Plan For Python Lecture 2

- For Loops and List Comprehensions
- Generators
- Imports
- Functions
  - *args, **kwargs, first class functions
- Classes
  - inheritance
  - “magic” methods (objects behave like built-in types)
- Profiling
  - timeit
  - cProfile
- Idioms
For Loops and List Comprehensions
For Loops

```
for <item> in <collection>:
    <statements>
```

- If you’ve got an existing list, this iterates each item in it.
- You can generate a list with `Range`:
  - `list(range(5))` returns `[0,1,2,3,4]`
  - So we can say:
    ```
    for x in range(5):
        print(x)
    ```
- `<item>` can be more complex than a single variable name.
  ```
  for (x, y) in [('a',1), ('b',2), ('c',3), ('d',4)]:
      print(x)
  ```
List Comprehensions replace loops!

```python
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# I want 'n*n' for each 'n' in nums
squares = []
for n in nums:
    for n in nums:
        squares.append(x*x)
print(squares)

squares = [x*x for x in nums]
print(squares)
```
List Comprehensions replace loops!

```python
>>> li = [3, 6, 2, 7]
>>> [elem * 2 for elem in li]
[6, 12, 4, 14]
```

```python
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]
```
Filtered List Comprehensions

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.
List Comprehension extra *for*

```
lst1, lst2, lst3 = [1, 2, 3], [2, 3, 4], [3, 4, 5]

res = [(x, y, z) for x in lst1 if x < 2 \   
    for y in lst2 \   
    for z in lst3 if x + y + z < 8]
```

```
res = [] # translation
for x in lst1:
    if x < 2:
        for y in lst2:
            for z in lst3:
                if x + y + z < 8:
                    res.append((x, y, z))
```

# Both value of res: [(1, 2, 3), (1, 2, 4), (1, 3, 3)]
Dictionary, Set Comprehensions

```python
lst1 = [('a', 1), ('b', 2), ('c', 'hi')]
lst2 = ['x', 'a', 6]

d = {k: v for k, v in lst1}
s = {x for x in lst2}

d = dict() # translation
for k, v in lst1:
    d[k] = v
s = set() # translation
for x in lst:
    s.add(x)

# Both value of d: {'a': 1, 'b': 2, 'c': 'hi'}
# Both value of d: {'x', 'a', 6}
```
Iterators
Iterator Objects

- Iterable objects can be used in a for loop because they have an `__iter__` magic method, which converts them to iterator objects:

```python
>>> k = [1, 2, 3]

>>> k.__iter__()
<list_iterator object at 0x104f8ca50>

>>> iter(k)
<list_iterator object at 0x104f8ca10>
```
Iterators

- Iterators are objects with a `__next__()` method:

```python
>>> i = iter(k)
>>> next(i)
1
>>> i.__next__()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- Python iterators do not have a `hasnext()` method!

- Just catch the `StopIteration` exception
Iterators: The real truth about For.. In..

- for <item> in <iterable>:
  <statements>

- First line is just syntactic sugar for:
  1. Initialize: Call <iterable>.__iter__() to create an iterator
     Each iteration:
     2. Call iterator.__next__() and bind <item>
     2a. Catch StopIteration exceptions

- To be iterable: has __iter__ method
  which returns an iterator obj

- To be iterator: has __next__ method
  which throws StopIteration when done
An Iterator Class

class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

    def __iter__(self):
        return self

>>> for char in Reverse('spam'):
    print(char)
Iterators use memory efficiently

Eg: File Objects

```python
>>> for line in open("script.py"): # returns iterator
...     print(line.upper())
...

import sys
print(sys.path)
x = 2
print(2 ** 3)
```

instead of

```python
>>> for line in open("script.py").readlines(): #returns list
...     print(line.upper())
...
```
Generators
Generators: using `yield`

- Generators are iterators (with `__next()` method)
- Creating Generators: `yield`
  - Functions that contain the `yield` keyword *automatically* return a generator when called

```python
>>> def f(n):
    ...   yield n
    ...   yield n+1
    ...

>>> type(f)
<class 'function'>
>>> type(f(5))
<class 'generator'>
>>> [i for i in f(6)]
[6, 7]
```
Generators: What does `yield` do?

- Each time we call the `__next__` method of the generator, the method runs until it encounters a `yield` statement, and then it stops and returns the value that was yielded. Next time, it resumes where it left off.

```python
>>> gen = f(5)  # no need to say f(5).__iter__()
>>> gen
<generator object f at 0x1008cc9b0>
>>> gen.__next__()
5
>>> next(gen)
6
>>> gen.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```
Generators

- **xrange(n) vs range(n) in Python 2**
  - `xrange` acts like a generator
  - `range(n)` keeps all n values in memory before starting a loop even if n is huge: `for k in range(n)`
  - `sum(xrange(n))` much faster than `sum(range(n))` for large n

- **In Python 3**
  - `xrange(n)` is removed
  - `range(n)` acts similar to the old `xrange(n)`
  - Can use list() to get similar behavior as in Python 2
  - Python 3’s range is more powerful than Python 2’s xrange
Generators

- **Benefits of using generators**
  - Less code than writing a standard iterator
  - Maintains local state automatically
  - Values are computed one at a time, as they’re needed
  - Avoids storing the entire sequence in memory
  - Good for aggregating (summing, counting) items. One pass.
  - Crucial for infinite sequences
  - Bad if you need to inspect the individual values
Using generators: merging sequences

- Problem: merge two sorted lists, using the output as a stream (i.e. not storing it).

```python
def merge(l, r):
    llen, rlen, i, j = len(l), len(r), 0, 0
    while i < llen or j < rlen:
        if j == rlen or (i < llen and l[i] < r[j]):
            yield l[i]
            i += 1
        else:
            yield r[j]
            j += 1
```
Using generators

```python
>>> g = merge([2,4], [1, 3, 5])  # g is an iterator
>>> while True:
...    print(g.__next__())
... 1
2
3
4
5
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
StopIteration

>>> [x for x in merge([1,3,5],[2,4])]
[1, 2, 3, 4, 5]
```
Generators and exceptions

```python
>>> g = merge([2,4], [1, 3, 5])
>>> while True:
...    try:
...        print(g.__next__())
...    except StopIteration:
...        print('Done')
...        break
...
1
2
3
4
5
Done
```
Generator comprehensions

- **Review**: generators are good for aggregating items.

- For example, in Python 2, `sum(xrange(n))` was much faster than `sum(range(n))` for large `n`.

- Similarly,

  ```python
  >>> sum(x for x in xrange(10**8) if x%5==0)
  999999950000000L
  which uses a *generator comprehension* is much faster than
  ```

  ```python
  >>> sum([x for x in xrange(10**8) if x%5==0])
  999999950000000L
  which creates the entire list before computing the sum
  ```
Imports
Import Modules and Files

```python
>>> import math
>>> math.sqrt(9)
3.0

# Not as good to do this:
>>> from math import *
>>> sqrt(9)  # unclear where function defined

>>> import queue as Q
>>> q = Q.PriorityQueue()
>>> q.put(10)
>>> q.put(1)
>>> q.put(5)
>>> while not q.empty():
    print q.get(),
1, 5, 10
```

Hint: Super useful for search algorithms
Import Modules and Files

# homework1.py
def concatenate(seqs):
    return [seq for seq in seqs] # This is wrong

# run python interactive interpreter (REPL) in directory of homework1.py
>>> import homework1
>>> assert homework1.concatenate([[1, 2], [3, 4]]) ==
[1, 2, 3, 4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError

>>> import importlib # after fixing homework1
>>> importlib.reload(homework1)
Functions

\[ f(x) \]
Defining Functions

Function definition begins with `def`. Function name and its arguments.

```python
def get_final_answer(filename):
    """Documentation String""
    line1
    line2
    return total_counter
```

First line with less indentation is considered to be outside of the function definition.

'`return`' indicates the value to be sent back to the caller.

No declaration of `types` of arguments or result.
Function overloading? No.

- There is no function overloading in Python 2
  - Unlike Java, a Python function is specified by its name alone
  - Two different functions can’t have the same name, even if they have different numbers, order, or names of arguments
  - But operator overloading – overloading +, ==, -, etc. – is possible using special methods on various classes

- In Python 3.4, partial support
  - Python 3 – Function Overloading with singledispatch
Default Values for Arguments

- You can provide default values for a function’s arguments
- These arguments are optional when the function is called

```python
>>> def myfun(b, c=3, d="hello"): return b + c
```

```python
>>> myfun(5,3,"bob")
8
>>> myfun(5,3)
8
>>> myfun(5)
8
```

- Non-default argument should always precede default arguments; otherwise, it reports `SyntaxError`
Keyword Arguments

- Functions can be called with arguments out of order
- These arguments are specified in the call
- Keyword arguments can be used after all other arguments.

>>> def myfun(a, b, c):
    return a - b

>>> myfun(2, 1, 43)  # 1
>>> myfun(c=43, b=1, a=2)  # 1
>>> myfun(2, c=43, b=1)  # 1
>>> myfun(a=2, b=3, 5)
   File "<stdin>", line 1
   SyntaxError: positional argument follows keyword argument
*args

- Suppose you want to accept a variable number of non-keyword arguments to your function.

```python
def print_everything(*args):
    
    for count, thing in enumerate(args):
        print('{0}. {1}'.format(count, thing))

>>> lst = ['a', 'b', 'c']
>>> print_everything('a', 'b', 'c')
0. a
1. b
2. c

>>> print_everything(*lst)  # Same results as above
```
**kwargs

Suppose you want to accept a variable number of keyword arguments to your function.

def print_keyword_args(**kwargs):
    # kwargs is a dict of the keyword args passed to the fn
    for key, value in kwargs.items():  # .items() is list
        print("%s = %s" % (key, value))

>>> kwargs = {'first_name': 'Bobby', 'last_name': 'Smith'}
>>> print_keyword_args(**kwargs)
first_name = Bobby
last_name = Smith

>>> print_keyword_args(first_name="John", last_name="Doe")
first_name = John
last_name = Doe
Python uses dynamic scope

- Function sees the most current value of variables

```python
>>> i = 10
>>> def add(x):
...     return x + i

>>> add(5)
15
>>> i = 20
>>> add(5)
25
```
Default Arguments & Memoization

- **Default parameter values are evaluated only when the `def` statement they belong to is first executed.**
- The function uses the same default object each call

```python
def fib(n, fibs={}):
    if n in fibs:
        print('n = %d exists' % n)
        return fibs[n]
    if n <= 1:
        fibs[n] = n  # Changes fibs!!
    else:
        fibs[n] = fib(n-1) + fib(n-2)
    return fibs[n]

>>> fib(3)
n = 1 exists
2
```
Functions are “first-class” objects

- First class object
  - An entity that can be dynamically created, destroyed, passed to a function, returned as a value, and have all the rights as other variables in the programming language have

- Functions are “first-class citizens”
  - Pass functions as arguments to other functions
  - Return functions as the values from other functions
  - Assign functions to variables or store them in data structures

- Higher order functions: take functions as input

```python
def compose(f, g, x):
    return f(g(x))
```

```bash
>>> compose(str, sum, [1, 2, 3])
'6'
```
Higher Order Functions: Map, Filter

```python
>>> [int(i) for i in ['1', '2']]
[1, 2]
```

```python
>>> list(map(int, ['1', '2'])) # equivalent to above
```

```python
def is_even(x):
    return x % 2 == 0
```

```python
>>> [i for i in [1, 2, 3, 4, 5] if is_even(i)]
[2, 4]
```

```python
>>> list(filter(is_even, [1, 2, 3, 4, 5])) # equivalent
```

```python
>>> lambda x: x%2==0
```

```python
>>> list(filter(lambda x: x%2==0, [1, 2, 3, 4, 5])) # also equivalent
```
from operator import itemgetter

def calc_ngram(inputstring, nlen):
    ngram_list = [inputstring[x:x+nlen] for x in \n        range(len(inputstring) - nlen + 1)]
    ngram_freq = {}  # dict for storing results
    for n in ngram_list:  # collect the distinct n-grams and count
        if n in ngram_freq:
            ngram_freq[n] += 1
        else:
            ngram_freq[n] = 1  # human counting numbers start at 1
    # Can set reverse to change order of sort
    # (reverse=True for ascending; reverse=False for descending)
    return sorted(ngram_freq.items(), \n        key=itemgetter(1), reverse=True)
Classes and Inheritance
Creating a class

```python
class Student:

    univ = "upenn"  # class attribute

    def __init__(self, name, dept):
        self.student_name = name
        self.student_dept = dept

    def print_details(self):
        print("Name: " + self.student_name)
        print("Dept: " + self.student_dept)

student1 = Student("julie", "cis")
student1.print_details()
Student.print_details(student1)
Student.univ
```

Called when an object is instantiated

Every method begins with the variable self

Another member method

Creating an instance, note no self

Calling methods of an object
Subclasses

- A class can **extend** the definition of another class
  - Allows use (or extension) of methods and attributes already defined in the previous one.
  - New class: **subclass**. Original: **parent**, **ancestor** or **superclass**

- To define a subclass, put the name of the superclass in parentheses after the subclass’s name on the first line of the definition.

  ```python
  class AiStudent(Student):
  ```

- Python has no ‘extends’ keyword like Java.
- Multiple inheritance is supported.
Constructors: __init__

- Very similar to Java
- Commonly, the ancestor’s __init__ method is executed in addition to new commands
- *Must be done explicitly*
- You’ll often see something like this in the __init__ method of subclasses:

  ```python
  parentClass.__init__(self, x, y)
  ```

  where `parentClass` is the name of the parent’s class

  ```python
  Student.__init__(self, x, y)
  ```
Redefining Methods

- Very similar to over-riding methods in Java

- To *redefine a method* of the parent class, include a new definition using the same name in the subclass.
  - The old code in the parent class won’t get executed.

- To execute the method in the parent class *in addition to* new code for some method, explicitly call the parent’s version of the method.

  ```python
  parentClass.methodName(self, a, b, c)
  ```

  - The only time you ever explicitly pass `self` as an argument is when calling a method of an ancestor.

  So use `myOwnSubClass.methodName(a, b, c)`
Multiple Inheritance can be tricky

class A(object):
    def foo(self):
        print('Foo!')

class B(object):
    def foo(self):
        print('Foo?')
    def bar(self):
        print('Bar!')

class C(A, B):
    def foobar(self):
        super().foo()  # Foo!
        super().bar()  # Bar!
Special Built-In Methods and Attributes
Magic Methods and Duck Typing

- **Magic Methods** allow user-defined classes to behave like built in types

- **Duck typing** establishes suitability of an object by determining presence of methods
  - Does it swim like a duck and quack like a duck? It’s a duck
  - Not to be confused with ‘rubber duck debugging’
Magic Methods and Duck Typing

class Duck:
    def fly(self):
        print("Duck flying")

class Airplane:
    def fly(self):
        print("Airplane flying")

class Whale:
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

duck = Duck()
airplane = Airplane()
whale = Whale()
lift_off(duck)  # prints `Duck flying`
lift_off(airplane)  # prints `Airplane flying`
lift_off(whale)  # Throws the error `'Whale' object has no attribute 'fly'`
Example Magic Method

class Student:
    def __init__(self, full_name, age):
        self.full_name = full_name
        self.age = age

    def __str__(self):
        return "I'm named " + self.full_name + " - age: " + str(self.age)
    ...

>>> f = Student("Bob Smith", 23)

>>> print(f)
I'm named Bob Smith - age: 23
Other “Magic” Methods

- Used to implement operator overloading
  - Most operators trigger a special method, dependent on class

  ```python
  __init__ : The constructor for the class.
  __len__  : Define how len(obj) works.
  __copy__ : Define how to copy a class.
  __cmp__  : Define how == works for class.
  __add__  : Define how + works for class
  __neg__  : Define how unary negation works for class
  ```

- Other built-in methods allow you to give a class the ability to use [ ] notation like an array or ( ) notation like a function call.
Profiling, function level

- **Rudimentary**

  ```python
  >>> import time
  >>> t0 = time.time()
  >>> code_block
  >>> t1 = time.time()
  >>> total = t1-t0
  ```

- **Timeit (more precise)**

  ```python
  >>> import timeit
  >>> t = timeit.Timer("<statement to time>",
                   "<setup code>")
  >>> t.timeit()
  ```

  - The second argument is usually an import that sets up a virtual environment for the statement
  - `timeit` calls the statement 1 million times and returns the total elapsed time, the `number` argument specifies number of times to run it.
Profiling, script level 1

```python
# to_time.py

def get_number():
    for x in range(500000):
        yield x

def exp_fn():
    for x in get_number():
        i = x ^ x ^ x
    return 'some result!'

if __name__ == '__main__':
    exp_fn()
```
# python interactive interpreter (REPL)

$ python -m cProfile to_time.py

500004 function calls in 0.203 seconds

Ordered by: standard name

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.203</td>
<td>0.203</td>
<td>to_time.py:1(&lt;module&gt;)</td>
</tr>
<tr>
<td>500001</td>
<td>0.071</td>
<td>0.000</td>
<td>0.071</td>
<td>0.000</td>
<td>to_time.py:1(get_number)</td>
</tr>
<tr>
<td>1</td>
<td>0.133</td>
<td>0.133</td>
<td>0.203</td>
<td>0.203</td>
<td>to_time.py:5(exp_fn)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>{method 'disable' of '_lsprof.Profiler' objects}</td>
</tr>
</tbody>
</table>

- For details see https://docs.python.org/3.7/library/profile.html
Idioms

- Many frequently-written tasks should be written Python-style even though you could write them Java-style in Python.
- Remember beauty and readability!

- A list of anti-patterns: [http://lignos.org/py_antipatterns/](http://lignos.org/py_antipatterns/)
A directed graph class

```python
>>> d = DiGraph([(1,2), (1,3), (2,4), (4,3), (4,1)])
>>> print(d)
1 -> 2
1 -> 3
2 -> 4
4 -> 3
4 -> 1
```
A directed graph class

```python
>>> d = DiGraph([(1, 2), (1, 3), (2, 4), (4, 3), (4, 1)])
>>> [v for v in d.search(1, set())]
[1, 2, 4, 3]
>>> [v for v in d.search(4, set())]
[4, 3, 1, 2]
>>> [v for v in d.search(2, set())]
[2, 4, 3, 1]
>>> [v for v in d.search(3, set())]
[3]
```

The `search` method returns a generator for the nodes that can be reached from a given node by following arrows “from tail to head”
class DiGraph:
    def __init__(self, edges):
        self.adj = {}
        for u, v in edges:
            if u not in self.adj: self.adj[u] = [v]
            else: self.adj[u].append(v)

    def __str__(self):
        return '\n'.join(['%s -> %s' % (u, v) \n                          for u in self.adj for v in self.adj[u]])

>>> d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
>>> d.adj
{1: [2, 3], 2: [4], 4: [3, 1]}

The constructor builds a dictionary (self.adj) mapping each node name to a list of node names that can be reached by following one edge (an “adjacency list”)
The search method

class DiGraph:
...

def search(self, u, visited):
    # If we haven't already visited this node...
    if u not in visited:
        # yield it
        yield u
        # and remember we've visited it now.
        visited.add(u)
    # Then, if there are any adjacent nodes...
    if u in self.adj:
        # for each adjacent node...
        for v in self.adj[u]:
            # search for all nodes reachable from *it*...
            for w in self.search(v, visited):
                # and yield each one.
                yield w