03-classes

January 22, 2017

In [2]:
```
# Ignore the code in this cell!!

import svgwrite
import collections

nobinding = "nobinding"

def binding(var):
    try:
        return eval(var)
    except NameError:
        return nobinding

class listis:
    def __init__(self):
        self.lis = []
    def get(self, key):
        for k,v in self.lis:
            if key is k:
                return v
    def put(self, key, val):
        new = True
        for pair in self.lis:
            if pair[0] is key:
                pair[1].append(val)
                new = False
        if new:
            self.lis.append([key, [val]])
    def keys(self):
        return [k for k,v in self.lis]

class memgraph:
    def __init__(self, vars):
        self.vars = sorted(vars)

    def _repr_svg_(self):
        d = svgwrite.Drawing(size=(800,200))
```

```
        left = 100
        right = 260
        dy = 30
        vv = listis()
        ais = listis()

        for var in self.vars:
            val = binding(var)
            if val != nobinding:
                vv.put(val,var)
                ais.put(val, val)

        vals = ais.keys()
        vary = dict()

        y = dy
        d.add(d.text("Variables", insert=(left, y), text_anchor="end", fill
        y += dy

        for var in self.vars:
            d.add(d.text(var, insert=(left, y), text_anchor="end", fill='bl
            vary[var] = y
            y += dy

        y = dy
        d.add(d.text("Objects(in the Heap)", insert=(right, y), fill='blue'
        y += dy

        for val in vals:
            d.add(d.text(str(val), insert=(right, y), fill='black'))

            for var in vv.get(val):
                ly = vary[var]
                d.add(d.line((left, ly ), (right, y),  stroke=svgwrite.rgb
            y += dy

        return d.tostring()

    def svg(self):
        return self._repr_svg_()
```

1 Class

- classes define "templates or blueprints" for building objects
- once a class is defined, any number of objects can be "constructed", or "instantiated"
- everything in Python is an 'object'
  - not true in Java/C++

- all python objects 'live' in the 'heap'
- each object has a fixed 'type', which can be accessed via the 'type' function
- objects have attributes, which are "named objects"
- a 'method' is an attribute holding a function object, which can access and modify the object attributes
- class methods are invoked by functions, operators, and the "." syntax. examples below in 'List'

2 Numbers

- int - arbitrary precision
- float - 64 bits
- complex

```
In [3]: # numbers evaluate to themselves

        1234 # anything after a '#" is a comment and ignored by Python
```

```
Out[3]: 1234
```

```
In [4]: # Python has the usual arithmetic operators

        3*4 - 2**3
```

```
Out[4]: 4
```

```
In [5]: # a float "contaminates" an expression and
        # makes it a float

        3*4 - 2**3.2
```

```
Out[5]: 2.810413160023719
```

```
In [6]: # arbitrary precision integers
        # integer size limited only by available memory

        2**250
```

```
Out[6]: 1809251394333065553493296640760748560207343510400633813116524750123642650624
```

```
In [7]: # 'type' returns the type or class name of an object

        type(2**100)
```

```
Out[7]: int
```

2.0.1 Division operators

- slightly different from most languages
- with integers

```
In [8]: # in most languages this would int 2, but in Python, it is a float

        7/2

Out[8]: 3.5

In [9]: # // is integer divide

        7//2

Out[9]: 3

In [10]: # mod or remainder

         7%2

Out[10]: 1
```

2.0.2 Division operators

- with floats

```
In [11]: 7.0 / 2.0

Out[11]: 3.5

In [12]: 7.0 // 2.0

Out[12]: 3.0

In [13]: 7.0 % 2.0

Out[13]: 1.0

In [14]: # XeY is X*10^Y

         3e3

Out[14]: 3000.0

In [15]: 2.3 * 3e3

Out[15]: 6899.999999999999
```

2.0.3 Complex numbers

```
In [16]: # a complex number times its conjagate is real
         # j is the square root of -1
         # type name is 'complex'

         [(3+4j)*(3-4j), type(3+4j)]

Out[16]: [(25+0j), complex]

In [17]: # int function tries to convert arg to an int

         int('2345')

Out[17]: 2345

In [18]: int(3.45)

Out[18]: 3

In [19]: # likewise for float

         float('3.45')

Out[19]: 3.45

In [20]: float(3)

Out[20]: 3.0
```

3 Object references and variables

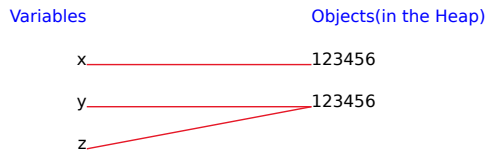
- variables hold 'references' to objects.
- variables do not have or enforce any notion of type
- a given object can have any number of references to it
- there are TWO notions of equality in Python
  - the 'is' operator is true if the two references are to the same object
  - the '==' operator is true if
    - the two references are to the same object, or two different objects "print the same way"(vague!! we will refine later)

```
In [21]: x = 123456
         y = 123456
         z = y

         # graph memory

         memgraph(['x', 'y','z'])
```

Out[21]:



In [22]:
```
# are x & y references to the same object?

x is y
```

Out[22]: False

In [23]:
```
# are y & z references to the same object?

y is z
```

Out[23]: True

In [24]:
```
# y is z => y == z

y == z
```

Out[24]: True

In [25]:
```
# are x & y 'equivalent' in some sense?
# yes - x & y are different objects, but they represent the same integer

x == y
```

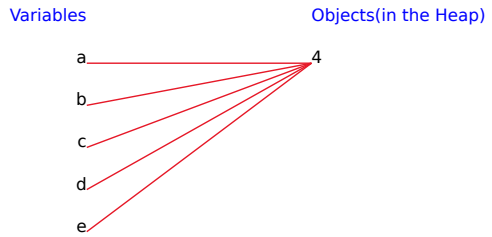
Out[25]: True

In [26]:
```
# if we try a small int, like 4, instead of 123456,
# we get a different result!

# small ints are singletons(interned) for efficiency reasons.
# so, no matter how you compute a '4', you'll get the same '4'
# object

a = 4
b = 4
c = 6 - 2
d = 2*2
e = 2**2
memgraph(['a','b','c','d','e'])
```

Out[26]:



4 Automatic memory Management

- when an object has no references to it, it becomes eligible for 'garbage collection'. the storage it uses is recycled
  - Python uses reference counting
- the user does not have to manage allocating and freeing memory, like Java, unlike C++

5 None

- Like 'null' in other languages
- Means failure or absence of a value
- is a singleton(there is only one object of class None)
- does not print at top level

```
In [27]: # no output

         None
```

```
In [28]: # explicit print will show it

         print(None)
```

```
None
```

```
In [29]: x = None
         y = None

         memgraph(['x', 'y'])
```

Out[29]:

6 Boolean

- Objects: False, True(both singletons)
- Operators: 'not', 'and', 'or'
- <.<=, etc
- unlike many languages, &, &&, |, ||, ~, are not boolean operators

```
In [30]: not(True and (True or False))

Out[30]: False

In [31]: 1234<=1234

Out[31]: True

In [32]: 123<345

Out[32]: True
```

7 Immutable vs Mutable Objects

- Immutable objects, once created, can never be modified
- Mutable objects can be modified at any time

8 Functions

- functions are "first class" objects in Python - they can be assigned as variables, passed as args
- functions are (mostly) immutable objects
- by default, functions return 'None' - you must use the 'return' statement to return a value
- note the ':' at the end of the first line, and the indenting of the function body. this is how you define a 'statement block' in python
- Java/C++ uses '{...}' for statement blocks
- much more about functions later

```
In [33]: # returns 'None'

         def add2(x):
             x + 2

         print(add2(4))
```

```
None
```

```
In [34]: # must use return

         def add2(x):
             return x + 2

         add2(3)
```

```
Out[34]: 5
```

```
In [35]: # can return multiple values by returning a list(or other data structures)

         def addsub2(x):
             return [x+2, x-2]

         addsub2(10)
```

```
Out[35]: [12, 8]
```

9 Collection Types

- hold multiple objects in various configurations
- several kinds are built into the language
- can write "collection literals"
- very easy to use

10 list

- the heart of Python
- much of the "art" of Python involves getting good at manipulating lists
- a list holds a ordered sequence of objects
- duplicates are allowed
- list objects do not have to be the same type
- lists are zero origin - index of first element is 0
- lists are mutable
- some methods, like 'index' and 'count', have no 'side effects' - they don't modify the list
- others, like reverse, modify the list
- methods that modify the list typically return 'None'
- type name is 'list'

```
In [36]: # can make a list by just typing it in

         [1,2,3]

Out[36]: [1, 2, 3]

In [37]: type([2,3,4])

Out[37]: list
```

11 range

- the 'range' form is often used to specify a list of numbers
- often used for iteration purposes
- range evaluates to itself
- range is our first example of "lazy evaluation"
  - major theme in Python 3.X

```
In [38]: range(0, 10)

Out[38]: range(0, 10)

In [39]: # to see the corresponding list, use the list function
         # note range arguments are inclusive/exclusive - there's no 10 in the list

         list(range(0, 10))

Out[39]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [40]: # same as above, assume 0 start

         list(range(10))

Out[40]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [41]: # 3rd arg is increment

         list(range(0, 10, 2))

Out[41]: [0, 2, 4, 6, 8]

In [42]: # can go backwards too - note no 0 in list

         list(range(12, 0, -3))

Out[42]: [12, 9, 6, 3]

In [43]: # 'len' forces evaluation,
         # and returns the "length" of a collection object

         len(range(12,0, -3))
```

```
Out[43]: 4

In [44]: # order matters for lists

         [1,2,3] == [2,1,3]

Out[44]: False

In [45]: [2,1,3] == [2,1,3]

Out[45]: True

In [46]: # duplicates are ok in a list

         [1,1,2,3]

Out[46]: [1, 1, 2, 3]

In [47]: # in languages like Java/C++ would have to select a
         # 'collection' type, instantiate it, and somehow
         # 'stuff' the values in.

         # in python, can just directly "write" a list
         # the assigment statement does not print the right hand side value

         x = [0, 111.111, "zap", True, None]
         y = x
         x

Out[47]: [0, 111.111, 'zap', True, None]

In [48]: # variable by itself prints its value

         x

Out[48]: [0, 111.111, 'zap', True, None]

In [49]: # len returns the length of a list

         len(x)

Out[49]: 5

In [50]: # 'count' method returns a value, does not modify the list
         # count the number of 'True's
         # here the 'dot syntax' is used to invoke the list 'count method'

         x.count(2343)

Out[50]: 0
```

```
In [51]: # reverse returns None - a hint that it modifies the list
         # the 'reverse method' on the list class is invoked

         x.reverse()

In [52]: x

Out[52]: [None, True, 'zap', 111.111, 0]

In [53]: # what happened to y?
         # we didn't explicitly do anything to y, but
         # since y references the same object as x,
         # it 'sees' the reverse that x.reverse() did

         y

Out[53]: [None, True, 'zap', 111.111, 0]

In [54]: # common mistake
         # reverse does NOT return the reversed list
         # if you do this, you just lost your list

         z = [1,2,3,4,5,6]
         z = z.reverse()
         print(z)

None


In [55]: # Another mistake
         # leaving off the '()' just
         # returns the function object
         # the function does NOT run

         z = [1,2,3,4,5,6]
         z.reverse

Out[55]: <function list.reverse>

In [56]: # so no change to z

         z

Out[56]: [1, 2, 3, 4, 5, 6]

In [57]: x

Out[57]: [None, True, 'zap', 111.111, 0]
```

```
In [58]: # Python has very convenient techniques for accessing
         # and modifying list elements
         # can index into the list like an array,
         # and retrieve one element

         x[2]

Out[58]: 'zap'

In [59]: # negative index starts from the last list element

         x[-1]

Out[59]: 0

In [60]: # can take a subsequences (slice) of the list
         # like range, inclusive/exclusive
         # slices always COPY the original list

         x[0:2]

Out[60]: [None, True]

In [61]: # missing second index means continue slice to the end of the list

         x[3:]

Out[61]: [111.111, 0]

In [62]: # missing first index means start slice at begining  of the list

         x[:2]

Out[62]: [None, True]

In [63]: # can add a index increment to a slice

         x[0:8:2]

Out[63]: [None, 'zap', 0]

In [64]: # slices can be named for readability

         triple = slice(0,8,2)
         x[triple]

Out[64]: [None, 'zap', 0]
```

```
In [65]: # index missing on both sides of ":" - slice
         # is the whole list.
         # common python shorthand for copying
         # an entire list

         x2 = x[:]

         # reverse modifies x2, but x will not be changed, because
         # x and x2 are referencing different objects
         # reverse() returns 'None'

         print(x2)
         print(x2.reverse())
         print(x2)
         print(x)

[None, True, 'zap', 111.111, 0]
None
[0, 111.111, 'zap', True, None]
[None, True, 'zap', 111.111, 0]


In [66]: # can set list elements

         x[0] = -1
         x

Out[66]: [-1, True, 'zap', 111.111, 0]

In [67]: # can set slices

         x[3:5] = [2**8, False]
         x

Out[67]: [-1, True, 'zap', 256, False]

In [68]: #  'in' operator - is an element in the list somewhere?
         # uses == to test

         ['zap' in x, 55 in x]

Out[68]: [True, False]

In [69]: # where is the element?
         # 'index' is a 'method' on the list class

         x.index('zap')

Out[69]: 2
```

```
In [70]: # index throws an error if it doesn't find anything
         # we will learn more about errors later

         x.index("not in there")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-70-83c8fd21a8b8> in <module>()
  2 # we will learn more about errors later
  3
----> 4 x.index("not in there")


ValueError: 'not in there' is not in list
```

```
In [ ]: # + concatenates lists
        # note: what '+' actually does depends on the type of its arguments

        x = list(range(5))
        x + x
```

```
In [ ]: x
```

```
In [ ]: # add one element at the end

        x.append([22,33])
        x
```

```
In [ ]: # add N elements at the end

        x.extend([22,33])
        x
```

```
In [ ]: # add one element anywhere

        x.insert(2, 5)
        x
```

```
In [ ]: # pop method removes and returns a
        # list element, by default the last element

        print(x.pop())
        print(x)
```

```
In [ ]: # but can specify which element to pop
```

```
        print(x.pop(2))
        print(x)

In [ ]: # remove first 4 found

        x.remove(4)
        print(x)

In [ ]: # sort modifies the list

        x = [34,3,5,22]
        x.sort()
        x

In [ ]: # can preserve original list by using 'sorted'
        # sorted makes a copy of the input list

        x = [34,3,5,22]
        y = sorted(x)
        [x, y]

In [ ]: # dir shows the methods defined on a class
        # __XYZ__ are "special" methods - ignore them for now

        dir(list)
```

12 Iterating over Lists

- Many ways to iterate, we'll look at the two most important here, 'for' and 'list comprehensions'
- Python does NOT have C++/Java style loops, like:

for(int j = 0; j<5; j++) { }

13 for loop

- Python version of C++/Java loop above
- Python loops are simpler
- note trailing ':', and indented print statements - defines a statement block
- Python uses idents and ':' to define blocks, unlike C/Java, which uses '{}'

```
In [ ]: for j in range(10,15):
            print(j)
            print(j+10)
        print('loop finished')

In [ ]: # to sum up a list of numbers
        # use zn 'acculumation variable'
```

```
sum = 0

for j in range(5):
    sum += j

sum
```

```
In [ ]: # add 10 to every element of a list
        # use list acculumation variable

        a10 = []

        for j in range(5):
            a10.append(10+j)

        a10
```

14 list comprehension

- above technique is not conidered 'pythonic'
- syntax is a little odd at first glance
- no accum var needed
- can optionally do filtering

```
In [ ]: # add 10 again

        [j+10 for j in range(5)]
```

```
In [ ]: # filter - add 10 only to even ints
        # '%' is mod operator

        [j+10 for j in range(5) if j % 2 == 0]
```

15 Tuples

- like lists, but immutable - can't be modified after creation
  - however, objects that the tuple refers to can still be modified
- useful for functional programming
- 'tuple' is the type name

```
In [ ]: # len returns length of top level elements

        t = (1,[5,6],4)
        [t , len(t), type(t)]
```

```
In [ ]: len(t)
```

```
In [ ]: # can retrieve

        t[0]

In [ ]: # but can't modify

        t[0] = 3

In [ ]: t

In [ ]: # but - objects the tuple refers to are NOT made immutable

        t[1][0] = 45
        t

In [ ]: # tuples loop like lists

        for x in (1,2,3):
            print(x)

In [74]: # a one element tuple has odd syntax...

         t =(1,)
         t

Out[74]: (1,)

In [75]: len(t)

Out[75]: 1

In [73]: # ...to distingish it from

         (1)

Out[73]: 1
```

16 Iterables

- 'iterables' are objects you can iterate over
- lists and tuples are iterables

17 Strings

- immutable - once created, cannot be modified
- in Python version 3.X, strings are unicode
- many useful methods
- the 're' module provides regular expression pattern matching

- three types of string literals ‘foo’, “foo”, and ‘”foo”’
- triple quotes can include multiple lines
- unlike other languages, there is no ‘character’ type
- a Python ‘character’ is just a length 1 string
- ‘str’ is the type name

In [ ]:
```
# len returns number of characters

['foobar', 'foo"bar', type('foobar'), len('foobar')]
```

In [ ]:
```
# various ways to embed quotes

['foo"bar', "foo'bar", 'foo\'bar']
```

In [ ]:
```
# use triple quotes to define multi-line strings

'''
foo'
bar"
'''
```

In [ ]:
```
# Strings are iterables

for s in 'FooBar':
    print(s)
```

In [ ]:
```
# string methods that return a string always return a NEW string.
# the original string is NEVER modified

s = 'FooBar'
ls = [s, s.lower(), s.upper(), s.replace('o','X'), s.swapcase()]
```

In [ ]:
```
# first element of list is the original 'FooBar' - has not
# been modified by any of the methods run above
# rest of list contains 4 NEW string objects, derived from the
# original 'FooBar'

ls
```

In [ ]:
```
# join is a very handy method

[','.join(ls), '|'.join(ls), '---'.join(ls)]
```

In [ ]:
```
# the inverse, split, creates a list of tokens

s = "foo,bar,34,zap"
s.split(",")
```

In [ ]:
```
# strip can remove chars at the begining(left) and/or end(right) of a strin
# Note middle 'X' is not removed
```

```
        # Most commonly used to remove new lines from a string

        s = 'XXfooXbarXXX'
        [s.strip('X'), s.lstrip('X'), s.rstrip('X')]
```

```
In [ ]: # '+' concatenates strings as well as lists
        # the operation '+' performs depends on the type of the arguments

        s + s
```

```
In [ ]: # can repeat strings

        [2*"abc", "xyz"*4]
```

```
In [ ]: # 'in' looks for substrings
        # case sensitive compares

        s = 'zappa'
        ['pa' in s, 'Za' in s, s.count('p'), s.count('ap')]
```

```
In [ ]: # search for a substring with 'find' or 'index'

        [s.find('pa'), s.index('pa')]
```

```
In [ ]: # on a miss, 'find' returns -1

        s.find('32')
```

```
In [ ]: # but index throws an error

        s.index('32')
```

```
In [ ]: # 'ord' and 'chr' do character-number conversions

        [ord('A'), chr(65)]
```

```
In [ ]: # make the lower case chars, a-z
        # somewhat terse one liner -
        # in Python you can do alot with a little code,
        # but can be hard to read

        lc= ''.join([chr(c) for c in range(ord('a'), ord('z')+1)])
        lc
```

```
In [ ]: # let's break it into separate steps:
        # get the ascii codes for 'a' and 'z'

        a = ord('a')
        z = ord('z')
        [a,z]
```

```
In [ ]: # now we have all the codes for 'a' to 'z'
        # note the z+1 - need the +1 to get the z code

        codes = [c for c in range(a,z+1)]
        print(codes)

In [ ]: # now we have a list of the lower case characters

        chars = [chr(c) for c in codes]
        print(chars)

In [ ]: # last step - using the 'join' method on string,
        # merge the chars into one string

        ''.join(chars)

In [ ]: # now that we have suffered, there is an easier way
        # string package has useful constants

        import string
        string.ascii_lowercase

In [ ]: # can slice strings too

        [len(lc), lc[10:20], lc[10:20:2], lc[10:11]]

In [ ]: # unlike a list, a string is immutable - you can't change anything

        s = 'foobar'
        s[0] = 't'

In [ ]: # unlike list objects, string objects don't have a reverse method
        # but you can reverse with a slice
        # works with lists as well

        s = '1234'
        z = [1,2,3,4]
        [s[::-1], z[::-1]]

In [ ]: # startswith, endwith string methods are sometimes
        # convenient alternatives to regular expressions

        a = "foo.txt"

        [a.startswith('foo'), a.endswith('txt'), a.endswith('txt2')]

In [ ]: # 'str' converts objects to strings

        [str(234), str(3.34), str([1,2,3])]
```

```
In [ ]: # 'list' converts a string into a list of
        # characters(length one strings)

        list('foobar')
```

18 'printf' style string formatting - old way

```
- still works, but deprecated

In [ ]: 'int %d float %f string %s' % (3, 5.5, 'printf')
```

19 'printf' style string formatting - new way

- preferred method
- looks at the type of the arg, so don't have to specify type in control string
- details

```
In [ ]: 'int {} float {} string {}'.format(3, 5.5, 'printf')
```

20 print function

- will print any number of args

```
In [76]: print(4,'asdf',[3,4])

4 asdf [3, 4]


In [ ]: # lots of methods on strings

        dir(str)
```