

CSC148 Ramp-up

Fall 2013

Orion Buske

(based on notes by Velian Pandeliev, Jonathan Taylor,
Noah Lockwood, and software-carpentry.org)

Overview

In the next 6 hours, we'll cover the background required for CSC148.

This session is for students with programming experience who haven't necessarily taken the prerequisite, CSC108.

Please ask questions!

Outline

- Talking
- Talking
- Talking
- Lunch
- Talking
- Talking
- Talking

More explicit outline

- Variables and types
- Lists, tuples, and for loops
- Conditionals and functions
- Lunch
- Dictionaries
- While loops and modules
- Classes and objects

Meet Python...



Let's speak some Python

- Python is interpreted (no compilation necessary)
- Whitespace matters (4 spaces for indentation)
- No end-of-line character (no semicolons!)
- No extra code needed to start (no "public static ...")
- Python has dynamic typing (all variables are void*)

```
# Comments start with a '#' character.
```

```
# Python has dynamic typing, so:
```

```
x = 5 # assignment statement (no type specified)
```

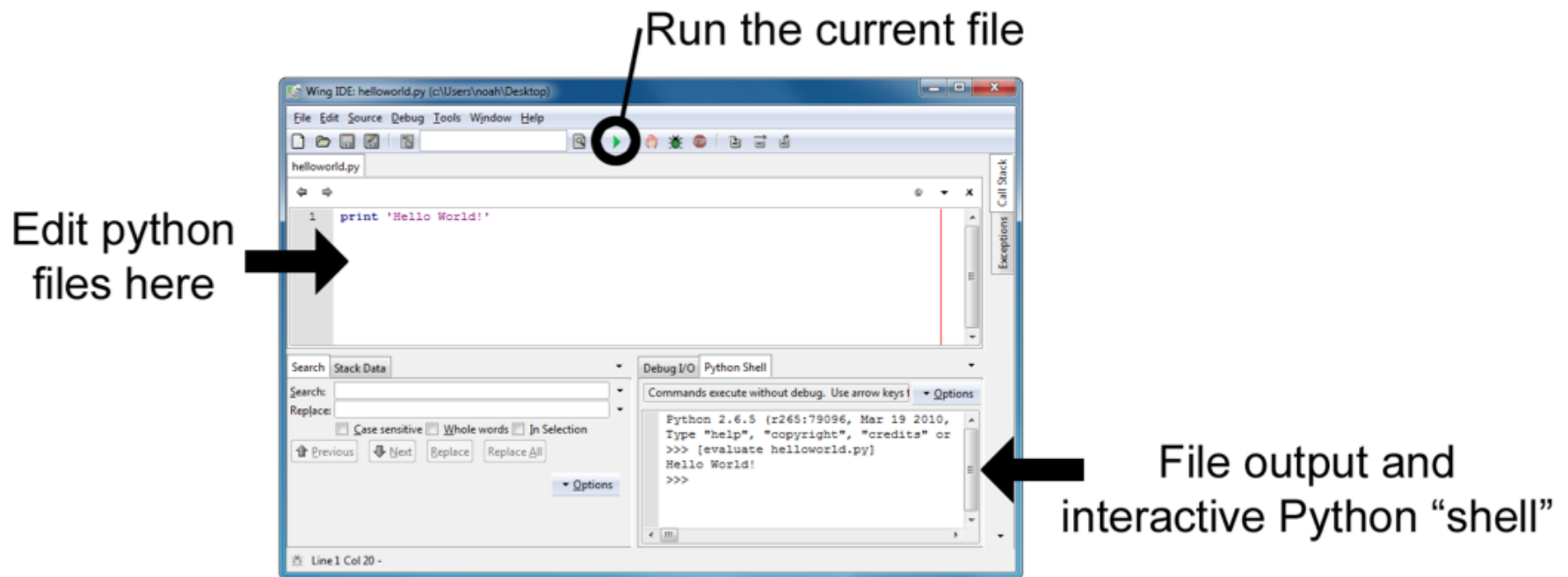
```
x = "jabberwocky" # re-assign x to a string
```

```
print(x) # prints "jabberwocky"
```

Python programs

- Programs are stored in .py files
- From the command line (for teh hax0rz):

```
#user@redwolf:~$ python myfile.py
```
- Using the Wing IDE (Integrated Dev. Environment)



The blueprint of a Python file:

```
from random import randint  
from math import cos
```

| import names from
other modules

The blueprint of a Python file:

```
from random import randint  
from math import cos
```

import names from
other modules

```
def my_function(arg):  
    ...  
    return answer
```

define functions
and classes

```
class MyClass(object):  
    ...
```

The blueprint of a Python file:

```
from random import randint  
from math import cos
```

import names from
other modules

```
def my_function(arg):  
    ...  
    return answer
```

define functions
and classes

```
class MyClass(object):  
    ...
```

```
if __name__ == "__main__":  
    ...
```

your main block
goes down here!

The blueprint of a Python file:

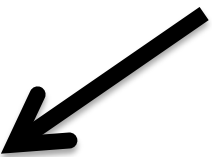
```
from random import randint
from math import cos
```

```
def my_function(arg):
    ...
    return answer
```

```
class MyClass(object):
    ...
```

```
if __name__ == "__main__":
    ...
```

**the
main
block
mantra**



Interactive Python

- Python can also be run interactively, from the bottom-right of Wing, or by typing `python` on the command line. The result is automatically printed.

```
#user@redwolf:~$ python
Python 3.2.3 (v3.2.3:3d0686d90f55, Apr 10 2012, 11:25:50)
Type "help", "copyright", "credits" or "license" for more
information.
>>> 42
42
>>> (2 ** 3 - 4) / 8
0.5
```

Getting help

Official Python documentation:

<http://docs.python.org/py3k/library/>

The `help` function provides usage information:

```
>>> help(print)
```

The `dir` function shows methods, names, attributes, etc. for a given type, module, object:

```
>>> dir(str)
```

Moar resources!

Last term's 108 and 148 course websites:

<http://www.cdf.utoronto.ca/~csc108h/summer/>

<http://www.cdf.utoronto.ca/~csc148h/summer/>

Software Carpentry (online lectures):

http://software-carpentry.org/4_0/python/

Google!

<http://imgtfy.com/?q=python+add+to+list>

Learn you speak good Python

Python's style guide:

`http://www.python.org/dev/peps/pep-0008/`

Google's Python style guide:

`http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`

Expert modes:

`pychecker: http://pychecker.sourceforge.net/`

`pyflakes: https://launchpad.net/pyflakes/`

Variables (storing data)

Variables refer to an **object** of some **type**

Several basic data types:

- Integers (whole numbers): `int`

```
>>> the_answer = 42
```

- Floating-point (decimal) numbers: `float`

```
>>> pi = 3.14159
```

```
>>> radius = 2.0
```

```
>>> pi * (radius ** 2)
```

```
12.56636
```

- operators: `*` `/` `%` `+` `-` `**`

- "shortcut" operators: `x = x + 1` \rightarrow `x += 1`

More types (kinds of things)

- Boolean (True/False) values: `bool`

```
>>> passed = False
```

```
>>> not passed
```

```
True
```

```
>>> 5 < 4 # comparisons return bool
```

```
False
```

```
>>> 5 and 4 # this can bite you
```

```
4
```

- Operators: `and` `or` `not`

More types (kinds of things)

- None (it's Python's NULL): `NoneType`

```
>>> x = None
```

```
>>> print(x)
```

```
None
```

```
>>> x
```

```
>>> # None is a little weird
```

Strings

- Strings (basically lists of characters): `str`

```
>>> welcome = "Hello, world!"
```

```
>>> welcome[1]    # index, starting with 0  
'e'
```

- Slices return sub-strings:

```
>>> welcome[1:5] # slice with [start:end]  
'ello'
```

```
>>> welcome[:3]  # start defaults to 0  
'wel'
```

```
>>> welcome[9:]  # end defaults to None (wtf?)  
'rld!'
```

```
>>> welcome[:-1] # index/slice with negatives  
'Hello, world'
```

Working with strings

- Stick strings together (concatenation):

```
>>> salutation = "Hello, "  
>>> name = "Orion"  
>>> salutation + name # evaluates to a new string  
'Hello, Orion'
```

- The `len` function is useful:

```
>>> len(name) # number of characters  
5
```

Tons of useful methods

- Here are some, look at `help(str)` for more:

```
>>> name = "Orion"
```

```
>>> name.lower()
```

```
'orion'
```

```
>>> name.endswith('ion')
```

```
True
```

```
>>> name.startswith('orio')
```

???? Thoughts?

```
>>> name.index('i')
```

```
2 # What did this do? Try help(str.index)
```

POP QUIZ!

Write a boolean expression that evaluates to:

`True` if the variable `response` starts with the letter "q", case-insensitive, or

`False` if it does not.

(in CS lingo, we'd say: `True` iff (if and only if) the variable `response` starts with the letter "q", case-insensitive)

POP QUIZ!

```
response.lower().startswith('q')
```

A little more on strings

```
>>> name = 'Orion'
```

```
>>> name[1] = 'n'
```

???? Thoughts?

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'str' object does not support item  
assignment
```

- Strings are **immutable**, meaning they can't be changed once created
- Empty strings are OK:

```
>>> what_i_have_to_say = ''
```


Making strings pretty

- String formatting (`str.format`):
 - <http://docs.python.org/release/3.1.5/library/string.html#formatstrings>
 - `{ }` are replaced by the arguments to format
 - Formatting parameters can be specified using `:format`
 - Similar to `printf`

```
>>> n = 99
>>> where = "on the wall"
>>> '{} bottles of {}'.format(n, where)
'99 bottles of beer on the wall'
```

Standard input/output

- Generating output (stdout): `print()`
 - Can take multiple arguments (will be joined with spaces)
- Reading keyboard input: `input()`

```
>>> name = input()
>>> name
'Orion'
>>> print("Hello " + name)
Hello Orion
>>> "Hello {}".format(name)
'Hello Orion' # Why quotes here?
```

Converting between types

- AKA: how to sanitize user input
- Functions: `int()`, `float()`, `str()`, `bool()`

```
>>> float('3.14')
```

```
3.14
```

```
>>> int(9 / 5)    # truncates
```

```
1
```

```
>>> float(3)
```

```
3.0
```

```
>>> str(3.14)
```

```
'3.14'
```

```
>>> '{:.4f}'.format(3.14159265358)
```

```
'3.1416'
```

Converting between types

- Don't do anything silly:

```
>>> int('fish')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base
10: 'fish'
```

- And beware:

```
>>> int('3.0')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base
10: '3.0'
```

Exercise 1: Temperature

$$C = (5 / 9) * (F - 32)$$

- Write a program that:
 - prompts the user for degrees in Fahrenheit
 - converts the number into Celsius
 - prints out the number in Celsius
 - to just 2 decimal places, if you dare

(You can assume the user enters a number)

Exercise 1: Solution

Self-check: does your code work for 98.6?

```
# Prompt the user
print("Input temperature (F):")

# Read in the input
fahrenheit = float(input("--> "))

# Convert to Celsius
celsius = (5 / 9) * (fahrenheit - 32)

# Display the answer
print("Temperature is {:.2f} degrees C".format(celsius))
```

Sequences, of, things!

There are two main kinds of sequences (things in an order) in Python:

- The **[mighty]** list
- The (humble,) tuple

[Lists, of, things]

- Lists are a very important data structure in Python
- They are a **mutable sequence of any objects**

```
>>> colours = ['cyan', 'magenta', 'yellow']
>>> friends = [] # forever alone
>>> random_stuff = [42, 3.14, 'carpe diem']
>>> wtf = [[], [2, 3], friends] # this is crazy
>>> my_friends = list(friends) # copy a list
```

- Index and slice like strings:

```
>>> colours[0] # indexing returns the element
'cyan'
>>> random_stuff[2:] # slicing returns a sub-list
['carpe diem']
```


[Lists, of, things].stuff()

- We can change, add, and remove elements from lists

```
>>> marks = [98, None, 62, 54]
```

```
>>> marks[1] = 75 # change that None
```

```
>>> marks.append(90)
```

```
>>> marks.remove(62)
```

```
>>> marks.sort()
```

```
>>> print(marks)
```

??? **Thoughts?**

```
[54, 75, 90, 98]
```

Variable aliasing

- Careful! Multiple variables might be referring to the **same** data structure:

```
>>> sorted_list = [1, 2, 3]
>>> copy_list = sorted_list # not a copy
>>> copy_list.append(0)
>>> sorted_list
[1, 2, 3, 0] # crap
```

- Or worse...

```
>>> nested_list = [sorted_list, sorted_list]
>>> nested_list[0][0] = 4
>>> nested_list
```

??? Thoughts?

(Tuples, of, things)

- Tuples are like fast, simple lists, that are **immutable**

```
>>> stuff = (42, 3.14, 'carpe diem')
```

```
>>> stuff[0] = 'a'
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item  
assignment
```

- Can always create a list from them:

```
>>> L = list(stuff)
```

- `>.<` A little weird to get a 1-element tuple:

```
('a') -> 'a'
```

```
('a',) -> ('a',)
```

For loops! (you spin me right round baby...)

- **For loops** repeat some code for **each** element in a sequence
 - This is a foreach loop in most languages

```
>>> colours = ['red', 'green', 'blue']
>>> for colour in colours:
...     print(colour)
...
red
green
blue
```

For loops! (you spin me right round baby...)

- But wait, I actually *wanted* the index!
 - Use **range** (n) in a for loop to loop over a range.

```
>>> for i in range(2):  
...     print(i)  
0  
1
```

- To start at a value other than 0:

```
>>> for i in range(4, 6):  
...     print(i)  
4  
5
```

For loops! (you spin me right round baby...)

- But wait, I actually *wanted* the index!
 - Now, over the indices in a list:

```
>>> for i in range(len(colours)):  
...     print("{} . {}".format(i, colours[i]))  
...  
0. red  
1. green  
2. blue
```

Exercise 2: Times table

Compute (and store in a variable) a times table for the numbers 0 through 9 as a **list of lists**.

For example, if it were just from 0 through 2, you should create:

```
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

Exercise 2: Solution

```
table = []
n = 10 # from 0 to (n - 1)
for i in range(n):
    # Compute the n'th row
    row = []
    for j in range(n):
        row.append(i * j)

    # Add row to full table
    table.append(row)
```


Exercise 2: Alternate solution

```
table = [[row * col for col in range(10)]  
         for row in range(10)]
```

(list comprehensions FTW!)

Eat ALL the things...

Conditionals (if, elif, else)

- **If statements** allow you to execute code sometimes (based upon some **condition**)
- **elif** (meaning 'else if') and **else** are optional

```
if amount > balance:
    print("You have been charged a $20"
          " overdraft fee. Enjoy.")
    balance -= 20
elif amount == balance:
    print("You're now broke")
else:
    print("Your account has been charged")

balance -= amount # deduct amount from account
```

Functions (basically the best things ever)

- They allow you to group together a bunch of statements into a block that you can call.
- "If you have the same code in two places, it will be wrong in one before long."
- "Never copy-paste code if at all possible."
- They can take in information (**arguments**) and give back information (**return value**).
- **Important:** If you don't specify a return value, it will be None

```
def celsius_to_fahrenheit(degrees):  
    return (9 / 5) * degrees + 32
```

```
f = celsius_to_fahrenheit(100)
```

Docstrings

- They should have a **docstring** (a multi-line, meaning triple-quoted, string right after the declaration) that describes **what** the function does, **not how** it does it.
- It should describe the argument and return types.
- It is shown when **help** is called on your function, so it should be sufficient for other people to know how to use your function.

```
def celsius_to_fahrenheit(degrees):  
    """(int or float) -> float  
    Convert degrees from C to F.  
    """  
    return (9 / 5) * degrees + 32
```

Changing things

- Functions can modify mutable arguments

```
def double(L):  
    """list -> NoneType  
    Modify L so it is equivalent to L + L  
    """  
    for i in range(len(L)):  
        L.append(L[i])
```

```
L = [1, 2, 3]
```

```
L = double(L) # Don't do this! Why?
```

What should we do?

Changing things

- Functions can modify mutable arguments
- **If no return is specified, None is returned**

```
def double(L):  
    """list -> NoneType  
    Modify L so it is equivalent to L + L  
    """  
    for i in range(len(L)):  
        L.append(L[i])
```

```
L = [1, 2, 3]  
double(L)  
print(L)    # [1, 2, 3, 1, 2, 3]
```

Exercise 3: Functions

Two words are a reverse pair if each word is the reverse of the other.

1) Write a function `is_reverse_pair(s1, s2)` that returns `True` iff `s1` and `s2` are a reverse pair.

2) Write a function `print_reverse_pairs(wordlist)` that accepts a list of strings and prints out all of the reverse pairs in the given list, each pair on a line.

Exercise 3: Solution

```
def is_reverse_pair(s1, s2):  
    if len(s1) != len(s2):  
        return False  
  
    for i in range(len(s1)):  
        if s1[i] != s2[len(s2) - 1 - i]:  
            return False  
  
    return True
```

Or, using slicing:

```
def is_reverse_pair(s1, s2):  
    return s1[::-1] == s2
```

Exercise 3: Solution

```
def print_reverse_pairs(wordlist):  
    for s1 in wordlist:  
        for s2 in wordlist:  
            if is_reverse_pair(s1, s2):  
                print("{} , {}".format(s1, s2))
```

{'dictionaries': 'awesome'}

- **Dictionaries** (type `dict`) are an **unordered** association of **keys** with **values**
- We usually use them to store associations:
 - like name -> phone number
 - phone number -> name
 - student id -> grade

{'dictionaries': 'awesome'}

- **Dictionaries** (type `dict`) are an **unordered** association of **keys** with **values**
- We usually use them to store associations:
 - like name -> phone number
 - phone number -> name
 - student id -> grade
 - grade -> student id **#BAD, why?**

{'dictionaries': 'awesome'}

- **Dictionaries** (type `dict`) are an **unordered** association of **keys** with **values**
- We usually use them to store associations:
 - like name -> phone number
 - phone number -> name
 - student id -> grade
 - grade -> list of student ids
- Keys must be **unique** and **immutable**

{'dictionaries': 'awesome'}

```
>>> scores = {'Alice': 90, 'Bob': 76, 'Eve': 82}
>>> scores['Alice'] # get
90
>>> scores['Charlie'] = 64 # set
>>> 'Bob' in scores # membership testing
True
>>> for name in scores: # loops over keys
...     print("{}: {}".format(name, scores[name]))
...
Charlie: 64
Bob: 76
Alice: 88
Eve: 82
```

Exercise 4: Dictionaries

- Write a function `print_record` that takes a dictionary as input. Keys are student numbers (`int`), values are names (`str`). The function should print out all records, nicely formatted.

```
>>> record = {1234: 'Tony Stark', 1138: 'Steve Rogers'}
>>> print_record(record)
Tony Stark (#1234)
Steve Rogers (#1138)
```

hint: "in" keyword

- Write a function `count_occurrences` that takes a list of strings as input, and returns a dictionary with key/value pairs of each word and the number of occurrences of that word.

```
>>> count_occurrences(['a', 'b', 'a', 'a', 'c', 'c'])
{'a': 3, 'b': 1, 'c': 2}
```

Exercise 4: Solution

```
def print_record(record):  
    for pin in record:  
        print('{} (#{})'.format(record[pin], pin))
```


Exercise 4: Solution

```
def count_occurrences(words):  
    counts = {}  
    for word in words:  
        if word in counts:  
            counts[word] += 1  
        else:  
            counts[word] = 1  
  
    return counts
```

While loops (right round right round...)

- **While loops** keep repeating a block of code while a condition is `True`

What does this code do?

```
val = 10
divisor = 2
i = 0
while val > divisor:
    val /= divisor
    i += 1
```

While loops (right round right round...)

- **While loops** keep repeating a block of code while a condition is `True`

What does this code do?

```
val = 167
while val > 0:
    if val % 2 == 0:
        print("0")
    else:
        print("1")
    val = int(val / 2)
```

prints (reverse) binary representation of val

While loops (right round right round...)

- **break** can be used to exit a loop early

What does this code do?

```
while True: # This is an infinite loop
    # Stop when the user types 'quit', 'Q', etc.
    response = input("Enter number or 'quit':")
    if response.lower().startswith('q'):
        break # This breaks out of the loop

...

```

Modules (why reinvent the wheel?)

Python has a spectacular assortment of **modules** that you can use (you have to import their **names** first, though)

```
>>> from random import randint # now we can use it!
>>> randint(1, 6) # roll a die
4 # http://xkcd.com/221/
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.cos(0)
1.0
>>> import datetime
>>> dir(datetime)
???
```

Exercise 5: Guessing game

Implement a guessing game:

Guess a number between 0 and 100: 50

Too high.

Guess a number between 0 and 100: 25

Too low.

Guess a number between 0 and 100: 40

Too low.

Guess a number between 0 and 100: -2

Guess a number between 0 and 100: 47

Correct. Thanks for playing!

hint: "random" module

Exercise 5: Solution

```
from random import randint
# Choose a random number
low = 0
high = 100
answer = randint(low, high)

found = False
while not found:
    print("Guess a number between {} and {}: "
          "{}.format(low, high), end="")
    guess = int(input())
    # Print response if guess is in range
    if guess >= low and guess <= high:
        if guess > answer:
            print("Too high.")
        elif guess < answer:
            print("Too low.")
        else:
            print("Correct. Thanks for playing!")
            found = True # Or you could use break here
```

Classes and objects - philosophy

- Classes are descriptions of types of things (like a blueprint), and objects are specific instances of a type (like the actual building).
- Objects have associated state (variables) and behavior (methods).
- We usually want to hide the implementation as much as possible, so that the people using our classes don't need to know how they are implemented, and so they don't go mucking around where they shouldn't.
- These will be discussed in much more detail in 148.

Classes and objects - simple e.g.

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        """Translate the point by dx and dy"""
        self.x += dx
        self.y += dy
```

Then we can make a Point object and use it!

```
where_i_am = Point(0, 0) # or Point(), since defaults
where_i_am.translate(5, -2)
print((where_i_am.x, where_i_am.y)) # (5, -2)
```

Classes and objects - complex e.g.

As a user of the iPhone class, we usually don't want to know what goes on under the surface. And Apple certainly doesn't want us messing around with what's inside (we might screw things up!). So this is how we might think about a class as a client:

```
class iPhone(object):
    def __init__(self):
        """Initialize the iPhone"""
        ...
    def call(self, phone_number):
        """Call the given phone number"""
        ...
    def kill_switch(self, authorization_code):
        """Brick the iPhone"""
        ...
```

Then we can make an iPhone object and use it!

```
precious = iPhone()
precious.call('123.456.7890')
```

Classes and objects - complex e.g.

As a developer, we want to hide the implementation as much as possible. This lets us change our implementation later without breaking everything!

```
class iPhone(object):
    def __init__(self):
        """Initialize the iPhone"""
        # Private attributes start with an underscore "_"
        self._call_timer = 0
        self._recent_calls = []
        self._network = RogersNetwork(self)

    def call(self, phone_number):
        """Call the given phone number"""
        self._recent_calls.append(phone_number)
        self._network.connect(phone_number)
```

Exercise 6: NumberList

Write a class that stores a list of integers/floats and provides the following methods:

`sum()` - return the sum of the list

`mean()` - return the average of the list as a float

`min()/max()` - return the maximum/minimum element

`num_unique()` - return the number of unique elements in the list

For example:

```
>>> nl = NumberList([1, 2, 5, 1, 4, 3, 3])
```

```
>>> nl.sum()
```

```
19
```

```
>>> nl.num_unique()
```

```
5
```

Hint: Use the in keyword:

```
>>> nums = [1, 3, 9, 16]
```

```
>>> 3 in nums
```

```
True
```

```
>>> 7 in nums
```

```
False
```

Exercise 6: Solution

```
class NumberList(object):  
    def __init__(self, L):  
        self._L = L  
  
    def sum(self):  
        result = 0  
        for value in self._L:  
            result += x  
  
        return result  
  
    def mean(self):  
        n = len(self._L)  
        return self.sum() / n
```

Exercise 6: Solution

...

```
def max(self):  
    result = None  
    for value in self._L:  
        if result is None or x > result:  
            result = x  
  
    return result
```

Exercise 6: Solution

...

```
def num_unique(self):  
    # One of many possible solutions  
    # Also: return len(set(self._L))  
  
    seen = []  
    for value in self._L:  
        if value not in seen:  
            seen.append(value)  
  
    return len(seen)
```

fin