Advanced data types

Reuven M. Lerner, PhD
reuven@lerner.co.il

Don't use "is"!

```
x = 'abc' * 5
y = 'abc' * 5
x is y              # True
x = 'abc' * 5000
y = 'abc' * 5000
x is y              # False!
```

2

Don't use "is"

```
x = 256
y = 256
x is y   # True
x = 257
y = 257
x is y   # False!
```

Huh?

- It turns out that Python allocates objects for all integers from -5 to 256 when it starts
- Which means that if you use just those, you stay within the "safe" range, in which objects stay the same
- But if you allocate numbers outside of that zone, they're different. Probably.

More fun

```
x = 257; y = 257

x is y   # True!
```

5

How can this be?

```
mycode  = 'x = 257; y=257'

code_obj = compile(mycode,

                   filename='', mode='exec')

code_obj.consts
```

6

Decimals

- We know that we shouldn’t use floats for exact measurements, and that Python doesn’t supply a “double” type
- The “decimal” module provides a “Decimal” class that can help us to get around this

Basic use

```
>>> from decimal import Decimal
>>> d1 = Decimal('0.7')
>>> d2 = Decimal('0.1')
>>> d1 + d2
Decimal('0.8')
>>> float(d1+d2)
0.8
```

8

Use strings, not floats!

```
>>> d1 = Decimal(0.7)

>>> d2 = Decimal(0.1)

>>> d1 + d2

Decimal('0.7999999999999999611421941381')

>>> float(d1+d2)

0.7999999999999999
```

Context

- The decimal “context” tells the system how to behave under certain circumstances
- The decimal.Context object allows us to set such things
- We can use decimal.getcontext() to get the current Context object
- We can use decimal.setcontext() to assign a new Context object to our current situation

Precision

```
>>> d1 * 5
Decimal('3.4999999999999997779553950750')
>>> decimal.setcontext(decimal.Context(prec=5))
>>> d1 * 5
Decimal('3.5000')
```

Rounding

- You can also set the rounding algorithm used

```
>>> decimal.setcontext(decimal.Context(rounding=decimal.ROUND_FLOOR))

>>> d1 * 5

Decimal(‘3.499999999999999777955395074')

>>> decimal.setcontext(decimal.Context(rounding=decimal.ROUND_UP))

>>> d1 * 5

Decimal('3.499999999999999777955395075')
```

isclose

- New in Python 3.5!

```
>>> from math import isclose

>>> isclose(5.0, 5.1)

# Relative tolerance vs. absolute tolerance

>>> isclose(5.0, 5.1, abs_tol=0.11)

>>> isclose(5.0, 5.1, rel_tol=0.11)
```

math module

- This module has existed for a long time
- It includes many math functions defined by the C standard
- If you're looking for basic math functionality, this is a good place to start
- It also defines math.pi and math.e as floats

The "string" module

- The functions aren't really that necessary
- But the data can be quite useful!

Data in "string" module

```
ascii_letters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
digits = '0123456789'
hexdigits = '0123456789abcdefABCDEF'
letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
lowercase = 'abcdefghijklmnopqrstuvwxyz'
octdigits = '01234567'
printable = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU...
punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
whitespace = '\t\n\x0b\x0c\r '
```

16

str.format

- The simplest way to use str.format is

```
"I have {0} and {1}".format('1st', '2nd')
```

- You can also say (in 2.7 and 3.x)

```
"I have {} and {}".format('1st', '2nd')
```

Unpacking

```
mylist = ['1st', '2nd']

"I have {} and {}".format(*mylist)
```

Named parameters

```
"I have {a} and {b}".format(a='1st',
                             b='2nd')
```

Unpacking kwargs

```
d = {'a':'1st', 'b':'2nd'}

"I have {a} and {b}".format(**d)
```

20

str.format mini-language

```
"a {} b".format('qqqq')

"a {:20} b".format('qqqq')  # flush left

"a {:<20} b".format('qqqq') # flush left

"a {:>20} b".format('qqqq') # flush right

"a {:^20} b".format('qqqq') # centered
```

21

str.format examples

- A guide to str.format is now available at
- http://pyformat.info/
- It contains lots of great examples of what you can do with str.format

Integers

```
>>> 'abc {:10} def'.format(100)
'abc        100 def'


>>> 'abc {:010} def'.format(100)
'abc 0000000100 def'
```


23

Floats

```
>>> 'abc {} def'.format(12345.67890)            # default, show all
'abc 12345.6789 def'

>>> 'abc {:.4} def'.format(12345.67890)         # max 4 digits
'abc 1.235e+04 def'

>>> 'abc {:12.4} def'.format(12345.67890)       # pad to 12 spaces
'abc    1.235e+04 def'

>>> 'abc {:012.4} def'.format(12345.67890)      # pad with zeroes
'abc 0001.235e+04 def'

>>> 'abc {:<012.4} def'.format(12345.67890)     # left-aligned, pad with zeroes
'abc 1.235e+04000 def'
```

24

StringIO and cStringIO

- StringIO is a class that pretends to be a file, but is really a string
- Great for when you want to simulate files, without actually using them
- cStringIO is a module that provides its own, C-based version of StringIO

Using StringIO

```
from StringIO import StringIO

output = StringIO()

output.write('This goes into the buffer.\n')

output.write('And so does this.')

print output.getvalue()

output.seek(0)

print output.read()
```

26

More with StringIO

- Just about anything you can do with a file, you can do with a StringIO

```
output.seek(0)

output.readlines()

['This goes into the buffer.\n', 'And so
does this.']
```

Reading Unicode files

- We can define Unicode strings with a leading "u"
- But what if we want to read Unicode data? The normal file operations return strings
- We can use the "codecs" module to open files for us, and thus return Unicode (or any encoding we want)

The wrong way

```
f= open('unicode.txt')
for line in f:
    for char in line:
        print char
    print
```

The right way

```
import codecs

f= codecs.open('unicode.txt',

                encoding='utf-8')

for line in f:

    for char in line:

        print char

    print
```

30

This won't work!

```
for line in f:
    for index, char in enumerate(line):
        print "{}: {}".format(index, char)
    print
```

Ah, much better…

```
for line in f:
    for index, char in enumerate(line):
        print u"{}: {}".format(index, char)
    print
```

32

Lists

- We know lists can be of any size, and contain any type(s)
- What you might not know:
    - Lists are fixed-length arrays of pointers
- So, what happens when we add elements?

List handling

- When the array grows or shrinks, it calls realloc() to reallocate memory
- If necessary, Python copies all of the objects (pointers) to somewhere new

realloc() every time?

- Well, sort of.
- Python assumes that your memory allocation system is bad, and compensates.
- So there's always a bit of extra room
- (If the size is known, such as from map() or range(), then the allocated array is precise.)

For example

```
>>> s = []
>>> for c in string.letters:
…         s.append(c)
[]                     # 0 items takes 0 space
[A . . .]              # 1 item takes 4 spaces
[A B . .]              # Second append() is free!
[A B C .]              # So is the third.
[A B C D]              # And the fourth.
[A B C D E . . .]      # Fifth item costs a realloc()
[A B C D E F . .]      # Sixth is free!
(Size is 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, ...)
```

Removing objects

- realloc() is called when we are using less than 50 percent of the space
- Modifying the end is very cheap
- Modifying the start or middle is O(n)
  - deque is faster on the ends, but slower on accesses and in the middle

Hashable vs. immutable

- All immutable types are hashable
- Many other types (e.g., user-defined classes) are hashable, too!
- Only hashable values can be used as dict keys or put in sets

38

Python's hash function

- Numbers hash to themselves (except -1)
- Objects hash to their ids (unless overridden)
- Hashing is deterministic, and thus subject to attack, unless you use the -R option to Python 2

setdefault

- setdefault lets you create a key-value pair in a dictionary, if the key doesn't exist

```
d = {'a':1, 'b':2}
d.setdefault('c', 3)
d  # Prints {'a': 1, 'b': 2, 'c': 3}

d.setdefault('c', 10) # No change!
d  # Prints {'a': 1, 'b': 2, 'c': 3}
```

Clear a dict!

- You probably never want to do this
- But just in case, you can reset a dictionary to be empty with the clear() method:

```
d = {'a': 1, 'b': 2, 'c': 3}

d.clear()

d                    # Prints { }
```

41

Named tuples

- More efficient than dictionaries, but easier to work with (semantically) than tuples
- In reality, it's a subclass of tuple with names as aliases for its numeric indexes
- Very useful, and very efficient!

Named tuples

```
Point = namedtuple('Point', ['x', 'y'])
Point.__doc__                # docstring for the new class
    'Point(x, y)'

p = Point(11, y=22)          # instantiate with positional
                             # args or keywords
p[0] + p[1]                  # indexable like a plain tuple
    33

x, y = p                     # unpack like a regular tuple
x, y
    (11, 22)

p.x + p.y                    # fields also accessible by name
    33

d = vars(p1)              # convert to a dictionary
d['x']
    11
```

43

More with named tuples

```
Point(**d)                # convert from a dictionary
Point(x=11, y=22)


p._replace(x=100)         # _replace() is like
                          # str.replace()
                          # but targets named fields
Point(x=100, y=22)
```

44

deque (“deck”)

- Double-ended queue
- Good for stacks and/or queues!

```
from collections import deque
d = deque('ghi')
for elem in d:
    print elem.upper()
```

Counter

- Class that provides some convenience functions for counting objects
- Sort of like using a dictionary (or a defaultdict) for counting items
- Elements don’t have to be hashable, though

46

Example

```
from collections import Counter

logins = Counter()

logins['reuven@lerner.co.il'] += 5

logins['foo@bar.com'] += 1

logins['reuven@lerner.co.il'] += 1

logins['reuven@lerner.co.il']

    6

logins

    Counter({'reuven@lerner.co.il': 6, 'foo@bar.com': 1})
```

47

Most common items

```
>>> c = Counter('aaaabbbcc')
>>> dict(c)
{'a': 4, 'b': 3, 'c': 2}

>>> c.most_common(3)
[('a', 4), ('b', 3), ('c', 2)]

>>> c.most_common(2)
[('a', 4), ('b', 3)]

>>> c.most_common(1)
[('a', 4)]
```

defaultdict

- A dictionary that will return a default value when a key doesn't exist
- Think of it as a dict on which retrievals are automatically wrapped with a get call
- The first retrieval of a key sets its value
- The default value is passed as a function, which is evaluated each time

Example

```
from collections import defaultdict
import time
d = defaultdict(time.time)

d['a']
    1368313801.971879
d['a']
    1368313801.971879
d['b']
    1368313804.420007
```

50

OrderedDict

- Just like a regular dictionary, except that it remembers the order in which keys were entered

OrderedDict usage

```
o = OrderedDict()
o['z'] = 26
o['a'] = 1
o
OrderedDict([('z', 26), ('a', 1)])
o['b'] = 2
o
  OrderedDict([('z', 26), ('a', 1), ('b', 2)])
o.pop('z')
  26
o['z'] = 26
o
  OrderedDict([('a', 1), ('b', 2), ('z', 26)])
```

52

@, for matrix multiplication

- New in Python 3.5!
- Python doesn't use it, but makes it available for your own use
- __matmul__ is available, as are __rmatmul__ and __imatmul__, in your classes
- (No ambiguity with decorators, because of parsing rules)