data[i]

X = StepperIndex()        # X is a StepperIndex object
X.data = "Spam“
for item in X:
    print(item, end=' ')
  # for loops call __getitem__    for indexes items 0..N
      #S p a m
```

Code one, get a bunch free

The **in** membership test, list comprehensions, the map built-in, list and tuple assignments, and type constructors will also call __getitem__ automatically, if it's defined:

```
>>> 'p' in X                    # All call __getitem__ too      True
>>> [c for c in X]              # List comprehension    ['S', 'p', 'a', 'm']
>>> list(map(str.upper, X))     # map calls (use list() in 3.X)
                                #['S', 'P', 'A', 'M']
>>> (a, b, c, d) = X            # Sequence assignments
>>> a, c, d                     #('S', 'a', 'm')
>>> list(X), tuple(X), ''.join(X)     # And so on...
                  #(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')
```

Iterable Objects: __iter__ and __next__

Today, all iteration contexts in Python will try the **__iter__** method first, before trying **__getitem__**. That is, they prefer the iteration protocol to repeatedly indexing an object; only if the object does not support the iteration protocol is indexing attempted instead. Generally speaking, you should prefer **__iter__** too—it supports general iteration contexts better than **__getitem__** can.

Technically, iteration contexts work by passing an iterable object to the **iter** built-in function to invoke an **__iter__** method, which is expected to return an iterator object. If it's provided, Python then repeatedly calls this iterator object's **__next__** method to produce items until a **StopIteration** exception is raised.

User-Defined Iterables

```
class Squares:
    def __init__(self, start, stop):
        self.value = start - 1
        self.stop  = stop
    def __iter__(self):    # Get iterator object on iter
        return self
    def __next__(self):     # Return a square on each iteration
        if self.value == self.stop:     # Also called by next built-in
            raise StopIteration
        self.value += 1
        return self.value ** 2

for i in Squares(1, 5):          # for calls iter, which calls __iter__
    print(i, end=' ')            # Each iteration calls __next__
1 4 9 16 25
```

Single versus multiple scans

Because the current **Squares** class's **__iter__** always returns **self** with just one copy of iteration state, it is a one-shot iteration; once you've iterated over an instance of that class, it's empty. Calling **__iter__** again on the same instance returns self again, in whatever state it may have been left. You generally need to make a new iterable instance object for each new iteration:

>>>X = Squares(1, 5)

>>> [n for n in X] *# Exhausts items: __iter__ returns self*
[1, 4, 9, 16, 25]

>>> [n for n in X] *# Now it's empty: __iter__ returns same self*
[]

3.X's __index__ Is Not Indexing!

Don't confuse the (perhaps unfortunately named) **__index__** method in Python 3.X for index interception—this method <u>returns an integer value for an instance</u> when needed and is used by built-ins that convert to digit strings (and in retrospect, might have been better named __asindex__):

```
class C:
    def __index__(self):
        return 255


>>> X = C()
>>> hex(X)           # Integer value '0xff'
>>> bin(X)       # '0b11111111'
>>> oct(X)       #'0o377'
```

Membership: __contains__, __iter__, and __getitem__

Operator overloading is often *layered*: classes may provide specific methods, or more general alternatives used as fallback options. For example: boolean tests try a specific **__bool__** first (to give an explicit True/False result), and if it's absent fall back on the more general **__len__** (a nonzero length means True).

In the iterations domain, classes can implement the **in** membership operator as an iteration, using either the **__iter__** or **__getitem__** methods. To support more specific membership classes may code a **__contains__** method—when present, this method is preferred over **__iter__**, which is preferred over **__getitem__.** The **__contains__** method should define membership as applying to keys for a mapping (and can use quick lookups), and as a search for sequences.

```
class Iters:
    def __init__(self, value):
        self.data = value
   def __getitem__(self, i):              # Fallback for iteration
        print('get[%s]:' % i, end='')        # Also for index, slice
        return self.data[i]
   def __iter__(self):                   # Preferred for iteration
        print('iter=> ', end='')            # Allows only one active iterator
        self.ix = 0
        return self
   def __next__(self):
        print('next:', end='')
        if self.ix == len(self.data):     raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item
    def __contains__(self, x):            # Preferred for 'in'
        print('contains: ', end='')
        return x in self.data

X = Iters([1, 2, 3, 4, 5])       # Make instance
print(3 in X)                   # Membership    for i in X:                # for loops
print(i, end=' | ')
print()
 print([i ** 2 for i in X])       # Other iteration contexts
print( list(map(bin, X)) )
   I = iter(X)                  # Manual iteration (what other contexts do)
while True:       try:          print(next(I), end=' @ ')       except StopIteration:          break
```

Attribute Access: __getattr__ and __setattr__

The **__getattr__** method catches attribute references and is called with the attribute name as a string whenever you try to qualify an instance with an undefined (nonexistent) attribute name. It is not called if Python can find the attribute using its inheritance tree search procedure. It's commonly used to delegate calls to embedded (or "wrapped") objects from a proxy controller object. This method can also be used to adapt classes to an interface, or add accessors for data attributes after the fact—logic in a method that validates or computes an attribute after it's already being used with simple dot notation.

Attribute Access: __getattr__ and __setattr__

```
class Empty:
    def __getattr__(self, attrname):        # On self.undefined
        if attrname == 'age':
            return 40
        else:           raise AttributeError(attrname)

>>> X = Empty()
>>> X.age 40
>>> X.name
 ...error text omitted...
AttributeError: name
```

Attribute Access: __getattr__ and __setattr__

__setattr__ intercepts all attribute assignments: **self.attr = value** is **self.__setattr__('attr', value).** Like **__getattr__** this allows your class to catch attribute changes, and validate or transform as desired.

*!!!! Assigning to any self attributes within **__setattr__** calls **__setattr__** again, potentially causing an infinite recursion loop.*

To avoid this use **self.__dict__['name'] = x**, not **self.name = x**.

```
class Accesscontrol:
    def __setattr__(self, attr, value):
        if attr == 'age':
        self.__dict__[attr] = value + 10      # Not self.name=val or setattr
      else:           raise AttributeError(attr + ' not allowed')
>>> X = Accesscontrol()
>>> X.age = 40                          # Calls __setattr__
 >>> X.age                              #50
>>> X.name = 'Bob'
...text omitted...
AttributeError: name not allowed
```

Other Attribute Management Tools

- The **__getattribute__** method intercepts all attribute fetches, not just those that are undefined, but when using it you must be more cautious than with __get attr__ to avoid loops.
- The **property** built-in function allows us to associate methods with fetch and set operations on a specific class attribute.
- **Descriptors** provide a protocol for associating **__get__** and **__set__** methods of a class with accesses to a specific class attribute.
- **Slots** attributes are declared in classes but create implicit storage in each instance.

See Chapter 38 Mark Lutz for detailed coverage of all the attribute management techniques.

String Representation: __repr__ and __str__

Why Two Display Methods?

- **__str__** is tried first for the print operation and the **str** built-in function (the internal equivalent of which print runs). It generally should return a user-friendly display.
- **__repr__** is used in all other contexts: for interactive echoes, the **repr** function, and nested appearances, as well as by print and **str** if no **__str__** is present. It should generally return an as-code string that could be used to re-create the object, or a detailed display for developers.

String Representation: __repr__ and __str__

That is, **__repr__** is used everywhere, except by **print** and **str** when a **__str__** is defined. This means you can code a **__repr__** to define a single display format used everywhere, and may code a **__str__** to either support **print** and **str** exclusively, or to provide an alternative display for them.

__repr__ may be best if you want a *single display* for all contexts. By defining both methods, though, you can <u>support different </u>displays in different contexts —for example, an end-user display with **__str__,** and a low-level display for programmers to use during development with **__repr__.** In effect, **__str__** simply overrides **__repr__** for more user-friendly display contexts.

Compare

```
class Printer:
    def __init__(self, val):
        self.val = val
    def __str__(self):        # Used for instance itself
        return str(self.val)  # Convert to a string
result

>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)       # 2 3
#__str__ run when instance printed
# But not when instance is in a list!
>>> print(objs)
[<__main__.Printer object at
0x000000000297AB38>,
<__main__.Printer obj...etc...>]
>>> objs
 [<__main__.Printer object at
0x000000000297AB38>,
<__main__.Printer obj...etc...>]
```

```
class Printer:
    def __init__(self, val):
        self.val = val
    def __repr__(self):
        return str(self.val)
# __repr__ used by print if no __str__
# __repr__ used if echoed or nested

>>> objs = [Printer(2), Printer(3)]
>>> for x in objs: print(x)
# No __str__: runs __repr__    2 3
>>> print(objs)
# Runs __repr__, not ___str__     [2, 3]
>>> objs        # [2, 3]
```

Right-Side and In-Place Uses: __radd__ and __iadd__

For every binary expression, we can implement a *left*, *right*, and *in-place* variant.

```
class Adder:
    def __init__(self, value=0):
        self.data = value
    def __add__(self, other):
        return self.data + other
>>> x = Adder(5)
>>> x + 2     #7
>>> 2 + x
```

TypeError: unsupported operand type(s) for +: 'int' and 'Adder'

Right-Side and In-Place Uses: __radd__ and __iadd__

- **__add__**: instance + noninstance
- **__radd__**: noninstance + instance
- **__add__**: instance + instance, triggers **__radd__**

Experiment with different types of operands:

```
class Adder1:
    def __init__(self, val):
        self.val = val
    def __add__(self, other):
        print('add', self.val, other)
        return self.val + other
    def __radd__(self, other):
        print('radd', self.val, other)
        return other + self.val
```

Right-Side and In-Place Uses: __radd__ and __iadd__

To implement **+=** in-place augmented addition, code either an **__iadd__** or an **__add__**. The latter is used if the former is absent.

```
class Number:
    def __init__(self, val):
        self.val = val
    def __iadd__(self, other):     # __iadd__ explicit: x += y
        self.val += other          # Usually returns self
        return self
```

Call Expressions: __call__

```
class Callee:
    def __call__(self, *pargs, **kargs):
        print('Called:', pargs, kargs)


>>> C = Callee()
>>> C(1, 2, 3)                # C is a callable object
Called: (1, 2, 3) {}
>>> C(1, 2, 3, x=4, y=5)
Called: (1, 2, 3) {'y': 5, 'x': 4}
```

Call Expressions: __call__

Intercepting call expression like this allows class instances to emulate the look and feel of things like functions, but also retain state information for use during calls.

```
class Prod:
    def __init__(self, value):
        self.value = value
    def __call__(self, other):
        return self.value * other

>>> x = Prod(2)          # "Remembers" 2 in state
>>> x(3)                 # 3 (passed) * 2 (state)     6
>>> x(4)                 #  8
```

Call Expressions: __call__

More useful example: in GUI

```
class Callback:
    def __init__(self, color):
        self.color = color
    def __call__(self):
        print('turn', self.color)

# Handlers
cb1 = Callback('blue')
cb2 = Callback('green')
B1 = Button(command=cb1)
B2 = Button(command=cb2)
# Events
cb1()
cb2()
```

Closure equivalent

```
def callback(color):
    def oncall():
        print('turn', color)
        return oncall
cb3 = callback('yellow')
cb3()                    # On event: prints 'turn yellow‘

cb4 = (lambda color='red': 'turn ' + color)
# Defaults retain state too
print(cb4())
```

Problems to solve

1. Think of a several sensible situations to overload arithmetic and comparison with classes.
2. Experiment with indexing and slicing operators in classes. Think of reasonable situations when it is useful.
3. Provide your own iterable object. Experiment with different iteration techniques.
4. Provide your own reasonable callable object. Experiment with equivalent closure techniques.