Built-In Functions

Special thanks to Scott Shawcroft, Ryan Tucker, and Paul Beck for their work on these slides.
Except where otherwise noted, this work is licensed under:
http://creativecommons.org/licenses/by-nc-sa/3.0
Exceptions

raise \texttt{type}(message)
raise Exception(message)

<table>
<thead>
<tr>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssertionError</td>
</tr>
<tr>
<td>TypeError</td>
</tr>
<tr>
<td>NameError</td>
</tr>
<tr>
<td>ValueError</td>
</tr>
<tr>
<td>IndexError</td>
</tr>
<tr>
<td>SyntaxError</td>
</tr>
<tr>
<td>ArithmeticError</td>
</tr>
</tbody>
</table>

http://docs.python.org/library/exceptions.html#bltin-exceptions
We already know about the `__str__()` method that allows a class to convert itself into a string.

class Rectangle:
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
    def __str__(self):
        return "(x=" + str(self.x) + ",y=" + 
                        str(self.y) + ",w=" + str(self.width) + 
                        ",h=" + str(self.height) + ")"
Underscored methods

- There are many other underscored methods that allow the built-in function of python to work
- Most of the time the underscored name matches the built-in function name

<table>
<thead>
<tr>
<th>Built-In</th>
<th>Class Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>str()</td>
<td><strong>str</strong>()</td>
</tr>
<tr>
<td>len()</td>
<td><strong>len</strong>()</td>
</tr>
<tr>
<td>abs()</td>
<td><strong>abs</strong>()</td>
</tr>
</tbody>
</table>
First Class Citizens

• For built-in types like `ints` and `strings` we can use operators like `+` and `*`.

• Our classes so far were forced to take back routes and use methods like `add()` or `remove()`.

• Python is super cool, in that it allows us to define the usual operators for our class.

• This brings our classes up to first class citizen status just like the built in ones.
Underscored methods

- There are underscore methods that you can implement in order to define logical operations and arithmetic operations.

### Binary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Class Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td><strong>neg</strong>(self, other)</td>
</tr>
<tr>
<td>+</td>
<td><strong>pos</strong>(self, other)</td>
</tr>
<tr>
<td>*</td>
<td><strong>mul</strong>(self, other)</td>
</tr>
<tr>
<td>/</td>
<td><strong>truediv</strong>(self, other)</td>
</tr>
</tbody>
</table>

### Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Class Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td><strong>eq</strong>(self, other)</td>
</tr>
<tr>
<td>!=</td>
<td><strong>ne</strong>(self, other)</td>
</tr>
<tr>
<td>&lt;</td>
<td><strong>lt</strong>(self, other)</td>
</tr>
<tr>
<td>&gt;</td>
<td><strong>gt</strong>(self, other)</td>
</tr>
<tr>
<td>&lt;=</td>
<td><strong>le</strong>(self, other)</td>
</tr>
<tr>
<td>&gt;=</td>
<td><strong>ge</strong>(self, other)</td>
</tr>
</tbody>
</table>

### Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Class Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td><strong>neg</strong>(self)</td>
</tr>
<tr>
<td>+</td>
<td><strong>pos</strong>(self)</td>
</tr>
</tbody>
</table>
ArrayIntList Operations

Lets write a method that we could add to arrayintlist.py that would allow us to apply the /= operation to the list. The operation would simply divide all elements of the list by the argument of the operator.

Method: `__itruediv__(self, num)`

Example run

```
1   print(int_list)       # [1, 2, 3, 4, 5, 6, 7]
2   int_list /= 2
3   print(int_list)       # [0.0, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5]
```
def __itruediv__(self, num):
    if num == 0 :
        raise ArithmeticError("Can't divide by zero.")
    for i in list(range(len(self))) :
        self.elementData[i] /= num
    return self
Lambda

- Sometimes you need a simple arithmetic function
- It's silly to write a method for it, but redundant not too much
- With lambda we can create quick simple functions
- Facts
  - Lambda functions can only be comprised of a single expression
  - No loops, no calling other methods
  - Lambda functions can take any number of variables

Syntax:

```
lambda param1,...,paramn : expression
```
# Example 1
square_func = lambda x : x**2
square_func(4) #return: 16

# Example 2
close_enough = lambda x, y : abs(x - y) < 3
close_enough(2, 4) #return: True

# Example 3
def get_func(n) :
    return lambda x : x * n + x % n
my_func = get_func(13)
my_func(4) #return: 56
Higher-Order Functions

- A higher-order function is a function that takes another function as a parameter.
- They are “higher-order” because it’s a function of a function.
- Examples
  - Map
  - Reduce
  - Filter
- Lambda works great as a parameter to higher-order functions if you can deal with its limitations.
filter(function, iterable)

- The filter runs through each element of iterable (any iterable object such as a List or another collection)
- It applies function to each element of iterable
- If function returns True for that element then the element is put into a List
- This list is returned from filter in versions of python under 3
- In python 3, filter returns an iterator which must be cast to type list with list()
Filter Example

Example

```
nums = [0, 4, 7, 2, 1, 0, 9, 3, 5, 6, 8, 0, 3]
nums = list(filter(lambda x: x != 0, nums))
print(nums)  # [4, 7, 2, 1, 9, 3, 5, 6, 8, 3]
```
**Filter Problem**

```python
NaN = float("nan")
scores = [[NaN, 12, .5, 78, math.pi],
          [2, 13, .5, .7, math.pi / 2],
          [2, NaN, .5, 78, math.pi],
          [2, 14, .5, 39, 1 - math.pi]]
```

Goal: given a list of lists containing answers to an algebra exam, filter out those that did not submit a response for one of the questions, denoted by NaN
Filter Problem

Solution

```python
NaN = float("nan")
scores = [[NaN, 12, .5, 78, pi],[2, 13, .5, .7, pi / 2],
          [2,NaN, .5, 78, pi],[2, 14, .5, 39, 1 - pi]]

#solution 1 - intuitive
def has_NaN(answers):
    for num in answers:
        if isnan(float(num)):
            return False
    return True

valid = list(filter(has_NaN, scores))
print(valid2)

#Solution 2 - sick python solution
valid = list(filter(lambda x : NaN not in x, scores))
print(valid)
```
map(function, iterable, ...)

• Map applies **function** to each element of **iterable** and creates a list of the results

• You can optionally provide more iterables as parameters to map and it will place tuples in the result list

• Map returns an iterator which can be cast to list
Example

```python
nums = [0, 4, 7, 2, 1, 0, 9, 3, 5, 6, 8, 0, 3]
nums = list(map(lambda x: x % 5, nums))
print(nums)
# [0, 4, 2, 2, 1, 0, 4, 3, 0, 1, 3, 0, 3]
```
Map Problem

Goal: given a list of three dimensional points in the form of tuples, create a new list consisting of the distances of each point from the origin

Loop Method:
- \( \text{distance}(x, y, z) = \sqrt{x^2 + y^2 + z^2} \)
- loop through the list and add results to a new list
Map Problem

Solution

```python
from math import sqrt

points = [(2, 1, 3), (5, 7, -3), (2, 4, 0), (9, 6, 8)]

def distance(point):
    x, y, z = point
    return sqrt(x**2 + y**2 + z**2)

distances = list(map(distance, points))
```
reduce function, iterable[,initializer])

- Reduce will apply function to each element in iterable along with the sum so far and create a cumulative sum of the results
- function must take two parameters
- If initializer is provided, initializer will stand as the first argument in the sum
- Unfortunately in python 3 reduce() requires an import statement
  - from functools import reduce
Reduce Example

Example

```python
nums = [1, 2, 3, 4, 5, 6, 7, 8]
nums = list(reduce(lambda x, y: (x, y), nums))
Print(nums)  #((((((1, 2), 3), 4), 5), 6), 7), 8)
```
Goal: given a list of numbers I want to find the average of those numbers in a few lines using `reduce()`

For Loop Method:
- sum up every element of the list
- divide the sum by the length of the list
nums = [92, 27, 63, 43, 88, 8, 38, 91, 47, 74, 18, 16, 29, 21, 60, 27, 62, 59, 86, 56]

sum = reduce(lambda x, y : x + y, nums) / len(nums)