Operator Overloading

Polymorphism

- The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Operator Overloading

- The concept of overloading is also a branch of polymorphism.
- In operator overloading different operators have different implementations depending on their arguments.
- Python operators work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings. This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

First Example

add.py - C:/Documents and Settings/admin/Desktop/intro-python/examples/opover/add.py (3.4.4)

File Edit Format Run Options Window Help

```python
class Number:
    def __init__(self, start):              # On Number(start)
        self.data = start
    def __sub__(self, other):              # On instance - other
        return Number(self.data - other)    # Result is a new instance
    def __add__(self, other):              # On instance + other
        return Number(self.data + other)    # Result is a new instance
    def __str__(self):
        return str(self.data)


X = Number(5)              # Number.__init__(X, 5)
Y = X - 2                  # Number.__sub__(X, 2)
print(Y.data)
Z= Y + 4
print(Z)

#output:
   3
   7
```

Common operator overloading methods

Method	Implements	Called for
`__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 (named __nonzero__ in 2.X)
__lt__, __gt__, __le__, __ge__, __eq__, __ne__	Comparisons	X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y (or else __cmp__ in 2.X only)
__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 (__next__ is named next in 2.X)
__contains__	Membership test	item in X (any iterable)
__index__	Integer value	hex(X), bin(X), oct(X), O[X], O[X:] (replaces 2.X __oct__, __hex__)
__enter__, __exit__	Context manager	with obj as var:
__get__, __set__, __delete__	Descriptor attributes	X.attr, X.attr = value, del X.attr
__new__	Creation	Object creation, before __init__

Indexing and Slicing

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

X = Indexer()

print(X[5])

for i in range(10):
    print(X[i], end=' ')

#output:
25
0 1 4 9 16 25 36 49 64 81
```

Intercepting Slices

```
class Indexer:
   def __init__(self, range):
        self.data = [0]*range
   def __setitem__(self, index, value): # Called for index or slice
      print('setitem:', index)
      self.data[index]=value
      return self.data[index]     #  Perform index or slice
   def __getitem__(self, index): # Called for index or slice
      print('getitem:', index, ' value: ', end=' ')
      return self.data[index]     #  Perform index or slice

X = Indexer(10)
for i in range(10):
   X[i]=i
X[2]=5; X[5]=7 ; X[9]=5
for i in range(10):
   print(X[i])
print(X[2:6])
print(X[1:])
print(X[:-1])
print(X[::2])
```

Intercepting Slices

```
setitem: 0
setitem: 1
setitem: 2
setitem: 3
setitem: 4
setitem: 5
setitem: 6
setitem: 7
setitem: 8
setitem: 9
setitem: 2
setitem: 5
setitem: 9
getitem: 0  value:  0
getitem: 1  value:  1
getitem: 2  value:  5
getitem: 3  value:  3
getitem: 4  value:  4
getitem: 5  value:  7
getitem: 6  value:  6
getitem: 7  value:  7
getitem: 8  value:  8
getitem: 9  value:  5
getitem: slice(2, 6, None)  value:  [5, 3, 4, 7]
getitem: slice(1, None, None)  value:  [1, 5, 3, 4, 7, 6, 7, 8, 5]
getitem: slice(None, -1, None)  value:  [0, 1, 5, 3, 4, 7, 6, 7, 8]
getitem: slice(None, None, 2)  value:  [0, 5, 4, 6, 8]
```

```python
class Indexer:
    def __init__(self, rang):
        self.data = [0]*rang
        for i in range(rang):
            self.data[i]=i
    def __setitem__(self, index, value):
        print('setitem:', index)
        self.data[index]=value
        return self.data[index]
    def __getitem__(self, index):
        if isinstance(index, int):
            print('getitem:', index, ' value: ', end=' ')
        else:
            print('slicing', index.start, index.stop, index.step,
                  ' value: ', end=' ')
        return self.data[index]    # Perform index or slice

X = Indexer(10)
X[2]=5; X[5]=7 ; X[9]=5
print(X[2:6])
print(X[1:])
print(X[:-1])
print(X[::2])

#output:
setitem: 2
setitem: 5
setitem: 9
slicing 2 6 None  value:  [5, 3, 4, 7]
slicing 1 None None  value:  [1, 5, 3, 4, 7, 6, 7, 8, 5]
slicing None -1 None  value:  [0, 1, 5, 3, 4, 7, 6, 7, 8]
slicing None None 2  value:  [0, 5, 4, 6, 8]
```

__index__ Is Not Indexing

- On a related note, don't confuse the (perhaps unfortunately named) __index__ method in Python 3.X for index interception—this method returns an integer value for an instance when needed and is used by built-ins that convert to digit strings (and in retrospect, might have been better named __asindex__)

__index__ Is Not Indexing

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

X = C()
print(hex(X))
print(bin(X))
print(oct(X))
print(('A' * 256)[255])
print(('B' * 256)[X])
print(('D' * 256)[X:])

#output:
0xff
0b11111111
0o377
A
B
D
```

Iterable : __iter__ and __next__

- 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. A next built-in function is also available as a convenience for manual iterations—next(I) is the same as I.__next__().

Iterable : __iter__ and __next__

```
class Squares:
    def __init__(self, start, stop): # Save state when created
        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__

#output:
1 4 9 16 25
```

```
class Squares:
    def __init__(self, start, stop):     # Save state when create
        self.value = start - 1
        self.stop  = stop
    def __iter__(self):                  # Get iterator object on
        return self
    def __next__(self):                  # Return a square on eac
        if self.value == self.stop:      # Also called by next bu
            raise StopIteration
        self.value += 1
        return self.value ** 2

for i in Squares(1, 5):              # for calls iter, which call
   print(i, end=' ')                 # Each iteration calls __nex
print()
X=Squares(1, 5)
I=iter(X)
print(next(I)); print(next(I))
print(next(I)); print(next(I))
print(next(I)); print(next(I))

#output:
1 4 9 16 25
1
4
9
16
25
Traceback (most recent call last):
  File "C:/Documents and Settings/admin/Desktop/intro-python/exa
    print(next(I)); print(next(I))
  File "C:/Documents and Settings/admin/Desktop/intro-python/exa
    raise StopIteration
StopIteration
```

```
class Squares:
    def __init__(self, start, stop):    # Save state when created
        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 iterat
        if self.value == self.stop:     # Also called by next built-in
            raise StopIteration ('There is no item')
        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__
print()
X=Squares(1, 5)
I=iter(X)
try:
    print(next(I)); print(next(I))
    print(next(I)); print(next(I))
    print(next(I)); print(next(I))
except StopIteration as err:
    print(err)

#output:
1 4 9 16 25
1
4
9
16
25
There is no item
```

Attribute : __getattr__ , __setattr__

```
class Accesscontrol:
    def __getattr__(self, attr):
        if attr == 'age':
            self.__dict__[attr]=0
            return self.__dict__[attr]
        else:
            raise AttributeError('There is not ' + attr)
    def __setattr__(self, attr, value):
        if attr == 'age':
            self.__dict__[attr] = value + 10
        else:
            raise AttributeError(attr + ' not allowed')

X = Accesscontrol()
try:
    X.age = 40   # Calls __setattr__
    print(X.age)
    print(X.name)
except AttributeError as err:
    print(err)


#output:
50
There is not name
```

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

- Our next group of overloading methods extends the functionality of binary operator methods such as __add__ and __sub__ (called for + and -), which we’ve already seen. As mentioned earlier, part of the reason there are so many operator overloading methods is because they come in multiple flavors—for every binary expression, we can implement a *left, right,* and *in-place* variant. Though defaults are also applied if you don’t code all three, your objects’ roles dictate how many variants you’ll need to code.

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

```
class Adder:
   def __init__(self, value=0):
      self.data = value
   def __add__(self, other):
      return self.data + other

x = Adder(5)
print(x + 2)
#print(2 + x)    # Error

#output:
7
```

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

```
class Adder:
    def __init__(self, value=0):
        self.data = value

    def __add__(self, other):
        print('add', self.data, other, end='=')
        return self.data + other

    def __radd__(self, other):
        print('radd', other, self.data, end='=')
        return other + self.data

x = Adder(5)
print(x + 2)
print(2 + x)

#output:
add 5 2=7
radd 2 5=7
```

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

```
class Adder:
    def __init__(self, value=0):
        self.data = value

    def __add__(self, other):
        print('add', self.data, other, end='=')
        return self.data + other

    __radd__ = __add__    # Alias: cut out the middleman


x = Adder(5)
print(x + 2)
print(3 + x)

#output:

add 5 2=7
add 5 3=8
```

```
class Adder:
    def __init__(self, value=0):
        self.data = value

    def __add__(self, other):
        print('add', self.data, other, end='=')
        return self.data + other

    __radd__ = __add__    # Alias: cut out the middleman

    def __iadd__(self, other):  # __iadd__ explicit: x += y
        self.data += other        # Usually returns self
        return self
    def __str__(self):
        return(str(self.data))

x = Adder(5)
x+=1
print(x)

y=Adder([1])
y+=[2]
y+=[3]
print(y)
#output:
6
[1, 2, 3]
```

```python
class Adder:
    def __init__(self, value=0):
        self.data = value

    def __add__(self, other):
        print('add', self.data, 'with', other, end=' = ')
        return self.data + other

    __radd__ = __add__    # Alias: cut out the middleman

    def __add__(self, other):   # __iadd__ explicit: x += y
        return Adder(self.data+other)

    def __str__(self):
        return(str(self.data))

x = Adder(5)
print(3+x)
print(x+4)
x+=5
print(x)

y=Adder([1])
y=[12]+y
print(y)
y+=[2]
y+=[3]
print(y)
```

```
add 5 with 3 = 8
9
10
add [1] with [12] = [1, 12]
[1, 12, 2, 3]
```

Call Expressions: __call__

- On to our next overloading method: the __call__ method is called when your instance is called. No, this isn’t a circular definition—if defined, Python runs a __call__ method for function call expressions applied to your instances, passing along whatever positional or keyword arguments were sent.

Call Expressions: __call__

```
class Callee:
   def __call__(self, *pargs, **kargs):       # Intercept instance calls
      print('Called:', pargs, kargs)         # Accept arbitrary arguments

C = Callee()
C(1, 2, 3)                                    # C is a callable object
C(1, 2, 3, x=4, y=5)
C(*[1, 2], **dict(c=3, d=4))                      # Unpack arbitrary arguments
C(1, *(2,), c=3, **dict(d=4))                     # Mixed modes

#output:
Called: (1, 2, 3) {}
Called: (1, 2, 3) {'x': 4, 'y': 5}
Called: (1, 2) {'d': 4, 'c': 3}
Called: (1, 2) {'d': 4, 'c': 3}
```

Call Expressions: __call__

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

x = Prod(2)        # "Remembers" 2 in state
print(x(3))        # 3 (passed) * 2 (state)

#output:
6
```

Comparisons: __lt__, __gt__

- Our next batch of overloading methods supports comparisons. Classes can define methods to catch all six comparison operators:
  - <,
  - >,
  - <=,
  - >=,
  - ==,
  - !=.
- These methods are generally straightforward to use.

Comparisons: __lt__, __gt__

```
class C:
    def __init__(self, str1, val=0):
        self.data = str1
        self.value=val

    def __gt__(self, other):
        return self.data > other
    def __lt__(self, other):
        return self.data < other

X = C('spam', 10)
print(X > 'ham')
print(X < 'ham')
Y=C('ham', 12)
print(X > Y)

#output:

True
False
True
```

End