
# Built-In Functions

Special thanks to Scott Shawcroft, Ryan Tucker, and Paul Beck for their work on these slides.

# Exceptions

```
raise type(message)
raise Exception(message)
```

| Exceptions |
| --- |
| AssertionError |
| TypeError |
| NameError |
| ValueError |
| IndexError |
| SyntaxError |
| ArithmeticError |


http://docs.python.org/library/exceptions.html#bltin-exceptions

# __str__()

- We already know about the __str__() method that allows a class to convert itself into a string

**rectangle.py**

```
1  class Rectangle:
2      def __init__(self, x, y, width, height):
3          self.x = x
4          self.y = y
5          self.width = width
6      def __str__(self):
7          return "(x=" + str(self.x) + ",y=" +
8              str(self.y) + ",w=" + str(self.width) +
9              ",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

| Built-In | Class Method |
|---|---|
| `str()` | `__str__()` |
| `len()` | `__len__()` |
| `abs()` | `__abs__()` |

# 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

| Operator | Class Method |
|---|---|
| - | `__neg__(self,other)` |
| + | `__pos__(self, other)` |
| * | `__mul__(self, other)` |
| / | `__truediv__(self, other)` |

Unary Operators

| Operator | Class Method |
|---|---|
| - | `__neg__(self)` |
| + | `__pos__(self)` |

Comparison Operators

| Operator | Class Method |
|---|---|
| == | `__eq__(self,other)` |
| != | `__ne__(self, other)` |
| < | `__lt__(self, other)` |
| > | `__gt__(self, other)` |
| <= | `__le__(self, other)` |
| >= | `__ge__(self, other)` |
| N/A | `__nonzero__(self)` |


http://docs.python.org/reference/datamodel.html#sequence-types

# 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]
4
```

# Solution

**arrayintlist.py**

```
1  def __itruediv__(self, num):
2      if num == 0 :
3          raise ArithmeticError("Can't divide by zero.")
4      for i in list(range(len(self))) :
5          self.elementData[i] /= num
6      return self
7
```

# Lambda

- Sometimes you need a simply arithmetic function
- Its silly to write a method for it, but redundant not too
- 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
```

# Lambda Syntax

**lambda.py**

```
1  #Example 1
2  square_func = lambda x : x**2
3  square_func(4)                          #return: 16
4
5  #Example 2
6  close_enough = lambda x, y : abs(x - y) < 3
7  close_enough(2, 4)                      #return: True
8
9  #Example 3
0  def get_func(n) :
1      return lambda x : x * n + x % n
2  my_func = get_func(13)
3  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

`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**

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

# Filter Problem

```
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**

```
1  NaN = float("nan")
2  scores = [[NaN, 12, .5, 78, pi],[2, 13, .5, .7, pi / 2],
3            [2,NaN, .5, 78, pi],[2, 14, .5, 39, 1 - pi]]
4  #solution 1 - intuitive
5  def has_NaN(answers) :
6     for num in answers :
7       if isnan(float(num)) :
8           return False
9     return True
0  valid = list(filter(has_NaN, scores))
1  print(valid2)
2  #Solution 2 - sick python solution
3  valid = list(filter(lambda x : NaN not in x, scores))
4  print(valid)
```

# Map

`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

# Map Example

**Example**

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

# 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:

- 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**

```
1  from math import sqrt
2
3  points = [(2, 1, 3), (5, 7, -3), (2, 4, 0), (9, 6, 8)]
4
5  def distance(point) :
6      x, y, z = point
7      return sqrt(x**2 + y**2 + z**2)
8
9  distances = list(map(distance, points))
```

# Reduce

`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**

```
1  nums = [1, 2, 3, 4, 5, 6, 7, 8]
2
3  nums = list(reduce(lambda x, y : (x, y), nums))
4
5  Print(nums)        #(((((((1, 2), 3), 4), 5), 6), 7), 8)
6
7
```

# Reduce Problem

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

# Reduce Problem

**Solution**

```
1  nums = [92, 27, 63, 43, 88, 8, 38, 91, 47, 74, 18, 16,
           29, 21, 60, 27, 62, 59, 86, 56]
2
3  sum = reduce(lambda x, y : x + y, nums) / len(nums)
4
```