# GIS 4653/5653: Spatial Programming and GIS

More Python:

Statements, Types, Functions, Modules, Classes

# Statement Syntax

# The if-elif-else statement

- Indentation and and colons are important
  - Parentheses and semicolons are optional
  - Do not need to use elif or else if you don't need it

```
for city in cities:
    name = city[0];
    clat = city[1];
    if ( clat> 36 ):
        regions[name] = "North";
    elif (clat < 35):
        regions[name] = "South";
    else:
        regions[name] = "Central";
```

```
for city in cities:
    name = city[0];
    clat = city[1];
    if clat> 36 :
        regions[name] = "North"
    elif clat < 35:
        regions[name] = "South"
    else:
        regions[name] = "Central"
```

# Level of indentation, Pass, +=

```
clon = city[2];
if ( clon> -97 ):
    regions[name] += "east";
elif (clon < -98):
    regions[name] += "west";
else:
    if regions[name] == "Central":
        pass
    else:
        regions[name] += "-central";
```

Level of indentation is important

Empty blocks are not allowed, so use “pass”

+= appends to the string

# Pass on nested indentations

- Better to rewrite earlier code to avoid nesting or the use of "pass"

```
clon = city[2];
if ( clon> -97 ):
    regions[name] += "east";
elif (clon < -98):
    regions[name] += "west";
elif regions[name] != "Central":
    regions[name] += "-central";
```

# and or not

- With x = 3 and y = 4

```
>>> if (x < 2 or y > 3):
        print "ha!"

ha!
```

- Also: (why?)

```
>>> if not (x < 2 and y > 3):
        print "ha!"

ha!
```

# Assignments

- Multiple assignment:

```
name, clat = city[0], city[1]
```

- Can assign the same value to multiple variables:

```
clat = clon = 0
```

# Ternary assignment

```
>>> exclaim = "ha" if (x< 2 and y > 3) else "hmm"
>>> print exclaim
hmm
```

- Can be a little cryptic

# Initialize variables before use

- Variables have to be initialized before they are used

```
>>> sum = sum + 3

Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    sum = sum + 3
TypeError: unsupported operand type(s) for +: 'builtin_function_or_method' and '
int'
>>> sum = 0
>>> sum = sum + 3
```

Bad (why?)

- Variables take "global" scope

```
>>> for i in range(5):
        sq = i*i

>>> sq
16
```

# Assignment-based idioms

- Swapping and sequence assignment

```
>>> x,y = range(3,5)
>>> print x,y
3 4
>>> y,x = x,y
>>> print x,y
4 3
```

- Slicing:

```
>>> arr = range(10)
>>> print arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> A,B = arr[:3], arr[3:]
>>> print A
[0, 1, 2]
>>> print B
[3, 4, 5, 6, 7, 8, 9]
```

# Caveat: Shared references

- What is happening here? Why?

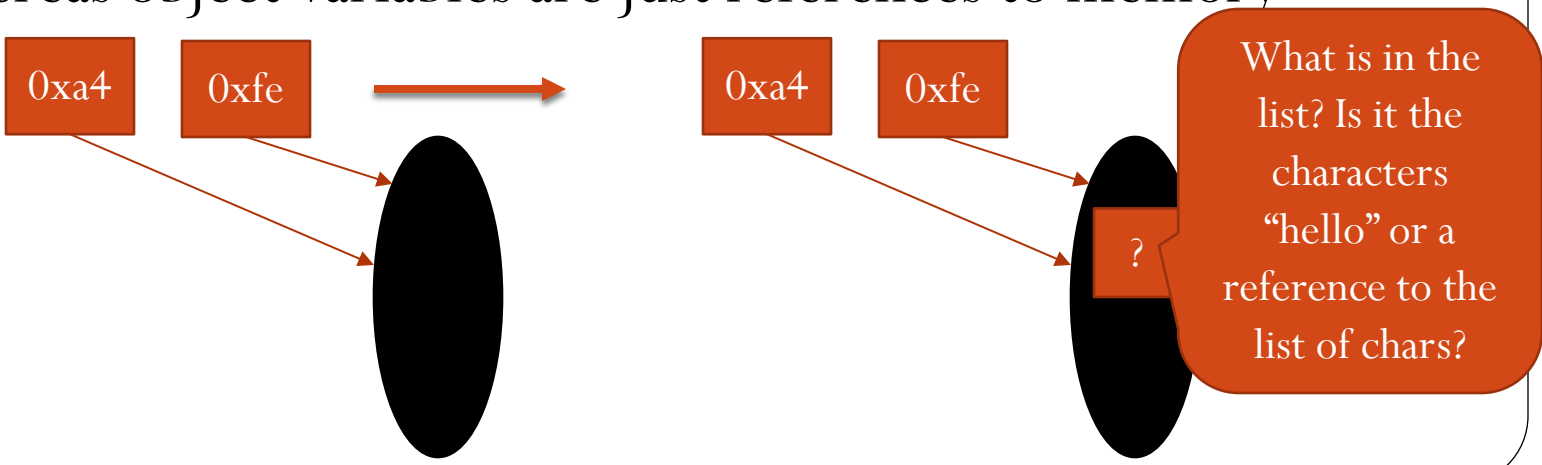
```
>>> A = B = []
>>> A.append("hello")
>>> print A
['hello']
>>> print B
['hello']
```

```
>>> A = B = 5
>>> A += 12
>>> print A
17
>>> print B
5
```

# Primitives vs. Objects

- Python distinguishes between primitives which are directly stored on the hardware and objects which are composed of primitives and are a software construct
- Primitives get their own piece of the hardware, and so:


- Whereas object variables are just references to memory



# Clones

- You can avoid this problem by using copies of the objects
  - However, shared references is usually what you want

```
>>> A = ["pizza", "soda"]
>>> B = A[:]
>>> A.append("breadsticks")
>>> A
['pizza', 'soda', 'breadsticks']
>>> B
['pizza', 'soda']
>>>
```

# Find out

- Is a string a primitive or is it an object?
  - How could you verify?

  - Try it out (use list and primitive examples earlier)
    - Here are some available operations on a String

# Strings are immutable

- Strings are objects, but they behave like primitives
  - Have no methods to alter them in-place

```
>>> A = B = "hello"
>>> A.capitalize()
'Hello'
>>> print A
hello
>>> A = A.capitalize()
>>> print A
Hello
>>> print B
hello
```

```
>>> A = B = "hello"
>>> A = A[:3]
>>> print A
hel
>>> print B
hello
```

# print without a newline

- Use a comma:

```
for key in regions.keys() :
    print key,
    print regions[key]
```

```
>>> for x in range(5):
        print x,

0 1 2 3 4
>>> for x in range(5):
        print x

0
1
2
3
4
```

# A while loop

```
>>> name = "Tulsa"
>>> while len(name) > 0:
        print name[0]
        name = name[1:]  # strip first letter

T
u
l
s
a
```

# Python Types

# Integers and floating point

- Basic numeric types in Python:
  - Integers
  - Floating point numbers
  - Base-8, base-16 numbers, etc.
- Normally, you can go with Python's default types but …

```
>>> x = 3
>>> y = 14
>>> y/x
4
>>> float(y)/x
4.666666666666667
```

# Floor division

- // is called a "floor" operation:

```
>>> x = 3
>>> y = 14
>>> y/x
4
>>> float(y)/x
4.666666666666667
>>> float(y)//x
4.0
>>>
```

- Works regardless of operand type (integer or float)

# Math operations

- Import the math library

```
>>> import math
>>> print(math.pow(math.e,2))
7.38905609893
>>> print(math.cos( math.sqrt(math.pi) ))
-0.200293541123
...
```

- Commonly, you may also see:

```
>>> from math import cos, sin
>>> x = 0.14
>>> print( sin(x)*sin(x) + cos(x)*cos(x) )
1.0
```

# Decimals

- Can avoid problems with rounding and precision by using decimal

```
>>> 0.1 + 0.1 - 0.2 + 0.1 - 0.3
-0.19999999999999998


>>> from decimal import Decimal
>>> a = Decimal('0.1')
>>> b = Decimal('0.2')
>>> c = Decimal('0.3')
>>> a + a - b + a - c
Decimal('-0.2')
```

# Fractions

```
>>> from fractions import Fraction
>>> Fraction(2,3) * Fraction(9,4)
Fraction(3, 2)
>>> Fraction(2,3) * Fraction('2.25')
Fraction(3, 2)
```

# Fractions from floats

```
>>> a = 1.0/6
>>> f = Fraction.from_float(a)
>>> f
Fraction(6004799503160661, 36028797018963968)
>>> f.limit_denominator(100)
Fraction(1, 6)
>>> f * Fraction(3,7)
Fraction(2573485501354569, 36028797018963968)
>>> f = f.limit_denominator(100)
>>> f * Fraction(3,7)
Fraction(1, 14)
```

# Sets have unique entries

```
>>> x = set( [34, 73, 86, 23, 34] )
>>> x
set([73, 34, 86, 23])
>>> 23 in x
True
>>> 43 in x
False
>>> y = set( [34, 74, 85, 23] )
>>> y - x
set([74, 85])
>>> y + x

Traceback (most recent call last):
  File "<pyshell#123>", line 1, in <mo
    y + x
TypeError: unsupported operand type(s)
>>> y | x
set([34, 73, 74, 85, 86, 23])
>>> y & x
set([34, 23])
```

```
>>> x.add(35)
>>> x
set([73, 34, 35, 86, 23])
>>> x.remove(34)
>>> x
set([73, 35, 86, 23])
>>> for item in x:
        print item*4


292
140
344
92
```

# Set comprehension

```
>>> for item in x:
        print item*4,

292 140 344 92
>>> {item*4 for item in x}
set([344, 140, 292, 92])
>>>
```

# String is an object and has methods

```
>>> S = ' hello  ';  S.rstrip()
' hello'
>>> S = ' hello  ';  S.lstrip()
'hello  '
>>> S = ' hello  ';  S.strip()
'hello'
```

```
>>> S = ' hello  ';  S.strip().endswith('llo')
True
>>> S = ' hello  ';  S.strip().upper()
'HELLO'
```

# Raw strings

- Can escape a string with backslashes
  - Or use the raw string format
  - Can use either single quote or double quote

```
>>> S = 'h\te\tl'
>>> print(S)
h       e       l
>>> S=r'h\te\tl'
>>> print(S)
h\te\tl
>>> S='h\\te\\tl'
>>> print(S)
h\te\tl
```

# Multi-line strings

```
>>> longline="""
There are many, many
ways to skin a cat,
but that is a really
a very gross analogy.
Why would you want to
skin a cat anyway?
"""
>>> longline
'\nThere are many, many\nways to skin a cat,\nbut
that is a really\na very gross analogy.\nWhy would
 you want to\nskin a cat anyway?\n'
```

- Can use multi-line strings as a way to comment out multiple lines of code

# Lists

```
>>> toppings = ['pepperoni', 'cheese', 'sausage']
>>> toppings
['pepperoni', 'cheese', 'sausage']
>>> toppings[1:]
['cheese', 'sausage']
>>> toppings[:2]
['pepperoni', 'cheese']
>>> toppings[1:1]
[]
>>> toppings[1:2]
['cheese']
>>> toppings.append('brocolli')
>>> toppings
['pepperoni', 'cheese', 'sausage', 'brocolli']
>>> toppings.sort()
>>> toppings
['brocolli', 'cheese', 'pepperoni', 'sausage']
```

# Sorting keys, order

```
>>> toppings = ['  sausage', 'cheese', ' brocolli']
>>> toppings.sort(reverse=True, key=str.lstrip)
>>> toppings
['  sausage', 'cheese', ' brocolli']
>>> toppings.sort(reverse=True)
>>> toppings
['cheese', ' brocolli', '  sausage']
```

# Custom comparator

- Just define a function and use it to sort

```
A = ["cheese", "  brocolli", "    sausage", "       calamari" ]
def thirdletter(s):
    return s.strip()[2]
A.sort(key=thirdletter)
print A
```

```
>>>
['cheese', '      calamari', '  brocolli', '    sausage']
>>>
```

# A dictionary is an associative array

```
>>> prices = { 'pizza' : 14.95,  'soda' : 1.99,  'breadsticks' : 2.49 }
>>> prices['pizza']
14.95
>>> prices['soda']
1.99
>>> len(prices)
3
>>> 'coke' in prices
False
>>> prices['pizza'] = 12.95
>>> prices
{'soda': 1.99, 'breadsticks': 2.49, 'pizza': 12.95}
>>> list( prices.values() )
[1.99, 2.49, 12.95]
>>> list( prices.keys() )
['soda', 'breadsticks', 'pizza']
>>> prices.pop('soda')
1.99
>>> prices
{'breadsticks': 2.49, 'pizza': 12.95}
```

# Dynamically building dictionary

```
>>> prices = {}
>>> prices['pizza'] = 12.95
>>> prices['soda'] = 1.99
>>> prices['breadsticks'] = 2.49
>>> prices
{'soda': 1.99, 'breadsticks': 2.49, 'pizza': 12.95}

>>> for item in prices:
        print item, prices[item]

soda 1.99
breadsticks 2.49
pizza 12.95
```

# Tuples work like immutable lists

- Use tuples if you want data integrity
  - Use lists otherwise

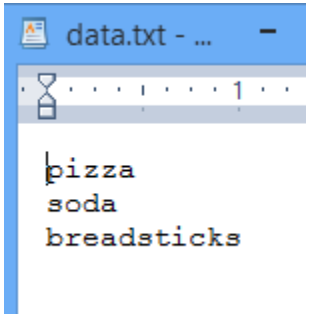
```
>>> items = ('pizza', 'soda', 34, 21)
>>> itemsList = ['pizza', 'soda', 34, 21]
>>> itemsList.append('monty')
>>> print itemsList
['pizza', 'soda', 34, 21, 'monty']
>>> items.append('monty')

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    items.append('monty')
AttributeError: 'tuple' object has no attribute 'append'
>>> items[1:]
('soda', 34, 21)
>>>
```

# Writing Files

- Files are first-class types in Python

```
file = open("data.txt", "w")
file.write("pizza\n")
file.write("soda\n")
file.write("breadsticks\n")
file.close()
```




- Unlike print(), write() does not automatically add a new line

# Reading files

- Can read an entire file as a string

```
>>> data = open("data.txt", "r").read()
>>> data
'pizza\nsoda\nbreadsticks\n'
```

- Can get the lines by splitting:

```
>>> data.split('\n')
['pizza', 'soda', 'breadsticks', '']
```

# Reading line-by-line

- Can iterate through a file line-by-line

```
> for line in open("data.txt","r"):
      print(line.upper())
```

- Can manually iterate through a file using next()

```
>>> f = open("hometowns.py","r")
>>> line = next(f)
>>> line
'from mpl_toolkits.basemap import Basemap\n'
>>> line = next(f)
>>> line
'import numpy as np\n'
```

# Iterating through lists, tuples, maps

- Using a for-loop:

```
>>> toppings
['  sausage', ' brocolli', 'cheese']
>>> for topping in toppings:
        print topping.upper(),

  SAUSAGE  BROCOLLI CHEESE
```

```
>>> prices
{'soda': 1.99, 'breadsticks': 2.49, 'pizza': 12.95}
>>> for item in prices.keys():
        print item.upper(),

SODA BREADSTICKS PIZZA
```

- For manual iteration, you need to first create an iter

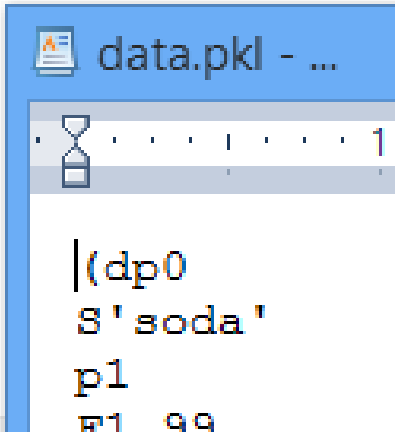
```
>>> it = iter(toppings)
>>> it.next()
'  sausage'
>>> it.next()
' brocolli'
```

  - File objects act as their own iterator, so this step was not needed

# Saving objects

- To save objects, use the pickle module

```
>>>
>>>
>>>
>>> file = open("data.pkl", "w")
>>> import pickle
>>> pickle.dump(prices, file)
>>> file.close()
>>>
```



# Reading pickled objects

```
>>> file = open("data.pkl", "r")
>>> obj = pickle.load(file)
>>> obj
{'pizza': 12.95, 'breadsticks': 2.49, 'soda': 1.99}
>>>
```

# Functions

# Why functions?

- Functions allow you to:
  - Create reusable code
  - Break down complex logic into easily understood pieces

# Example function

- The "def" keyword creates a function object

```
def degrees_to_radians(deg):
    """
    Converts the input deg in degrees into radians and returns the result
    """
    radians = deg * math.pi / 180;
    return (radians)
```

  - Note the Python documentation
- The return statement ends the function call and sends result back to caller
  - A function without a return statement returns "None"

# def is evaluated at runtime

- Can embed "def" inside an if statement or inside a function

```
if n > 10:
    def limit():
        return math.pow(2,10)
else
    def limit():
        return math.pow(2,n)
N = limit()
```

# Aliasing functions

- Simply assign to a new name
  - And call function using that new name

```
def great_circle_distance(lat1, lon1, lat2, lon2):
    d2r = degrees_to_radians
    lat1 = d2r(lat1);
    lat2 = d2r(lat2);
```

# Function arguments are typeless

- The arguments get their types at runtime
  - As if code is reinterpreted by Python compiler each time

```
>>> def repeat(a,n):
        return (a*n)

>>> repeat('the', 3)
'thethethe'
>>> repeat(4, 3)
12
```

# Variable scope is lexical

```
g = 100
def scopefunc():
    loc = g - 3;
    print("loc=" + str(loc))
    def scope2(loc):
        ll = loc - 3;
        print("ll=" + str(ll))
    scope2(loc)
scopefunc()
```

# Four ways of calling a function

- Given this function:

```
def great_circle_distance(lat1, lon1, lat2, lon2):
```

- Can call it by position or by argument name
  - Can use sequence to represent positions
  - Can use dictionary to represent argument names

```
>>> great_circle_distance(35, -97, 36, -98)
180.0441401750096
>>> great_circle_distance(lat1=35, lat2=36, lon1=-97, lon2=-98)
180.0441401750096
>>> seq=[35, -97, 36, -98]
>>> great_circle_distance(*seq)
180.0441401750096
>>> dict={ "lat1":35, "lat2":36, "lon1":-97, "lon2":-98 }
>>> great_circle_distance(**dict)
180.0441401750096
```

# Modules

# Using modules

- Fetch a module as a whole using import
- Fetch particular names from a module using from

```
import math;  # needed to get pi
from math import cos, sin;
```

# Module search path

- Modules are searched for in this order:
  - The home directory of the program
    - If you call import from a.py, it looks in the same directory as a.py
    - In interactive mode, it is the directory in which IDLE was started
  - The environment variable PYTHONPATH
  - Python install directories
  - A search path specified in a .pth file by person installing software

# Imports happen only once

- Imports happen only once
  - Use reload in interactive sessions

```
>>> reload(haversine)
```

  - Can also use this to change some code and re-read it

# Package imports

- A statement such as:

```
>>> import   sp.ch00.haversine
```

- Imports the file haversine.py from the directory sp\ch00
  - Relative to somewhere on the module search path
  - The directories themselves have to a file named __init__.py
- Relative paths (. and ..) do not work in 2.7 (only 3.0)

# Test methods

- To write code that runs only when a module is run as a main program, but not when it is imported, you can do:

```
if __name__ == '__main__' :
    datadir = "../data/40027_Cleveland_County/"
    write_named_shapes( datadir + "/tl_2009_40027_areawater",
                        datadir + "/ch03", "cleveland_county_waterbodies" )
```

- This is a convenient way to test functions

# Classes and objects

# Object = way of grouping fields

- Often convenient to group data values together
  - Define such groups in a class
  - Provide an initialization function (called a constructor)

```
class Point:
    lat, lon = 0, 0
    def __init__(self, latitude, longitude):
        self.lat, self.lon = latitude, longitude
```

- To use this object, you have to first create it:

```
>>> a = haversine.Point(35,-97)
>>> b = haversine.Point(38,-96)
```

# Object = fields + methods

- Can define methods on an object which operates on fields
  - Think of methods as being called on an object:

```
class Point:
    lat, lon = 0, 0
    def __init__(self, latitude, longitude):
        self.lat, self.lon = latitude, longitude
    def getLat(self):
        return self.lat
    def getLon(self):
        return self.lon
    def getDistance(self,other):
        return great_circle_distance(self.lat,self.lon,
                                     other.lat, other.lon)
```

```
>>> a = haversine.Point(35,-97)
>>> b = haversine.Point(38,-96)
>>> a.getDistance(b)
433.64690650848286
...
```

# Testing objects

- Often useful to test objects by putting usage code in module
  - But would like to prevent test code when running module
  - Use the __name__ construct as follows:

```
    def getLon(self):
        return self.lon
    def getDistance(self,other):
        return great_circle_dist



if __name__ == '__main__':
    a = Point(35,-97)
    b = Point(38,-96)
    print(a.getDistance(b))
```

# Inheritance

- Can add extra data to an object by inheriting it:

```
class Point3D(Point):
    ht = 0
    def __init__(self, latitude, longitude, height=0):
        Point.__init__(self,latitude, longitude)
        ht = height
    def getHeight(self):
        return self.ht
```

- Point3D has all the fields and methods of Point
  - And has an extra "ht" field and an extra "getHeight()" method

# Over-riding methods

- Can override methods to take advantage of new information

```
class Point3D(Point):
    ht = 0
    def __init__(self, latitude, longitude, height=0):
        Point.__init__(self,latitude, longitude)
        ht = height
    def getHeight(self):
        return self.ht
    def getDistance(self,other):
        dist2d = Point.getDistance(self,other)
        distht = (self.ht - other.ht)
        return math.sqrt(dist2d*dist2d + distht*distht)

if __name__ == '__main__':
    a = Point3D(35,-97,0.4)
    b = Point3D(38,-96,0.1)
    print(a.getDistance(b))
```

# Operator overloading

- The __init__ method is an example of operator overloading
- Can optionally implement the following methods:
  - __add__
    - What happens when someone does a + b?
    - Also: __sub__, __gt__, __lt__, __le__, __ge__
  - __str__
    - What happens when someone does print(a)?
  - __del__
    - What happens when object is garbage collected (reclaimed)

# Exceptions

# What are exceptions?

- Exceptions are errors that make the rest of the code pointless
  - Think of it as a "go to" that goes back to whoever called the code

```
class InvalidLocationError(Exception):
    msg = ""
    def __init__(self,message):
        self.msg = message
    def __str__(self):
        return self.msg

class Point:
    lat, lon = 0, 0
    def __init__(self, latitude, longitude):
        self.lat, self.lon = latitude, longitude
        if (self.lat < -90 or self.lat > 90):
            raise InvalidLocationError("Latitude out of range: " + str(self.lat)
        if (self.lon < -180 or self.lon > 180):
            raise InvalidLocationError("Longitude out of range: " + str(self.lo
    def getLat(self):
```

# Catching exceptions

- Put the code that could throw the exception in a "try"
- Catch it in an "except"

```
try:
    c = Point3D(35,-97,0.4)
    d = Point3D(38,-970,0.1)
    print(c.getDistance(d))
except InvalidLocationError as err:
    print(err)
```

```
Longitude out of range: -970
```

# Practice: Data Analysis of Text Files

- We will work with a dataset of major cities from: http://www.mongabay.com/cities_pop_01.htm
  - Select the top 20 or so cities from the list
  - Copy-and-paste into a .txt document
  - The data should come in as separate lines with the fields on each line separated by tabs
- Goal: Find the top 5 most dense cities
  - Density = population / area
- Also print out the city with the minimum density

# Solution: practice

```
import math

class City:
    name, area, pop, density = "", 0, 0, 0
    def __str__(self):
        return "{:s} area={:d} pop={:d} density={:.1f}".format(self.name,self

# read in data
chunk = open("cities.txt", 'r').read()
lines = chunk.split('\n')
cities = []
for myline in lines:
    fields = myline.split('\t')
    if len(fields) > 3:
        city = City()
        city.area = int( fields[3].replace(',','') )
        city.pop = int( fields[2].replace(',','') )
        city.name = fields[0]
        cities.append(city)

# compute the density
for city in cities:
    area = city.area
    pop = city.pop
    city.density = pop / float(area)
```

# Solution: practice

```
# compute the density
for city in cities:
    area = city.area
    pop = city.pop
    city.density = pop / float(area)

# top 5
def getdensity(city):
    return city.density
cities.sort(key=getdensity,reverse=True)
for i in range(5):
    print(cities[i])

# city with minimum density
# do not use sorted list ...
mindensity = 1000000
best = ""
for city in cities:
    if city.density < mindensity:
        mindensity = city.density
        best = city.name
print "mindensity={:f} in {:s}".format(mindensity, best)
```

# Homework

- Read iow_firehydrants.txt
  - Contains the locations of fire hydrants in Isle of Wight, North Carolina
  - For each hydrant, compute the distance of the hydrant nearest to it (this will be the backup hydrant if this hydrant were not available)
  - Print 5 most indispensable hydrants (there is nothing else close by)