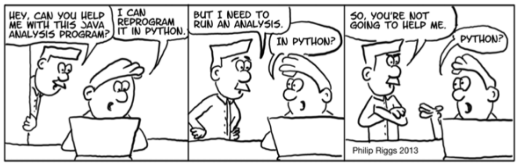
Plan For Python Lecture 2

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

Review

```
>>>import  this
The  Zen  of  Python,  by  Tim  Peters
```


HEY, CAN YOU HELP ME WITH THIS JAVA ANALYSIS PROGRAM?
I CAN REPROGRAM IT IN PYTHON.
BUT I NEED TO RUN AN ANALYSIS.
IN PYTHON?
SO, YOU'RE NOT GOING TO HELP ME.
PYTHON?
Philip Riggs 2013

List Comprehensions

```
[<statement> for <item> in <iterable> if <condition>]

#Translation
lst = []
for <item> in <iterable>:
    if <condition>:
        lst.append(<statement>)

>>> li = [(‘a’, 1), (‘b’, 2), (‘c’, 7)]
>>> [ n * 3 for (x, n) in li if x == ‘b’ or x == ‘c’]
[6, 21]

#Translation
lst = []
for (x,n) in li:
    if x == ‘b’ or x == ‘c’:
        lst.append(n*3)
```

List Comprehension extra for

```
[x 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)
```

Iterators use memory efficiently

- **Iterators are objects with a next() method:**
- **To be iterable: __iter__()**
- **To be iterators: next()**

```
>>> k = [1,2,3]
>>> i = iter(k) # could also use k.__iter__()
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
StopIteration
```

Generators (are iterators)

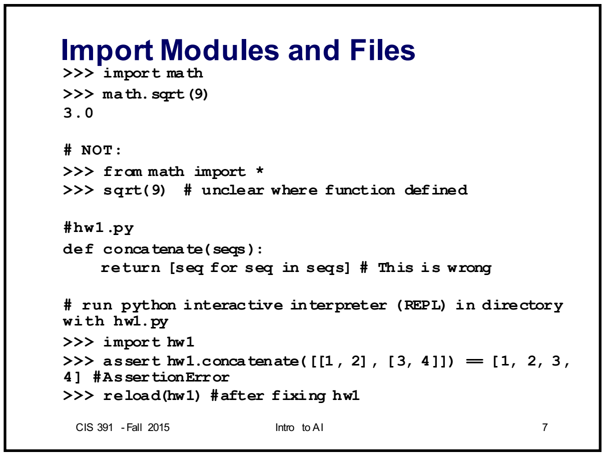
- **Function**

```
def reverse(data):
    for i in range(len(data)-1, -1, -1):
        yield data[i]
```

- **Generator Expression**

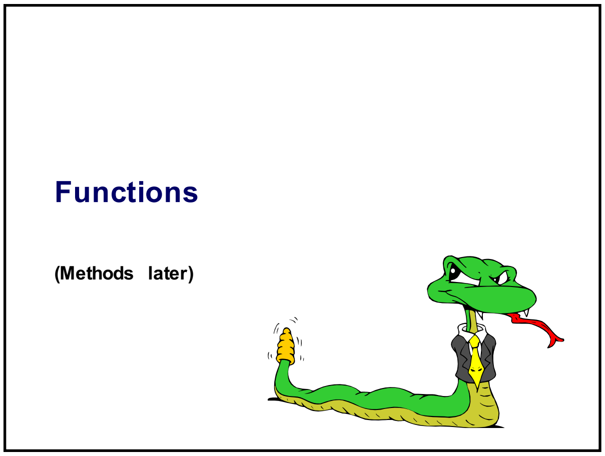
```
(data[i] for index in range(len(data)-1, -1, -1))

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec)) # dot product
260
```

- **Lazy Evaluation (on demand, values generated)**



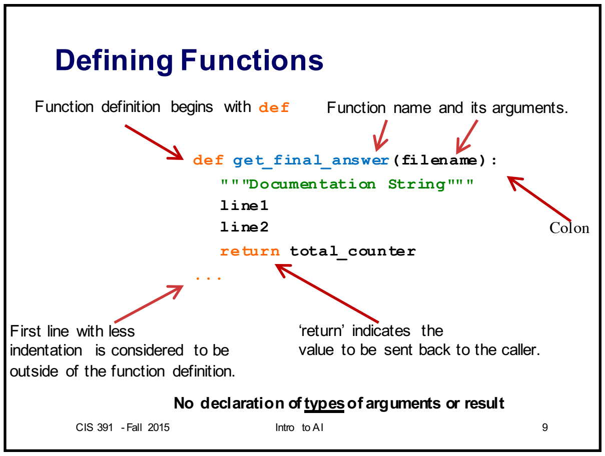
Import Modules and Files

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

# NOT:
>>> from math import *
>>> sqrt(9)  # unclear where function defined

#hw1.py
def concatenate(seqs):
    return [seq for seq in seqs] # This is wrong

# run python interactive interpreter (REPL) in directory
with hw1.py
>>> import hw1
>>> assert hw1.concatenate([[1, 2], [3, 4]]) == [1, 2, 3,
4] #AssertionError
>>> reload(hw1) #after fixing hw1
```



Functions

(Methods later)



Defining Functions

Function definition begins with def

Function name and its arguments.

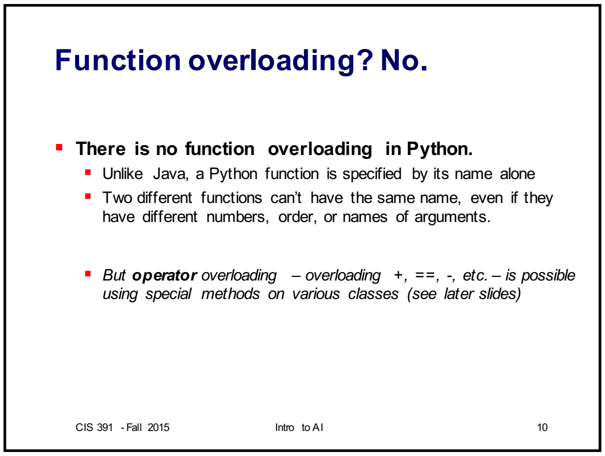
```
def get_final_answer(filename):
    """Documentation String"""
    line1
    line2
    return total_counter
...
```

Colon

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

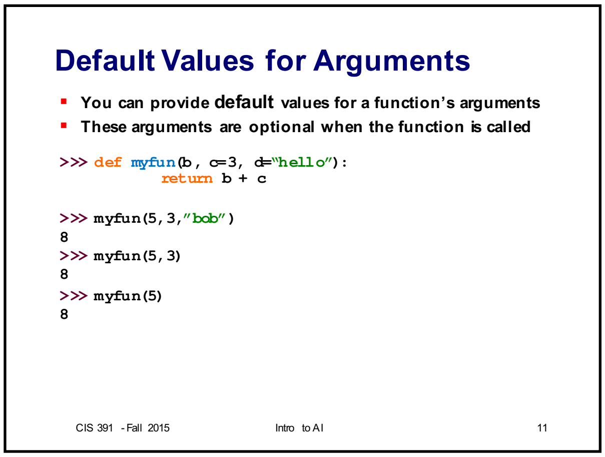
'return' indicates the value to be sent back to the caller.

No declaration of types of arguments or result



Function overloading? No.

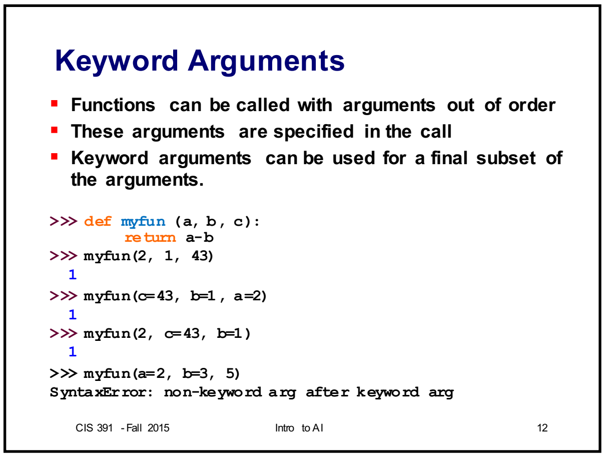
- **There is no function overloading in Python.**
  - Unlike Java, a Python function is specified by its name alone
  - Two different functions can't have the same name, even if they have different numbers, order, or names of arguments.
  - *But* ***operator*** *overloading – overloading +, ==, -, etc.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



Default Values for Arguments

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

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

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



Keyword Arguments

- **Functions can be called with arguments out of order**
- **These arguments are specified in the call**
- **Keyword arguments can be used for a final subset of the 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)
SyntaxError: non-keyword arg after keyword arg
```

*args

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

```
def print_everything(*args):
    # args is a tuple of arguments passed to the fn
    for count, thing in enumerate(args):
        print '{0}. {1}'.format(count, thing)
>>> lst = ['a', ’b', 'c’]
>>> print_everything(*lst)
0. a
1. b
2. c
>>> print_everything('a', ’b', 'c')
```

**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.iteritems(): #.items() is list
        print "%s = %s" % (key, value)

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

# Combining ideas
foo(arg1, *args, **kwargs) # one mandatory argument
```

Scope

- Function sees the most current value of variables

```
>>> 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 executed.
- The function uses the same default object each call

```
def fib(n, fibs={}):
    if n in fibs:
        return fibs[n]
    if n <= 1:
        fibs[n] = n
    else:
        fibs[n] = fib(n-1) + fib(n-2)
    return fibs[n]
```

First Class Functions

- Functions are “first-class citizens”
  - Pass functions as arguments to other functions,
  - returning functions as the values from other functions,
  - Assign functions to variables or store them in data structures
- Higher order functions: take functions as input

```
def compose(f, g, x):
   return f(g(x))
>>> compose(str, sum, [1,2,3])
 '6'
```

Higher Order Functions: Map, Filter

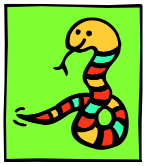
```
>>> [int(i) for i in ['1', '2']]
[1, 2]
>>> map(int, ['1', '2']) #equivalent to above
[1, 2]

def is_even(x):
   return x % 2 == 0
>>> [i for i in [1, 2, 3, 4, 5] if is_even(i)]
[2, 4]
>>> filter(is_even, [1, 2, 3, 4, 5]) [2, 4] #equivalent to above

>>> t1 = (0, 10)
>>> t2 = (100, 2)
>>> min([t1, t2], key=lambda x: x[1])
  (100, 2)
```

Sorted list of n-grams

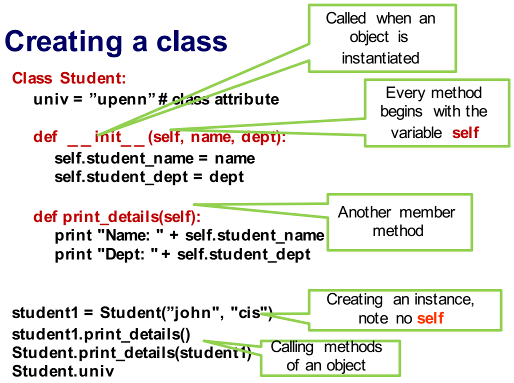
```
from operator import itemgetter
def calc_ngram(inputstring, nlen):
  ngram_list = [inputstring[x:x+nlen] for x in \
                xrange(len(inputstring)-nlen+1)]
  ngram_freq = {} # dict for storing results
  for n in ngram_list: # collect the distinct n-grams and
  count
    if n in ngram_freq:
      ngram_freq[n] += 1
    else:
      ngram_freq[n] = 1 # human counting numbers start at 1
  # set reverse = False to change order of sort
  (ascending/descending)
  return sorted(ngram_freq.iteritems(), \
               key=itemgetter(1), reverse=True)
```

http://times.jayliew.com/2010/05/20/a-simple-n-gram-calculator-pyngram/

Classes and Inheritance



Creating a class



```
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("john", "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.**

```
class ai_student(student):
```

- **Python has no 'extends' keyword like Java.**
- **Multiple inheritance is supported.**

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 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.**

```
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: `myOwnClass.methodName(a,b,c)`

__init__ constructors in subclasses:

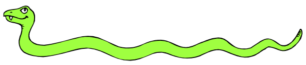
- **UNLIKE Java: To execute the ancestor's `__init__` method the ancestor's __init__ *must be called explicitly* (if the descendants __init__ is specified)**
- **The *first* line of the `__init__` method of a subclass will often be:**

```
parentClass.__init__(x, y)
super(self.__class__, self).__init__(x, y) #equivalent
```

Multiple Inheritance

```
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(C, self).foo() # Foo!
        super(C, self).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 ducky debugging'

```
class student:
    ...
     def __repr__(self):
       return "I'm named " + self.full_name + " - age: , ",
  self.age
    ...
>>> f = student("Bob Smith", 23)
>>> print f
I'm named Bob Smith - age: 23
```

Other "Magic" Methods

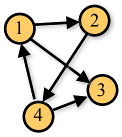
- Used to implement operator overloading
  - Most operators trigger a special method, dependent on class

`__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.

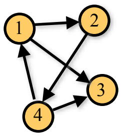
A directed graph class

```
>>> d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
>>> print d
1 -> 2
1 -> 3
2 -> 4
4 -> 3
4 -> 1
```



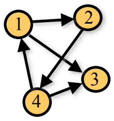
A directed graph class

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



`search` method returns a *generator* for the nodes that can be reached from a given node by following arrows "from tail to head"

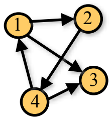
The DiGraph constructor



```
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) \
                        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=set()):
    # 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
```

Profiling, function level

- **Rudimentary**

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

- **Timeit (more precise)**

```
>>> 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

Profiling, script level

```
#to_time.py

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

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

if __name__ == '__main__':
    exp_fn()
```

Profiling, script level

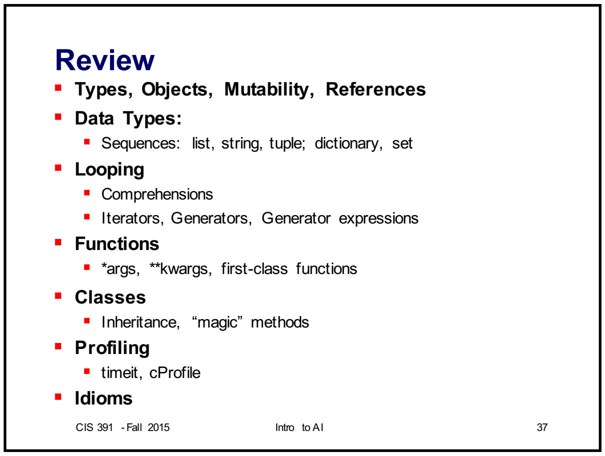
```
#python interactive interpreter (REPL)

$ python -m cProfile to_time.py
500004 function calls in 0.203 seconds
Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1       0.000    0.000    0.203    0.203   to_time.py:1(<module>)
500001  0.071    0.000    0.071    0.000   to_time.py:1(get_number)
1       0.133    0.133    0.203    0.203   to_time.py:5(exp_fn)
1       0.000    0.000    0.000    0.000   {method 'disable' of
'_lsprof.Profiler' objects}
```

- **If you need real speed (eg real time voice recognition), write C++**

Idioms

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

- **http://safehammad.com/downloads/python-idioms-2014-01-16.pdf**



Review

- **Types, Objects, Mutability, References**
- **Data Types:**
  - Sequences: list, string, tuple; dictionary, set
- **Looping**
  - Comprehensions
  - Iterators, Generators, Generator expressions
- **Functions**
  - *args, **kwargs, first-class functions
- **Classes**
  - Inheritance, “magic” methods
- **Profiling**
  - timeit, cProfile
- **Idioms**