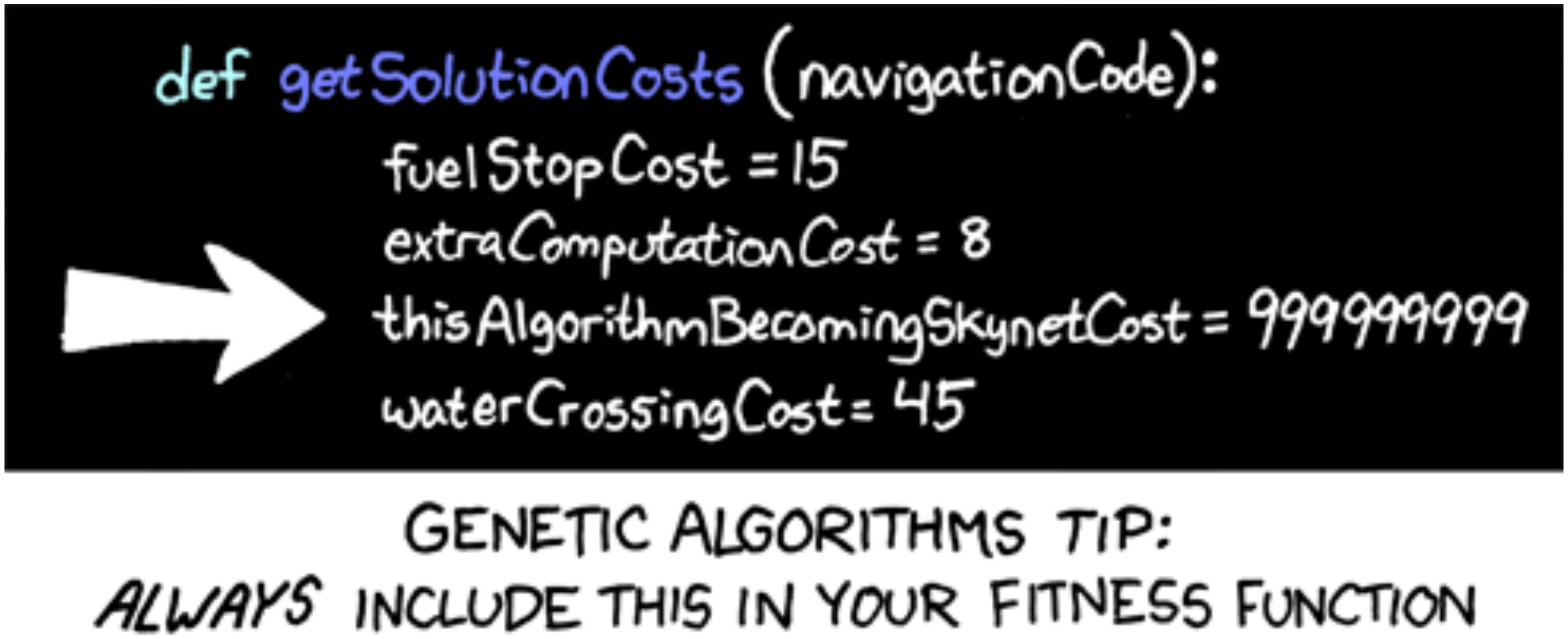
CS3101.3

Lecture 2



```
def getSolutionCosts(navigationCode):
        fuelStopCost = 15
        extraComputationCost = 8
        thisAlgorithmBecomingSkynetCost = 999999999
        waterCrossingCost = 45
```

Source: http://xkcd.com/534/

Agenda

- Dictionaries, sets
- Sequences, iterables, slicing
- List comprehensions
- Sorting, custom comparators
- Functions, modules, imports
- Command line arguments
- File I/O
- Homework 2

Questions? Homework, general?

Today’s theme: Files and Functions with the Simpsons

- By the end of this lecture you will be able to
- Produce an HTML webpage from this CSV file

```
Character,Meal,Ate,Quantity,Comment
Barney Gumble,Breakfast,Duff Beer,1,I could’ve gone to Harvard
Duffman,Breakfast,Power bar,4,Duffman - can't breathe!
Duffman,Lunch,Protein shake,5,Duffman - has the power!
Homer Simpson,Snack,Donut - Jelly,1,Mmm Donut
Homer Simpson,Snack,Donut - Cream,1,Mmm Donut
Homer Simpson,Snack,Donut - Strawberry,1,Mmm Donut
Homer Simpson,Dinner,Brocolli,2,So long cruel world
Lisa Simpson,Breakfast,Oranges,1,Satisfying
Lisa Simpson,Lunch,Kale,2,Einstein's favorite
Lisa Simpson,Dinner,Tofu,4,Animals are cute!
Klang,Snack,Humans,40, *how*to*cook*for*forty?*humans*
Montgomery Burns,Snack, Life Extension Elixer,1,Excellent…
```

Dictionaries (cont'd)

- Review:
  - Dictionaries are Python's built in *mapping type*
  - Mutable, not ordered
  - Keys in a dictionary must be *hashable*
  - Values are arbitrary objects, may be of different types
  - Items in a dictionary are a key/value pair

Specifying Dictionaries

- Use a series of pairs of expressions, separated by commas within braces
- i.e. {‘x’: 42, ‘y’: 3.14, 26: ‘z’} will create a dictionary with 3 items, mapping ‘x’ to 42, ‘y’ to 3.14, 26 to ‘z’.
- Alternatively, use the dict() function
  - Less concise, more readable
  - dict(x=42, y=3.14)
- Dictionaries do not allow duplicate keys
  - If a key appears more than once, only one of the items with that key is kept (usually the last one specified)

Dictionaries (cont'd)

- Suppose x = {'a':1, 'b':2, 'c':3}
  - To access a value: x['a']
  - To assign values: x['a'] = 10
  - You can also specify a dictionary, one item at a time in this way
- Common methods
  - .keys(), .values(), .items(), .setdefault(...), .pop(...)
  - Sequences returned in arbitrary order
- Dictionaries are containers, so functions like len() will work
  - To see if a key is in a dictionary, use in keyword
  - E.g. 'a' in x will return True, 'd' in x will return False.
  - Attempting to access a key in a dictionary when it doesn't exist will result in an error

More Examples

```
>>> x['dad'] = 'homer’
>>> x['mom']= 'marge’
>>> x['son'] = 'bart’
>>> x['daughter'] = ['lisa', 'maggie']
>>> print 'Simpson family:', x
Simpson family: {'dad': 'homer', 'daughter': ['lisa', 'maggie'], 'son': 'bart',
    'mom': 'marge’}
>>> 'dog' in x
False
>>> print 'Family members: ', x.values()
Family members: ['homer', ['lisa', 'maggie'], 'bart', 'marge']
 >>> x.items()
[('dad', 'homer'), ('daughter', ['lisa', 'maggie']),, ('son', 'bart'), ('mom',
    'marge')]
```

Python has built in sets

```
simpsons = set(["homer", "marge", "bart", "lisa"])
philosophers = set(["aristotle", "sophocles", "homer"])


print simpsons.intersection(philosophers)
set(['homer'])

print simpsons.union(philosophers)
set(['homer', 'marge', 'aristotle', 'lisa’, 'sophocles', 'bart'])

print set(['homer']).issubset(philosophers)
True
```

Sets

- Python has two built-in types: set and frozenset
  - Instances of type set are mutable and not hashable
  - Instances of type frozenset are immutable and hashable
  - Therefore, you may have a set of frozensets, but not a set of sets
- Methods (given sets S, T):
  - Non-mutating: S.intersection(T), S.issubset(T), S.union (T)
  - Mutating: S.add(x), S.clear(), S.discard(x)
  - Standard operators are overloaded for the non-mutating methods: e.g., Set1 & Set2 overloads to intersection_update

Sets cont’d

- To create a set, call the built-in type set() with no argument (i.e. to create an empty set)
  - You can also include an argument that is an iterable, and the unique items of the iterable will be added into the set
- Exercise: count the number of unique items in the list L generated using the following:
- import random
- L = [random.randint(1, 50) for x in range (100)]

Sequences

- An ordered collection of items
- Indexed by nonnegative integers
- Lists, Strings, Tuples
- Librarians and extensions provide other kinds
- You can create your own (later)
  - E.g., the sequences of prime numbers
- Sequences can be sliced (which means extracting a section)
- Not Pythonic in style – but powerful

Iterables

- Python concept which generalizes idea of sequences
- All sequences are iterables
- Beware of bounded and unbounded iterables
  - Sequences are bounded
  - Iterables don't have to be
    - E.g., iterating over a generator computing the next digit of PI
  - Beware when using unbounded iterables: it could result in an infinite loop or exhaust your memory

Manipulating Sequences

- Concatenation
  - Most commonly, concatenate sequences of the same type using + operator
    - Same type refers to the sequences (i.e. lists can only be concatenated with other lists, you can't concatenate a list and a tuple)
    - The items in the sequences can usually vary in type
    - Less commonly, use the * operator
- Membership testing
  - Use of in keyword
  - E.g. 'x' in 'xyz' for strings
  - Returns True / False
- Example ['a', 'b', 'c'] + ['d', 'e', 'f']

Slicing

- Extract subsequences from sequences by slicing
- Given sequence s, syntax is s[x:y] where x,y are integral indices– x is the inclusive lower bound – y is the exclusive upper bound
- If x<y, then s[x:y] will be empty
- If y>len(s), s[x:y] will returns [x:len(s)]
- If lower bound is omitted, then it is by default 0
- If upper bound is omitted, then it is len(s) by default

Slicing (cont'd)

- Also possible: s[x:y:n] where n is the stride
- n is the positional difference between successive items
- s[x:y] is the same as s[x:y:1]
- s[x:y:2] returns every second element of s from [x, y]
- s[::3] returns every third element in all of s
- x,y,n can also be negative
- s[y:x:-1] will return items from [y, x] in reverse (assuming y > x)
- s[-2] will return 2nd last item

Slicing (cont'd)

- It is also possible to assign to slices
- Assigning to a slicing s[x:y:n] is equivalent to assigning to the indices of s specified by the slicing x:y:n
- Examples:

```
>>> s = range(10)
>>> s[::-1] # Reverses the entire list
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
>>> s = range(10)
>>> s[1::2] = s[::2]
>>> s
[0, 0, 2, 2, 4, 4, 6, 6, 8, 8]
>>> s = range(10)
>>> s[5:0:-1]
 [5, 4, 3, 2, 1]
```

Rewriting set intersections with a list comprehension

```
def intersection(groupA, groupB):
    result = []
    for obj in groupA:
        if obj in groupB:
            result.append(obj)
    return result
```

```
def intersection(groupA, groupB):
    return [obj for obj in groupA if obj in groupB]
```

List Comprehensions

- It is common to use a for loop to go through an iterable and build a new list by appending the result of an expression
- Python list comprehensions allows you to do this quickly
- General syntax: [expression for target in iterable clauses]
- Underlying implementation is efficient

- The following are equivalent:
- result = []
- for x in sequence:
  - result.append(x + 1)
- result=[x+1 for x in sequence]
- Never sacrifice readability for elegance
- General rule: one or two lines, fine – more, bad idea

Examples

- >>> a = ['dad', 'mom', 'son', 'daughter', 'daughter']
- >>> b = ['homer', 'marge', 'bart', 'lisa', 'maggie']
- >>> [a[i] + ': ' + b[i] for i in range(len(a))]
- ['dad: homer', 'mom: marge', 'son: bart', 'daughter: lisa', 'daughter: maggie']
- >>> [x + y for x in range(5) for y in range(5, 10)]
- [5, 6, 7, 8, 9, 6, 7, 8, 9, 10, 7, 8, 9, 10, 11, 8, 9, 10, 11, 12, 9, 10, 11, 12, 13]
- >>> [x for x in range(10) if x**2 in range(50, 100)]
- [8, 9]

Sorting

- sorted() vs. list.sort()
  - For iterables in general, sorted() can be used
  - For lists specifically, there is the .sort() method
- Sorted()
  - Returns a new sorted list from any iterable
- List.sort()
  - Sorts a list in place
  - Is extremely fast: uses “timsort”, a “non-recursive adaptive stable natural mergesort/binary insertion sort hybrid”
  - Read the PEP if curious – pretty impressive

Sorting (cont'd)

- Standard comparisons understand numbers, strings, etc.
- What if you have to compare something different?
- Built in function cmp(x, y) is the default comparator
  - Returns-1 if x < y
  - Returns 0 if x == y
  - Returns 1 if x > y
- You can create custom comparators by defining your own function
  - Function compares two objects and returns -1, 0, or 1 depending on whether the first object is to be considered less than, equal to, or greater than the second object
  - Sorted list will always place "lesser" objects before "greater" ones
  - More on functions later

Sorting (cont'd)

- For list L, sort syntax is:
  - L.sort(cmp=cmp, key=None, reverse=False)
  - All arguments optional (default ones listed above)
  - cmp is the comparator function
  - If key is specified (to something other than None), items in the list will be compared to the key(item) instead of the item itself
- For sorted, syntax is:
  - sorted(cmp, key, reverse)

What if you had to sort a dictionary?

- Dictionaries cannot be directly sorted – a mapping has no order
  - You need to select an index of either keys or values

```
simpsons = {'bart' : 7, 'marge' : '41', 'homer' : '42', 'lisa' : 6}

print sorted(simpsons) #sorts by keys
['bart', 'homer', 'lisa', 'marge']

from operator import itemgetter
print sorted(simpsons.iteritems(), key=itemgetter(1), reverse=True)
[('homer', '42'), ('marge', '41'), ('bart', 7), ('lisa', 6)]

#if you just want the keys (sorted by value)
x = sorted(simpsons.iteritems(), key=itemgetter(1), reverse=True)
print [obj[0] for obj in x]
['homer', 'marge', 'bart', 'lisa']
```

None

- Built in data type to denote a null object
- No attributes or methods
- Use as a placeholder when you need a reference, but don’t care about the object
- Functions return None unless they have specific return statements meant to return other values

Functions

Python is all about the libraries

- Python’s libraries are HUGE
- Check first before reinventing the wheel

Libraries Preview
http://docs.python.org/library/

- Out of the box:
  - Numeric and mathematical modules
  - Files and directory handling
  - Data persistence
  - Data compression and archiving
  - File formats (csv, etc)
  - Cryptographic services
  - Operating system hooks
  - Interprocess communication and threading
  - Internet data handling
  - Structured markup handling (html, xml, etc)
  - Internet protocols (you name it)
  - Multimedia services (presentation, manipulation)
  - Internationalization
  - GUIs
  - Debugging, profiling
  - Windows, Mac, *nix, Sun specific services

Functions

- Most statements in a typical Python program are grouped together into functions or classes
- Request to execute a function is a function call
- When you call a function, you can pass in arguments
- A Python function always returns a value
  - If nothing is explicitly specified to be returned, None will be returned
- Functions are also objects!
  - May be passed as arguments to other functions
  - May be assigned dynamically at runtime
  - May return other functions
  - May even be keys in a dictionary or items in a sequence!

The def Statement

- The most common way to define a function
- Syntax:
  - def function-name(parameters):
    - statement(s)
- function-name is an identifier; it is a variable name that gets bound to the function object when def executes
- Each call to a function supplies *arguments corresponding to the parameters listed in the function definition*
- Example:
  - def sum_of_squares(x, y):
    - return x**2 + y**2
- Discussion, difference between x = sum_of_squares(1, 2) and x = sum_of_squares

Parameters

- Parameters are local variables of the function
- Pass by reference vs. pass by value
  - Passing mutable objects is done by reference
  - Passing immutable objects is done by value
- Supports named keyword and default arguments
  - def f(x, y=[]):
    - y.append(x)
    - return y
  - Beware: default value gets computed when def is executed, not when function is called
- Supports optional, arbitrary number of argument
  - def sum_args(*numbers):
    - return sum(numbers)
  - sum_args([1,2,3])

Functions are polymorphic

```
def multiply(x, y):
    return x * y

print multiply (2,4)
8

print multiply (.5,2)
1.0

print multiply("foo", 3)
foofoofoo
```

```
def intersection(groupA, groupB):
    result = []
    for obj in groupA:
        if obj in groupB:
            result.append(obj)
    return result

print intersection([1,2,3], (1,3,4))
[1, 3]


foo = {"homer" : "donuts", "lisa" : "kale"}
bar = ["homer", "kale"]
print intersection(foo, bar)
['homer']
```

Notice the list and tuples?

Docstrings

- An attribute of functions
- Documentation string (docstring)
- If the first statement in a function is a string literal, that literal becomes the docstring
- Usually docstrings span multiple physical lines, so triple-quotes are used
- For function f, docstring may be accessed or reassigned at runtime! using f.__doc__
- def foo(x):
  - ‘’’Hello docstring world’’’

The return Statement

- Allowed only inside a function body and can optionally be followed by an expression
- None is returned at the end of a function if return is not specified or no expression is specified after return
- Point on style: you should never write a return without an expression

Recursion

```
# returns the factorial of x
def fact(x): return (1 if x == 0 else x * fact(x-1))
```

```
# equivalently...
def fact2(x):
    if x == 0:
        return 1
    else:
        return x * fact2(x-1)
```

```
# an iterative solution
def fact3(x):
    result = 1
    while (x > 1):
        result = result * x
        x = x - 1
    return result
```

Discussion: Question from last week: is vs. ==. Is (is) identity function, == (is) equality

def is a statement

Legal in Python

```
import random

if random.randint(0,1):
    def func(x):
        return x + 1
else:
    def func(x):
        return x -1

print func(1)

0
2
…
```

Why is this legal?

- Python statements are executable – we have runtime, but not compile time in the C sense
- Functions are objects like everything else in Python and can be created on the fly

Modules and imports

Modules

- From this point on, we’re going to need to use more and more modules
- A typical Python program consists of several source files, each corresponding to a module
- Simple modules are normally independent of one another for ease of reuse
- Packages are (sometimes huge!) collections of modules

Imports

- Any Python source file can be used as a module by executing an import statement
- Suppose you wish to import a source file named MyModule.py, syntax is:
  - import MyModule as Alias
  - The Alias is optional
- Suppose MyModule.py contains a function called f()
  - To access f(), simply execute MyModule.f()
  - If Alias was specified, execute Alias.f()
- Can also say "from MyModule import f"
- Stylistically, imports in a script file are usually placed at the start
- Modules will be discussed in greater detail when we talk about libraries
- Common question: place them in the same directory as your script, next python will look in sys.path
- This is why we need if __name__ == "__main__"

Command-Line Arguments

- Arguments passed in through the command line (the console)
- $ python script.py arg1 arg2 …
- Note the separation by spaces
- Encapsulation by quotes is usually allowed
- The sys module is required to access them
  - import sys
  - Command-line arguments are stored in sys.argv, which is just a list
  - Recall argv, argc in C---no need for argc here, that is simply len(sys.argv)

Common pattern

```
import sys
def main():
    if len(sys.argv) != 2:
        print 'Invalid Input’
        print 'Usage: python myscript.py arg1'
    else:
        something = sys.argv[1]
        …
```

Example

```
# multiply.py
import sys
product = 1
for arg in sys.argv:
    print arg, type(arg) # all arguments stored as strings
for i in range(1, len(sys.argv)): # first arg is the script name!
        product *= int(sys.argv[i])
print product

$ python multiply.py 1 2 3 4 5
```

File I/O

File Objects

- Built in type in Python: file
- You can read/write data to a file
- File can be created using the open function
  - E.g. open(filename, mode='rU')
  - A file object is returned
  - filename is a string containing the path to a file
  - mode denotes how the file is to be opened or created

File Modes

- Modes are strings, can be:
  - 'r': file must exist, open in read-only mode
  - 'w': file opened for write-only, file is overwritten if it already exists, created if not
  - 'a': file opened in write-only mode, data is appended if the file exists, file created otherwise
  - 'r+': file must exist, opened for reading and writing
  - 'w+': file opened for reading and writing, created if it does not exist, overwritten if it does
  - 'a+': file opened for reading and writing, data is appended, file is created if it does not exist
- Binary and text modes
  - The mode string can also have 'b' to 't' as a suffix for binary and text mode (default)
  - On Unix, there is no difference between opening in these modes
  - On Windows, newlines are handled in a special way, 'rU'

File Methods and Attributes

- Assume file object
- Common methods
  - f.close(), f.read(), f.readline(), f.readlines(), f.write(), f.writelines(), f.seek()
- Common attributes
  - f.closed, f.mode, f.name, f.newlines

Reading from File

- Say (for some inexplicable reason) you needed to parse this CSV file to extract Klang’s comments

```
#read the entire file into a list, one line per entry
lines = open(‘simpsons_diet.csv’).readlines()
#for each line / entry in the list
for line in lines:
    if line.startswith("Klang"):
        print line
f.close()
```

```
…
Lisa Simpson,Dinner,Tofu,4,Animals are cute!
Klang,Snack,Humans,40, *how*to*cook*for*forty?*humans*
Montgomery Burns,Snack, Life Extension Elixer,1,Excellent…
…
```

Closing can be done automatically by the garbage collector, but it is innocuous to call, and is often cleaner to do so explicitly.

Writing to File

```
f = open('sample2.txt', 'w')
for i in range(99, -1, -1):
    if i > 1 or i == 0:
        f.write('%d donuts in the box\n' % i)
    else:
        f.write('%d donuts in the box\n' % i)
    f.close()
```

- Notice how file objects have a readline() method, but not # a writeline() method, only a writelines() method which writes # lines from list.

CSV – often messy, always useful

- Comma-separated values
- Common file format, great for spreadsheets, small databases
- Basically:
  - Each record (think row in spreadsheets) is terminated by a line break, BUT
  - Line breaks may be embedded (and be part of fields)
  - Fields separated by commas
  - Fields containing commas must be encapsulated by double-quotes
  - Occasionally has a header line with column values
- Various formats – some quirky. Use the CSV Module

Reading a CSV File

- Use the csv module to create a reader instance
- Dialect, delimiter, quotechar (format parameters) are all optional
  - Controls various aspects of the CSV file
  - E.g. use a different delimiter other than a comma
- Reader is an iterator containing the rows of the CSV file
- Also available are dictionary readers which automatically parse headers, return an iterator over a sequence of dictionaries mapped to the rows

Reading with the CSV library

```
import csv
reader = csv.reader(open(‘simpsons_diet.csv’),
              delimiter=',', quotechar='“')
for row in reader:
    print row
```

```
…
[Barney Gumble,Breakfast,Duff Beer,1,I could’ve gone to Harvard]
[Duffman,Breakfast,Power bar,4,Duffman - can't breathe!]
[Duffman,Lunch,Protein shake,5,Duffman - has the power!]
…
```

Reading CSVs directly into variables

```
import csv
reader = csv.reader(open(r'c:\simpsons_diet.csv'), \
            delimiter=',', quotechar='"')
for char, meal, ate, quantity, comment in reader:
    print char + “ had “ + quantity + “ “ + ate + “ for “ + \
    meal
```

```
…
Lisa Simpson had 4 Tofu for Dinner
Klang had 40 Humans for Snack
Montgomery Burns had 1  Life Extension Elixer for Snack
```

Writing a CSV file

The writer function creates a CSV writer object, which converts values to strings and escapes them properly.

```
import csv
reader = csv.reader(open(r'c:\simpsons_diet.csv'), \
            delimiter=',', quotechar=‘"')
out  = open(r'c:\simpsons_new.csv', "wb")
writer = csv.writer(out, delimiter=',', quotechar='"')
for row in reader:
   writer.writerow(row)
new = ['Chronos', 'Snack', 'Klang', '1', 'The humans made him tasty!']
writer.writerow(new)
out.close()
```

```
...
Klang,Snack,Humans,40, *how*to*cook*for*forty?*humans*
Montgomery Burns,Snack, Life Extension Elixer,1,Excellent…
Chronos, Snack, Klang, The humans made him tasty!
```

Basic string manipulation

“Klang,Snack,Humans,40, *how*to*cook*for*forty?*humans*”

```
x= "Klang,Snack,Humans,40,*how*to*cook*for*forty?*humans*"
columns = x.split(",")
print columns
['Klang', 'Snack', 'Humans', '40', '*how*to*cook*for*forty?*humans*']
comment = columns [-1]
print comment
*how*to*cook*for*forty?*humans*
cleaned = comment.replace("*", " ").strip()
print cleaned [0].upper() + cleaned [1:] + "."
How to cook for forty? humans.
```

How about reading that CSV file from a network?

```
import urllib, csv


#grab our csv file from the network
url = http://some_path
simpsons = urllib.urlopen(url)


reader = csv.reader(simpsons, delimiter=',',
    quotechar='"')
for char, meal, ate, quantity, comment in reader:
    …
```

Homework II

Due before the start of class next week

Exercise 1: Producing a webpage from a CSV file

- Write a Python script which parses this CSV file: http://www.cs.columbia.edu/~joshua/teachng/cs3101/simpsons.csv
- and creates an HTML file describing the contents.
- Requirements:
  - Your script must use command line arguments to input the path to which the HTML file will be written.
  - Your script must either use Python's urllib library (see the online doc) to automatically download the CSV file prior to parsing, or must take it's location as a command line argument

This weeks extra credit

For a (very large) triangle in the form:

40

73 11

52 10 40

26 53 06 34

10 51 87 86 90

Compute the maximum path value from the upper most entry in the triangle to an entry on the bottom row. Legal moves are those which move exactly one entry adjacent from the present index. E.g., from the 2nd entry on the fourth row (in blue), you may access the 1st or 3rd entry on the fifth row.

http://www.cs.columbia.edu/~joshua/teachng/cs3101/triangle.txt