6.009: Fundamentals of Programming

Lecture 6: Objects

[Slides, code on course website]
Objects = data + associated operations

• Modularity ⇒ reuse, testability, readability
  – A central tenant of procedural programming

• Python is built using modular objects
  – Strings, dicts, lists, ...
  – Each object encapsulates data
  – Each object provides associated operations
    • read data, add/change/delete data, search data

• Python classes allow programmers to create their own objects
  – Provide easy-to-use functional interface
  – Implementation details are hidden
  – OOP motto: “spare me the details!”
Class: a template for building objects

class Circle:
    # handy constant for circle hacking
    PI = 3.141592653

def __init__(self, x, y, r):
    # set up "private" attribute value.
    self._x = x
    self._y = y
    self._r = r

A class variable: shared by all Circle objects: Circle.PI

self: reference to a particular Circle object

Method: a procedure associated with Circle objects

Convention: attributes starting with "_" are considered private and shouldn’t be accessed externally. "__" ⟵ "name mangling"
Creating an object

• c = Circle(1,2,3)
  – Allocate storage for object
  – Call Circle.__init__(pointer to storage,1,2,3)
  – Return pointer as the value to be stored in variable c

• print(c) ⇒ <__main__.Circle object at 0x103a4df10>

• Urk! Let’s add a method for representing a Circle

  def __str__(self):
      return '<circle r=%g center=(%g,%g)>' % 
      (self._r,self._x,self._y)

  __str__ method is called by Python whenever it needs a
  string representation of an object. (Also see __repr__)

• print(c) ⇒ <circle r=3 center=(1,2)>
Properties: managed attributes

- Add “getter” and “setter” methods to read and write the value of an attribute
- Use property() to tell Python that references to the property should be translated to calls to getter and setter methods

```python
def get_r(self):
    return self._r

def set_r(self, r):
    if r < 0:
        raise ValueError('circle radius must be non-negative')
    self._r = r

# r is the publically accessible radius property
r = property(get_r, set_r)
```

These def’s belong inside the body of the class statement
Synthetic Properties

• You can create properties that don’t correspond to actual attributes of the object

```python
# circumference property
def get_c(self):
    return self._r * self.PI * 2
def set_c(self, c):
    if c < 0:
        raise ValueError("circumference must be non-negative")
    self._r = c/(2*self.PI)
circumference = property(get_c, set_c)

# area property
def get_a(self):
    return self._r * self._r * self.PI
def set_a(self, a):
    if a < 0:
        raise ValueError("area must be non-negative")
    self._r = math.sqrt(a/self.PI)
area = property(get_a, set_a)
```
Properties in action

```python
print(c, c.r, c.circumference)
⇒ <circle r=4 center=(1,2)> 4 25.132741224

c.circumference = 2*Circle.PI*6
print(c, c.area)
⇒ <circle r=6 center=(1,2)> 113.097335508

c.area = 25*Circle.PI
print(c)
⇒ <circle r=5 center=(1,2)>
```

Python calls Circle.set_r(c,4)
Python calls Circle.get_r(c)
class Drawing:
    def __init__(self, init=[]):
        self._components = init[:]

    def append(self, component):
        self._components.append(component)

    def svg(self):
        # any component that supports the SVG method
        # can be part of a Drawing!
        return '
'.join(c.svg() for c in self._components)

Be hygienic: make copies of mutable args

Note: generator expression
Add Drawing Support to Circles

class Circle:
   ...
   # drawing support
def svg(self):
      return '<circle r="%g" cx="%g" cy="%g"/>
(\self._r, \self._x, \self._y)

```
d = Drawing([Circle(1,2,3), Circle(4,5,6)])
d.append(Circle(7,8,9))
print(d.svg())
⇒ <circle r="3" cx="1" cy="2"/>
   <circle r="6" cx="4" cy="5"/>
   <circle r="9" cx="7" cy="8"/>
```

Note that Drawing doesn’t “know” about the component types – add new component types after Drawing was written!
Making an Object Support Iteration

Drawings are a sequence of components, so we’d like to be able to write

```python
for c in d:
    print(c)
```

In Python this for loop expands into

```python
iterator = iter(d)
try:
    c = iterator.next()
    ... body of for loop ...
except StopIteration:
    pass
```

If d is an object, call its `__iter__()` method

So, let’s add an `iter` method to Drawing...
__iter__ method

class Drawing:
    ...
    # iterate over components:
    # support iter() built-in function
    def __iter__(self):
        self._index = 0
        return self  # because class implements next()

    def next(self):
        if self._index == len(self._components):
            raise StopIteration
        self._index += 1
        return self._components[self._index - 1]

In this case, we could have just done the following

def __iter__(self):
    return iter(self._components)
### Binary search tree with dict-like methods

```python
# Binary search tree and associated methods
class BSTree:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```
Inserting (key, val) in BSTree

```python
# insert new value into tree
def insert(self, key, val):
    node = self
    while True:
        if key == node.key:
            node.val = val
            return
        elif key < node.key:
            # key belongs in left subtree
            if node.left is None:
                node.left = BSTree(key, val)
                return
            node = node.left
        else:
            # key belongs in right subtree
            if node.right is None:
                node.right = BSTree(key, val)
                return
            node = node.right
```
Access Data in BSTree

# does tree have a particular key?
def __contains__(self, key):
    node = self
    while node is not None:
        if key == node.key: return True
        node = node.left if key < node.key else node.right
    return False

# find key in tree, return associated value
def get(self, key, default = None):
    node = self
    while node is not None:
        if key == node.key:
            return node.val
        node = node.left if key < node.key else node.right
    return default
BSTree in Action

```python
tree = BSTree(22, "root")
tree.insert(14, "A")
tree.insert(33, "B")
tree.insert(2, "C")
tree.insert(17, "D")
tree.insert(27, "E")
tree.insert(45, "F")
tree.insert(47, "G")

print(17 in tree, 46 in tree)  # True False
print(tree.get(45), tree.get(0,"Bingo!"))  # F Bingo!
```
def print_tree(tree, prefix = ''):
    if tree is None:
        print('%sNone' % prefix)
    else:
        print('%s%s: %s' % (prefix, tree.key, tree.val))
        print_tree(tree.left, prefix + '  ')
        print_tree(tree.right, prefix + '  ')

22: root
14: A
  2: C
    None
    None
17: D
    None
    None
33: B
  27: E
    None
    None
45: F
    None
47: G
    None
    None
Iterating Over Keys in BSTree

```python
# iterator support
def __iter__(self):
    keys = []
    def walk(node):
        if node is None: return
        walk(node.left)
        keys.append(node.key)
        walk(node.right)
    walk(self)
    return iter(keys)
```

But this generates *all* the keys, even if user of iteration doesn’t want them all because the iteration was halted early.
A Better Iterator Implementation

```python
# a more incremental iterator
def __iter__(self):
    if self.left:
        for key in self.left:
            yield key
    yield self.key
    if self.right:
        for key in self.right:
            yield key
```

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed).
A Even Better Iterator Implementation

```python
# a better incremental iterator
def __iter__(self):
    if self.left:
        yield from self.left
    yield self.key
    if self.right:
        yield from self.right

print(iter(tree))
⇒ <generator object BSTree.__iter__ at 0x10f124f10>

for key in tree: print(key)
⇒ 2
   14
   17 ...
   47
```
In-order access

# return in-order list of (key,value) pairs

def items(self):
    items = []
    def walk(node):
        if node is None: return
        walk(node.left)
        items.append((node.key, node.val))
        walk(node.right)
    walk(self)
    return items

print(tree.items())
⇒ [(2, 'C'), (14, 'A'), (17, 'D'), (22, 'root'), (27, 'E'), (33, 'B'), (45, 'F'), (47, 'G')]
In-order access (generator version)

```python
# generate in-order list of (key,value) pairs
def items(self):
    if self.left:
        yield from self.left.items()
    yield (self.key, self.val)
    if self.right:
        yield from self.right.items()
```

```python
print(tree.items())
⇒ <generator object BSTree.items at 0x10f124f10>
```

```python
print(list(tree.items()))
⇒ [(2, 'C'), (14, 'A'), (17, 'D'), (22, 'root'),
   (27, 'E'), (33, 'B'), (45, 'F'), (47, 'G')]
```
Making tree[key] work

Google “python special method names” ⇒ § 3.3 of Python3 Reference

```
# support tree[key]
def __getitem__(self, key):
    if key not in self:
        raise KeyError
    else:
        return self.get(key)

# support tree[key] = val
def __setitem__(self, key, value):
    self.insert(key, value)
```

Testing it out:  
```python
tree[23] = 'skidoo'
print(tree[23])
⇒ skidoo
print(tree[42])
⇒ KeyError
```
Making `del tree[key]` work

Case 1: deleted node has no children

Action: delete node from tree (set parent’s link to None)

Example: `del tree[17]`
Making del tree[key] work

Case 2: deleted node has one child

Action: replace node’s contents with child’s contents
link to child is replaced ⇒ child is deleted

Example: del tree[45]
Making \texttt{del tree[key]} work

Case 3: deleted node has two children

Action: let \( s \) = in-order successor
replace node’s key, \( \text{val} \) with \( s’ \) key, \( \text{val} \)
delete \( s \) from tree

Example: \texttt{del tree[22]}
Making del tree[key] work

```python
# support del tree[key]
def __delitem__(self, key):
    if key not in self:
        raise KeyError
    else:
        self.delete(key)

# helper function for removing nodes
def _replace_child(self, child, new_child):
    if self.left == child:
        self.left = new_child
    elif self.right == child:
        self.right = new_child
```
```python
# remove key from tree
def delete(self, key):
    parent, node = None, self
    while node is not None:
        if key == node.key: break
        parent, node = node, node.left if key < node.key else node.right
    else: raise KeyError

    # node has the key we wish to remove
    if node.left is None:
        if node.right is None:
            assert parent is not None  # bug: can't remove last node!
            parent._replace_child(node, None)
        else:
            # node only has right child, so copy right child contents
            node.key, node.val, node.left, node.right = \
            node.right.key, node.right.val, node.right.left, node.right.right
    else:
        if node.right is None:
            # node only has left child, so copy left child contents
            node.key, node.val, node.left, node.right = \
            node.left.key, node.left.val, node.left.left, node.left.right
        else:
            # node has both children, so replace key,val with that of in-order successor.
            parent, successor = node, node.right
            while successor.left:
                parent, successor = successor, successor.left
            # copy successor key,val to node
            node.key, node.val = successor.key, successor.val
            # now remove successor node
            parent._replace_child(successor, successor.right)
```